MSc thesis in Computer Science

Blockchain-Based Verifiable and Privacy-Preserving Machine Learning Inference

Mariana Samardžić

2023

MSc thesis in Computer Science

Blockchain-Based Verifiable and Privacy-Preserving Machine Learning Inference

Mariana Samardžić

July 2023

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Mariana Samardžić: *Blockchain-Based Verifiable and Privacy-Preserving Machine Learning Inference* (2023)
This work is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

The work in this thesis was carried out in the:

Cybersecurity research group
Delft University of Technology

Supervisors: Dr. Roland Kromes
Dr. Kaitai Liang
Prof.dr. George Smaragdakis

Abstract

The Machine Learning (ML) technology has taken the world by storm since it equipped the machines with previously unimaginable decision-making capabilities. However, building powerful ML models is not an easy task, but the demand for their utilization in different industries and areas of expertise is high. This was recognized by entities that have managed to create ML models and they started offering ML prediction services to clients in exchange for financial compensation. In this work, we explore how a ML predication service platform can be built in which we focus on two things: (1) privacy-preservation which entails keeping the client's datasets and service provider's ML models private and (2) inference verifiability ensuring that the ML prediction service providers do not commit fraud. The result are two platforms: ML Prediction Service Platform (MLPSP) which does not protect the secrecy of the client's datasets but offers model privacy and verifiability of the predictions and Input-Privacy ML Prediction Service Platform (IP-MLPSP) which protects the secrecy of the client's dataset and model privacy but the verifiability is probabilistic.

Acknowledgements

I would like to express my deepest gratitude and appreciation to all those who have contributed to the successful completion of this master's thesis directly or indirectly.

First and foremost, I am immensely grateful to my daily supervisor, Dr. Roland Kromes, who has devoted countless hours in guiding me in my first research work. I want to thank him for always being there when I needed help, for giving me valuable feedback and constructive criticism.

I extend my sincere thanks to my other supervisors, Dr. Kaitai Liang and Prof.dr. George Smaragdakis for their insightful comments, constructive criticism, and valuable suggestions.

I want to say thank you to the many great friends I have made during my studies: Dan Andreescu, Ion Babalau, Marin Duroyon, Andrei Geadau, Dan Plamadeala, Ioana Savu, Konrad Ponichtera, Natalia Struharova, Wessel Thomas and Bram Verboom. Your friendship has made my two years at the Delft University of Technology a truly fun and enjoyable experience.

Finally, I express my heartfelt appreciation to my fiancé Uros Dzinovic and my family for their unconditional love, unwavering support, and patience throughout my academic journey. Their belief in me and constant encouragement have been my pillars of strength.

Thank you all

Mariana

July 2023

Contents

1. Introduction **1**
- 1.1. Machine Learning . . . 1
- 1.2. ML Prediction Service . . . 1
- 1.3. ML Prediction Service Platform . . . 3
- 1.4. Key Contributions . . . 4
- 1.5. Organization of the Report . . . 4

2. Related work **5**
- 2.1. MVP . . . 5
- 2.2. Mutually Private Verifiable Machine Learning As-a-service: A Distributed Approach . . . 6
- 2.3. VPMLP . . . 6

3. Supported ML Models **7**
- 3.1. Operations . . . 7
- 3.2. Linear ML Models . . . 8
- 3.3. SVM . . . 9
  - 3.3.1. SVM with Linear Kernel . . . 10
  - 3.3.2. SVM with Polynomial Kernel . . . 11

4. Polynomial Commitment Scheme **13**
- 4.1. Prerequisites . . . 13
  - 4.1.1. Commitment Scheme . . . 13
  - 4.1.2. Pedersen Commitment Scheme . . . 14
  - 4.1.3. Zero-Knowledge Proof Systems . . . 15
  - 4.1.4. Bilinear Groups . . . 16
  - 4.1.5. Function Extensions . . . 16
- 4.2. Polynomial Encoding . . . 18
- 4.3. PolyCom . . . 21
  - 4.3.1. Linear Homomorphism . . . 21
  - 4.3.2. Extractability . . . 21
  - 4.3.3. $\mathrm{ZK}_{\mathrm{eq}}$ . . . 22
  - 4.3.4. $\mathrm{ZK}_{\mathrm{prod}}$ . . . 24
  - 4.3.5. Polynomial Commitment . . . 24
  - 4.3.6. Overview of PolyCom . . . 25
- 4.4. Converting a Vector into a Polynomial . . . 27

5. Proving correct ML inference computation **31**
- 5.1. LegoSNARK . . . 31
- 5.2. Overview of CP-SNARKs . . . 32
- 5.3. Fundamental CP-SNARKs . . . 34
  - 5.3.1. CP-Poly . . . 34

5.3.2. CP-Sumcheck . . . 37
5.4. CP-SNARKs for ML Operations . . . 39
5.4.1. CP-MM . . . 39
5.4.2. CP-EHad . . . 41
5.4.3. CP-ColumnSum . . . 42
5.4.4. CP-Expo . . . 43
5.4.5. CP-ScalarAdd . . . 45
5.5. CP-SNARKs for ML Inference . . . 46
5.5.1. CP-Linear . . . 46
5.5.2. CP-SVMLinear . . . 47
5.5.3. CP-SVMPoly . . . 47
5.6. Complexity Analysis . . . 47
5.7. Implementation and Experimental Evaluation . . . 49

6. ML Prediction Service Platforms **53**
6.1. Prerequisites . . . 53
6.1.1. Blockchain . . . 53
6.1.2. Homomorphic Encryption . . . 53
6.2. MLPSP . . . 54
6.2.1. Security Discussion . . . 58
6.3. IP-MLPSP . . . 60
6.3.1. Motivation . . . 60
6.3.2. Homomorphic Encryption of the Dataset . . . 60
6.3.3. Main Idea . . . 62
6.3.4. Design . . . 63
6.3.5. Security Discussion . . . 69

7. Discussion **71**
7.1. Comparison of IP-MLPSP with Related Work: Advantages . . . 71
7.1.1. Trust Assumptions . . . 71
7.1.2. Model Extraction Attack . . . 72
7.1.3. Batch Verification . . . 72
7.2. Comparison of IP-MLPSP with Related Work: Disadvantages . . . 72
7.3. Future Work . . . 73

8. Conclusion **75**

A. Pedersen Commitment Scheme **77**

B. ZK_{eq} **79**

C. ZK_{prod} **81**

D. CP-MM **83**

E. CP-EHad **85**

F. CP-Expo **87**

G. CP-ColumnSum **91**

H. CP-ScalarAdd 93

I. CP-Linear 95

J. CP-SVMLinear 97

K. CP-SVMPoly 99

List of Figures

4.1. Example execution of the polynomial evaluation algorithm 20
4.2. ZK_{eq} 23
4.3. ZK_{eq}' 23
4.4. ZK_{prod} 24
4.5. PolyCom 26
4.6. The MLE algorithm 29

5.1. Prover algorithm for CP-Poly 34
5.2. Verifier algorithm for CP-Poly 35
5.3. Example Execution of the Polynomial Decomposition Algorithm 36
5.4. Prover algorithm for CP-Sumcheck 38
5.5. Verifier algorithm for CP-Sumcheck 39

6.1. MLPSP Data Types 55
6.2. MLPSP Methods 55
6.3. MLPSP Platform Setup 56
6.4. MLPSP Model Registration 56
6.5. MLPSP Request Registration 57
6.6. MLPSP Proof Registration 57
6.7. MLPSP Verification 58
6.8. IP-MLPSP Data Types 64
6.9. IP-MLPSP Request Registration 65
6.10. IP-MLPSP Output Registration 66
6.11. IP-MLPSP Index Registration 67
6.12. IP-MLPSP Proof Registration 68
6.13. IP-MLPSP Verification 69

D.1. CP-MM.Prove 83
D.2. CP-MM.Verify 84

E.1. CP-EHad.Prove 85
E.2. CP-EHad.Verify 86

F.1. CP-Expo.Prove 88
F.2. CP-Expo.Verify 89

G.1. CP-ColumnSum.Prove 91
G.2. CP-ColumnSum.Verify 92

H.1. CP-ScalarAdd.Prove 93
H.2. CP-ScalarAdd.Verify 93

I.1. CP-Linear.Prove 95

I.2. CP-Linear.Verify 96

J.1. CP-SVMLinear.Prove 97
J.2. CP-SVMLinear.Verify 98

K.1. CP-SVMPoly.Prove 100
K.2. CP-SVMPoly.Verify 101

List of Tables

3.1. Linear ML model inference operations 9
3.2. SVM linear kernel inference operations 10
3.3. SVM polynomial kernel inference operations 11

5.1. Overview of CP-SNARKs 33
5.2. Complexity: CP-Poly, CP-Sumcheck, ZK_{eq}, ZK_{prod}, l is the number of variables in the polynomial 48
5.3. Complexity: (1)CP-MM, (2)CP-EHad, (3)CP-Expo, (4)CP-ColumnSum, (5)CP-ScalarAdd 48
5.4. CP-Linear Runtime analysis 50
5.5. CP-SVMLinear Runtime analysis 51
5.6. CP-SVMPoly Runtime analysis 52

6.1. MLPSP Security Requirements 59
6.2. IP-MLPSP Security Requirements 70

List of Algorithms

1. Polynomial Evaluation Algorithm 19
2. MLE Algorithm 28
3. Polynomial Decomposition Algorithm 36

Acronyms

API Application Programming Interface . . . 2
CP-SNARK Commit-and-Prove Zero-Knowledge Succinct Non-Interactive Argument of Knowledge . . . 31
CP Commit-and-Prove . . . 31
HCPS Human Cyber-Physical System . . . 6
IoT Internet of Things . . . 6
IP-MLPSP Input-Privacy ML Prediction Service Platform . . . v
MLPSP ML Prediction Service Platform . . . v
MLE Multilinear Extension . . . 17
ML Machine Learning . . . v
RBF Radial Basis Function . . . 5
SVM Support Vector Machines . . . 1
TTP Trusted Third Party . . . 5
ZK-NARK Zero-Knowledge Non-Interactive Argument of Knowledge . . . 15
ZK-SNARK Zero-Knowledge Succinct Non-Interactive Argument of Knowledge . . . 5

1. Introduction

1.1. Machine Learning

ML [1] is essentially a new way of programming which has taken the world by storm. Instead of supplying the machine with detailed step-by-step instructions on how to achieve a certain goal, the machine is supplied with a set of desirable input-output pairs of the program and the machine creates the program itself. That is why we call this process machine learning. Due to the fact that in many different use cases it is easier to train a system by giving it examples of desired input-output behaviours than to program the system manually, machine learning has found its place in many different areas of industry and science. In healthcare, for example, machine learning is used to detect and diagnose diseases from medical images as shown in [2]. In finance, machine learning models are used for evaluating creditworthiness and predicting default risk [3]. Machine learning algorithms have also found their place in retail and e-commerce where they are used for personalized product recommendations based on user behavior and preferences [4].

In a nutshell, a machine is given a set of desirable input-output pairs called the training set. The machine is running a ML algorithm that creates a ML model from the training set. This phase is called ML training. Once the model is created or ”trained”, it can then be supplied with new inputs to obtain new outputs, which are often called predictions. This phase is called ML inference. If calculating predictions is a linear function then the ML model is called linear, otherwise it is called non-linear. Many different ML algorithms have been developed such as Support Vector Machines (SVM) [5], convolutional neural networks [6] , decision trees [7], random forest [8] and many more. In this work, we will focus on general linear ML models and SVM models and we will provide more details on them later on.

In general, to develop a powerful machine learning model, a company should have access to a set of resources. First, the company has to have a large representative training set. Secondly, the company needs skilled ML professionals who have to perform different tasks such as: preprocessing the training set, choosing which ML algorithm to use based on the training set and the goal and tuning the hyper-parameters of the algorithm. Lastly, the company has to have access to sufficient computational resources. Not all companies are in the position to acquire all of these resources and develop their own models. However, they would still like to use the ML technology to improve their business and stay competitive in the market. The solution for them could be to use ML prediction services.

1.2. ML Prediction Service

Some companies, institutions or other organizations that were able to create powerful ML models might want to offer prediction services on their models to generate profit. In other words, they will compute predictions using their own models on someone else’s dataset

in exchange for financial compensation. For example, Google offers such services on the Google Cloud AI Platform [9], Amazon on Amazon Web Services Marketplace [10] and Microsoft on the Microsoft Azure Marketplace [11].

There are two types of users in a ML prediction service: a service provider and a client. The service provider owns a trained machine learning model and the client owns a dataset. The client sends the dataset to the service provider and the service provider evaluates the dataset on its model and returns the predictions to the client in exchange for money. We can identify three main security requirements in such system:

1. **Model privacy:** To create a highly accurate machine learning model, a large training set first needs to be assembled and the data in the training set needs to be preprocessed. Then, machine learning experts have to choose the most suitable machine learning algorithm and tune the hyper-parameters of the algorithm. Powerful models, like neural networks, require vast computational resources in order to be trained and the training can be time-consuming. Furthermore, once the model is trained and ready for making predictions, the company has to create an Application Programming Interface (API) and provide support for concurrent requests from clients. All these factors indicate that a company has to invest many resources in developing a ML model. The trained machine learning model has great commercial value and some greedy clients and/or competitors may try to steal it to perform the predictions for free or to make a profit themselves. If the model privacy is not guaranteed, model owners will not offer prediction services.
2. **Input privacy:** To evaluate the dataset on a model, the client has to send the dataset to the service provider. However, the dataset can contain sensitive data like medical records or personal information of the client company's users that the client cannot reveal to the service provider. Even if the client trust the service provider, the client cannot share its users' data with other third parties without the knowledge or agreement of its users. Therefore, input privacy is a requirement for clients that want to perform predictions on sensitive data, in order to use ML prediction services.
3. **Outcome Verifiability:** The returned predictions might be faulty for three reasons. Firstly, the faults could be non-intentional like software/hardware bugs or external attacks. Secondly, the service provider has the incentive to return a random prediction back if that way the provider will not have to spend its resources computing and the client will not be able to tell the difference between a random and a computed prediction. Thirdly, the client might be a competitor of the service provider and the provider might want to sabotage the client's business by returning some false predictions. If the client is using these faulty predictions in some integrity-sensitive applications like risk assessment, investment suggestions or disease diagnosis, then the damage that these faulty predictions could cause is substantial. *Hence, having outcome verifiability is a basic requirement for many clients that are using ML prediction services in integrity-sensitive applications. Even if the applications are not integrity-sensitive, every client can only benefit from having a proof of correct prediction computation. By having this proof, the clients can be sure that they got what they payed for.*

The main challenges of building a ML prediction service that satisfies these security requirements have been identified in [12] which is the first work that dealt with this topic. Firstly, outcome verifiability and model privacy seem to be contradicting tasks. In standard verifiable computation schemes [13], the client that verifies if the outsourced computation is correct, has access to the outsourced function. In a ML prediction service, the outsourced

function is the ML model. Since we want to have model privacy, the verifier cannot have access to it. Secondly, the input privacy makes the outcome verifiability an even harder task. To assure input privacy, the data can be encrypted using a homomorphic encryption scheme [14] and then the computation can be performed on an encrypted dataset. However, most verifiable computation schemes assume that the input data is in plaintext and it is not straightforward for these schemes to work with encrypted data. Lastly, the third challenge is to verify the integrity of the massive underlying operations on the client's dataset.

1.3. ML Prediction Service Platform

The goal of this research is to create a ML Prediction Service Platform which allows service providers to register their models for which they offer prediction services. Clients can browse the registered models and choose the models whose prediction services they want to purchase. The platform takes care that the returned predictions are correct and that the payments are fair. We have identified the security requirements that we want to achieve:

1. **Model Privacy:** The service provider's machine learning model should stay secret.
2. **Input Privacy:** The client's dataset should stay secret.
   a) **Output Privacy:** Only the client can see the returned predictions.
3. **Outcome Verifiability:** The service provider has to generate a proof of correct ML inference computation.
   a) **Model Registration:** Outcome verifiability assures that the service provider cannot generate a proof of correct ML inference if it has not actually done the computation. However the service provider could evaluate the client's data on some other model instead of the model that the client has chosen. It can be that the client is the service provider's competitor and the service provider wants to sabotage the client's business by returning a prediction from a less accurate model. A ML Prediction Service Platform should assure that this scenario cannot happen.
   b) **Dataset Registration:** A malicious service provider could evaluate some other dataset instead of the client's dataset and still generate a proof of executing an ML inference. A ML Prediction Service Platform should assure that the proof can only be generated for the correct dataset.
   c) **Matching Output:** A malicious service provider could calculate the correct predictions and generate a proof of correct computation but still return some other, fake predictions. Perhaps, the client is a competitor and the service provider wants to sabotage it. A ML Prediction Service Platform should assure that the returned predictions are the same predictions that have been used in the proof generation.
4. **Batch verification:** Usually the client's dataset will consist of more than one data instance. It is important that the scheme supports proof generation for the entire dataset instead of generating one proof for every instance in the dataset.
5. **Fair payments:** The client should only pay for the predictions that are correct and the client should only get the correct predictions after the payment.

6. **Trustless system:** An easy way to implement this platform would be to have a trusted third party or a trusted execution environment do the inference. However, in that case, we would have to ask from the users to place their trust in those entities. That would mean that users are actually risking the leakage of their models, leakage of their datasets and the generation of false proofs if those entities turn out to be non-trustworthy. Not many users are ready to take those risks. A better approach would be to have a platform where such risk is non-existent or at least a platform where the severity of the damage caused by a malicious trusted entity is decreased.

1.4. Key Contributions

The key contributions of this work are the following:

1. We have created three protocols: one for general linear ML models, one for linear-kernel SVM models and one for polynomial-kernel SVM models that allow a service provider to generate a publicly verifiable proof that the ML inference has been correctly computed on a plaintext dataset without leaking the secret ML model parameters. We have built these protocols using the LegoSNARK[15] framework. We also provide an implementation and evaluation.
2. We have built these protocols by combining protocols which can be used for proving the correct computation of some basic operations. In the LegoSNARK framework these protocols are called gadgets. We have used some gadgets provided in the LegoSNARK framework and we ourselves have created four new gadgets that we describe in Section 5.4. These gadgets are reusable and can be used in proving the correct computations of other programs.
3. We have created two ML prediction service platforms where the outcome verifiability is assured by using the protocols we have created. The first one, simply called MLPSP, satisfies all requirements except of the input privacy and the second one, called IP-MLPSP, also satisfies the input privacy requirement, but at the expense of the outcome verifiability being probabilistic. We also show that our IP-MLPSP has the least amount of trust assumptions compared to related work.

1.5. Organization of the Report

The report is organized as follows: In Chapter 2 we give an overview of related work. In Chapter 3 we outline all operations that are part of ML inference computation for linear models, linear-kernel SVM models and polynomial-kernel SVM models. In Chapter 4 we explain how we implemented the polynomial commitment scheme from [15], which is a building block of our verifiable computation protocols. Next, in Chapter 5 we present how we have built the scheme for proving the correct computation of ML inference. In Chapter 6 we describe our two ML prediction service platforms. Finally, in Chapter 7 we provide a discussion about our results including a comparison with state-of-the-art related work and the planned future work and in Chapter 8 we give our conclusion.

2. Related work

The related work can be split into three categories. The first category focuses on privacy-preserving ML-as-a-service, or in other words, on protecting the model and the input privacy. One of the most notable works is Cryptonets. CryptoNets [16] allows a service provider to evaluate an encrypted dataset with a neural network and return high throughput, accurate, and encrypted predictions back. The second category focuses on the verifiability of outsourced ML tasks. One of the most notable works in this category is SafetyNets [17] whose system model consists of a client that outsources the ML inference computation to a cloud provider. The cloud provider computes the predictions on behalf of the client and provides a client with a short mathematical proof of the correctness of computation. However, it does not protect the input privacy. Another notable work is vcNN [18] which is an efficient verifiable convolutional neural network framework. This work also does not focus on protecting the privacy of the data. VeriML [19] can be used when a client wants to outsource ML tasks to an untrusted server. By implementing fair payments, they ensure that the claimed resource consumption by the service provider corresponds to the actual workload. To enhance efficiency, they utilize Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARK) (more information will be given in Section 5.1) on randomly-selected iterations during the machine learning training phase. The probabilistic assurance is tunable like in our work. The last category would be works that, just like our work, try two combine these two concepts: privacy-preservation and verifiability. However, there are not many works in this category. We have identified three main works and we will give an overview of them in the following sections and we are going to compare them to our work in the Discussion chapter.

2.1. MVP

The scheme MVP proposed in [12] is the first scheme to achieve outcome verifiability, model privacy and input privacy simultaneously in ML prediction services while not relying on any trusted hardware. Their system model is very similar to ours: there is a service provider, a client and a Trusted Third Party (TTP). The client wants to get a verifiable prediction from the service provider's model while keeping its dataset and the model secret. MVP supports SVM with linear, polynomial and Radial Basis Function (RBF) kernel. In a nutshell, they have created two verifiable and privacy-preserving schemes: one for the dot product computation and one for the euclidean square distance computation. If the SVM model has a polynomial kernel $K(x_j, z_i) = (x_j^T z_i + c)^d$,where x is a support vector, z a data instance and c, d kernel parameters, the service provider uses the dot product scheme to compute the dot product for every data instance-support vector pair $\{V_j^{(i)} = x_j^T z_i | j \in \mathcal{SV}\}$. This result gets decrypted by a TTP and sent back to the client, who completes the kernel function on its own $(V_j^{(i)} + c)^d$. If the model has the RBF kernel $K(x_j, z) = exp(-\gamma||x_j - z_i||^2)$, the the provider uses the

euclidean square distance scheme to compute $\{V_j^{(i)} = ||x_j - z_i||^2 | j \in \mathcal{SV}\}$ for every data instance-support vector pair. The result gets decrypted by the TTP and sent to the client who completes the kernel function $exp(-\gamma V_j^{(i)})$. The verifiability proof relies on the **Polynomial Decomposition** lemma which is also at the core of the LegoSNARK framework, which we use for verifiability.

2.2. Mutually Private Verifiable Machine Learning As-a-service: A Distributed Approach

In "Mutually Private Verifiable Machine Learning As-a-service: A Distributed Approach" [20], the authors also use the LegoSNARK framework and they present a protocol only for linear kernel SVMs. The architecture of their system is different than ours. The entities in their system are:

1. **ML service provider (honest-but-curious):** Owns a ML model but outsources the inference computation to three different cloud providers by partitioning the model vertically into three parts which allows every cloud provider to compute a specific sub-computation of the ML inference computation. The ML service provider wants to be sure that the cloud providers do the computations correctly and that the client does not steal the model.
2. **Cloud providers P1, P2 and P3 (malicious):** Every cloud provider consists of multiple servers. The cloud providers already know the model, but they would also like to steal the client's input data. They would also like to return some faulty predictions back.
3. **Client:** To avoid sending its dataset in plaintext to P1 in order to perform the first sub-computation, the client encodes its dataset in a way that every server in P1 gets a different variation of the dataset on which they can perform their sub-computation. The servers in P2 aggregate the outputs from P1 and are able to extract the true output of the first sub-computation for further processing. The client trusts that the ML service provider will make sure that the cloud providers do the correct computations.

2.3. VPMLP

The authors of [21] have created VPMLP, a Verifiable Privacy-Preserving Machine Learning Prediction Scheme for Edge-Enhanced Human Cyber-Physical System (HCPS). In their system, the cloud server (CS) has a trained linear ML model. To support many request from User Terminals (UT),which collect data from Internet of Things (IoT) devices and decrease the response latency, the CS outsources the inference computation to Edge Servers (ES) by creating a different masked model for every ES. The UT sends a prediction request that contains an encrypted dataset to the closest available ES and the ES together with the CS calculates the prediction. The UT can ask the CS for a proof of correct computation. However, the UT assumes that the CS and ES are honest and compute the prediction correctly. The purpose of the proof is to make sure that no external adversary has tempered with the data shared between the UT and the prediction service.

3. Supported ML Models

Our ML prediction service platforms are suitable for working with linear ML models and SVM models that have either a linear or a polynomial kernel. The ML inference computation can be divided into several sub-computations. We will prove the correct computation of ML inference by proving the correct computation of the different sub-computations and linking them together. In Section 3.1 we have defined 5 sub-computations that are part of ML inference computation for the aforementioned models. In Section 3.2 we give an overview of linear ML model inference and in Section 3.3 we give an overview of SVM inference.

3.1. Operations

Definition 3.1.1 (Matrix Multiplication). Matrix Multiplication $mm(A, B) = C$ is a function that takes as input an $m \times n$ matrix A and an $n \times k$ matrix B and outputs an $m \times k$ matrix C such that $\forall i \in \{0, ..., m-1\}, j \in \{0, ..., k-1\} : C_{i,j} = \sum_{l=0}^{n-1} A_{i,l} \cdot B_{l,j}$.

Example 1.

Given $A = \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix}$ and $B = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$ then $mm(A, B) = \begin{bmatrix} 18 \\ 33 \end{bmatrix}$

Definition 3.1.2 (Extended Hadamard Product). Extended Hadamard Product $ehad(A, B) = C$ is a function that takes as input a $m \times n$ matrix A and a vector B of size m and outputs an $m \times n$ matrix C such that $\forall i \in \{0, ..., m-1\}, j \in \{0, ..., n-1\} : C_{i,j} = A_{i,j} \cdot B_i$.

Example 2.

Given $A = \begin{bmatrix} 3 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$ then $ehad(A, B) = \begin{bmatrix} 6 & 4 \\ 15 & 20 \end{bmatrix}$

Definition 3.1.3 (Matrix Exponentiation). Matrix exponentiation $expo(A, d) = B$ is a function that takes as input a $m \times n$ matrix A and a positive integer d and outputs am $m \times n$ matrix B such that $\forall i \in \{0, ..., m-1\}, j \in \{0, ..., n-1\} : B_{i,j} = A_{i,j}^d$.

Example 3.

$$\text{Given } A = \begin{bmatrix} 3 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } d = 2 \text{ then } expo(A, d) = \begin{bmatrix} 9 & 4 \\ 9 & 16 \end{bmatrix}$$

Definition 3.1.4 (Column Summation). Column Summation $cs(A) = B$ is a function that takes as input an $m \times n$ matrix A and outputs a vector B of size n such that $\forall j \in \{0, ..., n-1\}$: $B_j = \sum_{i=0}^{m-1} A_{i,j}$.

Example 4.

$$\text{Given } A = \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \text{ then } cs(A) = \begin{bmatrix} 7 & 10 \end{bmatrix}$$

Definition 3.1.5 (Scalar Addition). Scalar addition $sa(A, C) = B$ is a function that takes as input an $m \times n$ matrix A and a scalar value C and outputs an $m \times n$ matrix B such that $\forall i \in \{0, ..., m-1\} \wedge \forall j \in \{0, ..., n-1\} : B_{i,j} = A_{i,j} + C$.

Example 5.

$$\text{Given } A = \begin{bmatrix} 3 & 4 & 2 & 6 \end{bmatrix} \text{ and } C = 3 \text{ then } sa(A, C) = \begin{bmatrix} 6 & 7 & 5 & 9 \end{bmatrix}$$

3.2. Linear ML Models

Linear machine learning models are a class of ML models that assume a linear relationship between the input features and the target variable. If there are n input features, then the result of the training phase is a model that consists of a weight vector w of size n and a bias value b. The target variable t of a new data instance z is calculated as $t = w^T z + b$. The output of the decision function can be interpreted differently depending on the specific linear model being used. For example, in linear regression [22], the decision function represents the predicted continuous value. In logistic regression [22], the decision function is typically passed through a sigmoid function to obtain the predicted probability of belonging to a particular class. We will give a formal definition of the linear ML model inference function that makes predictions on a dataset Z.

Definition 3.2.1 (Linear ML inference). $f_{Linear}(model, Z) = o$ is a function that takes as input a tuple $model = (w, b)$ where w is a vector of size n and b is a scalar value and an $n \times k$ matrix Z, where each column i represents a data instance z_i, and outputs a vector o of size k where $\forall i \in \{0, .., k-1\} : o_i = w^T z_i + b$

	operation	$\forall j \in \{0,...,n-1\}, i \in \{0,...,k-1\}$
1.	$ehad(Z,w) = H$	$H_{j,i} = Z_{j,i} \cdot w_j$
2.	$cs(H) = s$	$s_i = \sum_{j=0}^{n-1} H_{j,i} = w^T z_i$
3.	$sa(s,b) = o$	$o_i = s_i + b = w^T z_i + b$

Table 3.1.: Linear ML model inference operations

This computation can be divided into 3 sub-computations as can be seen in Table 3.1. First we perform the extended hadamard product between the matrix Z and vector w and obtain the matrix H. Then, we perform a column summation of the matrix H and obtain the vector s. Finally we perform scalar addition between the vector s and scalar b and obtain the output vector o.

3.3. SVM

Support vector machines are supervised machine learning models which can be used to create a two class classifier. Given a training dataset, and for every training point an output label 1 or -1 which defines to which of the two classes the point belongs to, the SVM algorithm finds the hyperplane that separates the training data by a maximum margin. More precisely, the algorithm finds the vector α that contains the Lagrangian multiplier value for every training point. The training points whose Lagrangian multiplier values are $\alpha \neq 0$ lie closest to the margin and are called support vectors.

Given that there are m support vectors, for every $j \in \{0,...,m-1\}$, we will mark with x_j the support vector j,with $t_j \in \{-1,1\}$ the label of the support vector x_j and with α_j the Lagrangian multiplier for support vector x_j. A new data instance z can be classified by performing the following computation:

$$f_{SVM}(z) = sign(\sum_{j=0}^{m} \alpha_j t_j (x_j^T z) + b)$$

If the training dataset is not linearly separable, the dataset and the support vectors can be mapped using a non-linear map Φ into higher dimensions in which the training set is linearly separable.

$$f_{SVM}(z) = sign(\sum_{j=1}^{m} \alpha_j t_j \langle \Phi(x_j), \Phi(z) \rangle + b)$$

However, by the use of kernels, no computations have to actually be performed in that high-dimensional space, rather, all computations are performed in the input space. A kernel function K has the property that $K(x_j, z) = \langle \Phi(x_j), \Phi(z) \rangle$ for a certain map Φ. There are various kernel functions that can be used. For example:

- Linear kernel $K(x_j, z_i) = x_j^T z_i$

- Polynomial kernel $K(x_j, z_i) = (x_j^T z_i + c)^d$
- RBF kernel $K(x_j, z_i) = exp(-\gamma||x_j - z_i||^2)$

Thus, the decision function can then be rewritten as:

$$f_{SVM}(z) = sign(\sum_{j=1}^{m} \alpha_j t_j K(x_j, z) + b)$$

3.3.1. SVM with Linear Kernel

An SVM model with a linear kernel is represented as a tuple $model = (X, Y, B)$. X is an $m \times n$ matrix where m is the number of support vectors and n is the number of features. Every row j of X represents the support vector x_j .The vector Y of length m is defined such that every element $Y_j = t_j \cdot \alpha_j$. The value B corresponds to the bias value. The dataset that should be evaluated is represented by an $n \times k$ matrix Z, where each column i represents a data instance z_i. The prediction of data instance z_i is equal to $f(z_i) = sign(\sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i) + b)$. In our work, we assume that the service provider computes $f(z_i) = \sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i) + b$ and sends that value to the client, who then does the sign operation on its own. Next, we will give a formal definition of the linear-kernel SVM inference.

Definition 3.3.1 (Linear-Kernel SVM inference). $f_{SVMLinear}(model, Z) = O$ is a function that takes as input a tuple $model = (X, Y, B)$, where X is an $m \times n$ matrix and x_j is the j-th row of X, Y is a vector of length m and Y_j is the j-th element of Y and B is a scalar value and an $n \times k$ matrix Z where z_i is the i-th column of Z and outputs a vector O of size k where $\forall i \in \{0, .., k-1\} : O_i = \sum_{j=0}^{m-1} Y_j \cdot (x_j^T z_i) + B$

This function can be divided into 4 distinct sub-computations as can be seen in Table 3.2.

	operation	$\forall j \in \{0, ..., m-1\}, i \in \{0, ..., k-1\}$
1.	$mm(X, Z) = M$	$M_{j,i} = x_j^T z_i$
2.	$eHad(M, Y) = H$	$H_{j,i} = Y_j M_{j,i} = t_j \alpha_j (x_j^T z_i)$
3.	$cs(H) = S$	$S_i = \sum_{j=0}^{m-1} H_{j,i} = \sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i)$
4.	$sa(S, B) = O$	$O_i = S_i + B = \sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i) + B$

Table 3.2.: SVM linear kernel inference operations

3.3.2. SVM with Polynomial Kernel

An SVM model with a polynomial kernel is represented as a tuple $model = (X, Y, B, C, d)$. X is a $m \times n$ matrix where m is the number of support vectors and n is the number of features. Every row j of X represents the support vector x_j .The vector Y of length m is defined such that every element $Y_j = t_j \cdot \alpha_j$. The value B corresponds to the bias value. C is a scalar value which is a parameter of the kernel and d is a positive integer. The dataset that should be evaluated is represented by a $n \times k$ matrix Z, where each column i represents a data instance z_i. The prediction of data instance z_i is equal to $f(z_i) = sign(\sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i + c)^d + b)$. Just like in the linear-kernel version, the service provider does all the operations except of the sign operation. We will give a formal definition of the polynomial-kernel SVM inference.

Definition 3.3.2 (Polynomial-Kernel SVM inference). $f_{SVMPoly}(model, Z) = O$ is a function that takes as input a tuple $model = (d, X, C, Y, B)$, where X is a $m \times n$ matrix X and x_j is the j-th row of X, Y is a vector of length m and Y_j is the j-th element of Y, B and C are scalar values and d is a positive integer and an $n \times k$ matrix Z and outputs a vector O of size k where $\forall i \in \{0, .., k-1\} : O_i = \sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i + C)^d + B$

This function can be divided into 6 sub-computations as can be seen in Table 3.3.

	operation	$\forall j \in \{0, ..., m-1\}, i \in \{0, ..., k-1\}$
1.	$mm(X, Z) = M$	$M_{j,i} = x_j^T z_i$
2.	$sa(M, C) = \Gamma$	$\Gamma_{j,i} = M_{j,i} + C = x_j^T z_i + C$
3.	$expo(\Gamma, d) = E$	$E_{j,i} = \Gamma_{j,i}^d = (x_j^T z_i + C)^d$
4.	$eHad(E, Y) = H$	$H_{j,i} = Y_j E_{j,i} = t_j \alpha_j (x_j^T z_i + C)^d$
5.	$cs(H) = S$	$S_i = \sum_{j=0}^{m-1} H_{j,i} = \sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i + C)^d$
6.	$sa(S, B) = O$	$O_i = S_i + B = \sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i + C)^d + B$

Table 3.3.: SVM polynomial kernel inference operations

4. Polynomial Commitment Scheme

Let us say that a service provider owns an $m \times n$ matrix A and a client owns an $n \times k$ matrix B. The client sends B to the service provider and the service provider evaluates $mm(A, B) \rightarrow C$ and returns C to the client together with a proof of correct computation π. A ML prediction service platform is supposed to verify that the proof π is valid and that it is linked to the values A, B and C. However, the platform does not have access to these values because of the input and model privacy requirements. The question is how can we still verify that π is connected to A, B and C even though these three values cannot be seen. The solution is to use a commitment scheme.

Our proofs of correct computations are based on the proofs from the LegoSNARK framework [15] that are defined for operations over vectors committed using a polynomial commitment scheme from [23]. Authors of LegoSNARK have named this polynomial commitment scheme PolyCom. In this section we will explain what commitments are, how PolyCom is specified and how we have implemented it.

4.1. Prerequisites

4.1.1. Commitment Scheme

A commitment scheme is a cryptographic protocol that allows a user to commit to a value that will be revealed later. No one will be able to see the committed value until the user reveals it and the user will not be able to reveal another value other than the value that was committed. We will present the definition of a commitment scheme from [15]. A commitment scheme is a tuple of algorithms *Com=(Setup,Commit,VerCommit)* that are defined as follows:

1. $Setup(1^\lambda) \rightarrow ck$: Given a security parameter λ, outputs a commitment key ck that includes the descriptions of the input space $\mathcal{D}$, commitment space $\mathcal{C}$ and the opening space $\mathcal{O}$.
2. $Commit(ck, u) \rightarrow (c, o)$: Takes the commitment key and a value $u \in \mathcal{D}$ that should be committed and outputs a commitment $c \in \mathcal{C}$ and an opening $o \in \mathcal{O}$.
3. $VerCommit(ck, c, u, o) \rightarrow b \in \{0, 1\}$: Once the committer wants to reveal the committed value, the committer reveals the input u along with the opening o. The verifier can verify that that u is the value committed in c using the opening o by running the algorithm $VerCommit$.

The commitment scheme Com has to satisfy the notions of:

1. **Correctness:** For every $\lambda \in \mathbb{N}$ and every $u \in \mathcal{D}$ it should hold that if $ck \leftarrow Setup(1^\lambda)$ and $(c, o) \leftarrow Commit(ck, u)$ then $Pr(VerCommit(ck, c, u, o) = 1) = 1$.

2. **Binding**: For every polynomial-time adversary $\mathcal{A}$ it should hold that if $ck \leftarrow Setup(1^\lambda)$ and $(c, u, o, u', o') \leftarrow \mathcal{A}(ck)$ then $Pr(VerCommit(ck, c, u, o) = 1 \wedge VerCommit(ck, c, u', o') = 1 \wedge u \neq u') = negl$. In other words, an adversary can commit to one value u and later reveal another value u' only with negligible probability.

3. **Hiding** For $ck \leftarrow Setup(1^\lambda)$ and every two values $u, u' \in \mathcal{D}$, we require that $Com(ck, u)$ and $Com(ck, u')$ are statistically indistinguishable. In other words, even a computationally unbounded adversary cannot see which value is committed in a commitment c.

4.1.2. Pedersen Commitment Scheme

The Pedersen commitment scheme is a popular commitment scheme introduced in [24]. The algorithms $(Setup, Commit, VerCommit)$ are defined as:

1. $Setup(1^\lambda) \rightarrow ck$

   a) Generate a group $\mathbb{G}$ whose order is a λ-bit prime q. The discrete log problem should be hard in $\mathbb{G}$.

A way to generate $\mathbb{G}$ is to generate a large prime p such that $p-1$ is divisible by q. Then Z_p^* has order $p-1$ and since the order of every subgroup divides the order of the group, there is a subgroup $\mathbb{G}$ of order q.

   b) Generate two random generators g and h of $\mathbb{G}$ such that no one knows $dlog_g h$.

A way to generate g is to perform the following operations: i. $r \leftarrow_R Z$. ii. $f \leftarrow H(r) \in \mathbb{F}_p^*$ for some cryptographic hash function H. iii. $g \leftarrow f^{(p-1)/q} \pmod{p}$. iv. If $g = 1$ then return to (i), else output (r, g) Repeat the same process for h. Given r every user can check if a generator was generated randomly.

   c) Output $ck = (q, \mathbb{G}, g, h)$

2. $Commit(ck, u) \rightarrow (c, o)$

   Generate a random opening $o \in_R Z_q$ and then calculate the commitment c as $g^u h^o$.

3. $VerCommit(ck, c, u, o) \rightarrow b \in \{0, 1\}$

   Return 1 if $c = g^u h^o$ else return 0.

The Pedersen commitment scheme satisfies the correctness, binding and hiding properties as can be seen in Appendix A.

4.1.3. Zero-Knowledge Proof Systems

The following definitions are adapted from [25] and [15] and the reader is referred to these works for more information.

Definition 4.1.1 (Relation). $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ is a family of relations R on pairs (x, w). The value x is called the public statement and w is called the witness. If R holds on (x, w), we write $R(x, w) = 1$ else we write $R(x, w) = 0$. Deciding if a relation holds takes polynomial time.

Informally, a relation can be seen as a set of valid statement-witness pairs.

Definition 4.1.2 (Σ protocol). A Σ protocol for $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ is a tuple of algorithms $(\mathcal{P}, \mathcal{V})$ that allows a prover to convince a verifier that he knows a pair (x, w) such that a relation $R(x, w)$ holds without revealing the witness w. It is a 3-move interactive protocol that works as follows:

1. $\mathcal{P}(x, w) \rightarrow a$: The prover computes an initial message and sends it to the verifier
2. $S \rightarrow c$: The verifier picks a challenge c from a large set S
3. $\mathcal{P}(c) \rightarrow z$: The prover computes the response z using the challenge c.
4. $\mathcal{V}(x,(a, c, z)) \rightarrow b \in \{0, 1\}$ The verifier checks the transcript (a, c, z) and returns 1 if the transcript is valid and 0 otherwise.

Definition 4.1.3 (Public coin Σ protocol). A Public coin Σ protocol is a Σ protocol where the verifier picks the challenge uniformly at random and independently from the initial message. A public-coin Σ protocol satisfies the notions of:

1. **Completeness:** For every pair (x,w) that satisfies the relation R, the verifier will accept the proof.
2. **Special Soundness:** Given two accepting transcripts for the same pair (x, w) with distinct challenges and same initial message it is possible to extract the witness.
3. **Special Honest Verifier Zero-Knowledge:** There should be a simulator that given a challenge, can simulate an accepting transcript without knowing a witness. This might sound counter-intuitive however, the simulator is less restricted then an adversary since it can create messages in different order, more specifically, it can first create the response z and then the initial message a.

Definition 4.1.4 (Zero-knowledge non-interactive argument of knowledge). A Zero-Knowledge Non-Interactive Argument of Knowledge (ZK-NARK) for $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ is a tuple of algorithms $\Pi = (KeyGen, Prove, VerProof)$ that allows a prover to convince a verifier that he knows a pair (x, w) such that a relation $R(x, w)$ holds without revealing the witness w. The protocol works as follows:

1. $KeyGen(R) \rightarrow crs = (ek, vk)$: Takes a relation as input and outputs a common reference string crs which consists of an evaluation key ek and a verification key vk.
2. $Prove(ek, x, w) \rightarrow \pi$: The prover generates the proof π
3. $VerProof(vk, x, \pi) \rightarrow b \in \{0, 1\}$. The verifier checks the proof and returns 1 if the proof is correct, else returns 0

A ZK-NARK has the following properties:

1. **Completeness:** For every pair (x,w) that satisfies the relation R, the verifier will accept the proof
2. **Soundness:** If the prover does not know the witness, then the verifier will accept the proof with negligible probability.
3. **Zero Knowledge:** There should be a simulator that can simulate an accepting transcript without knowing a witness. We have to give some additional powers to the simulator, because we do not want to let anyone create fake proofs without having access to a witness. We allow the simulator to create *crs* himself and some additional information τ called the trapdoor.

Fiat-Shamir Heuristics can be used to convert a public-coin Σ protocol into a ZK-NARK. The verifier no longer generates the challenge, instead the prover uses a cryptographic hash function H to generate a hash of the protocol transcript up to that point and uses this hash as a challenge. During verification, the verifier will perform the same procedure to obtain the challenge. This assures that the prover only obtains the challenge after creating the initial message. If we assume that the cryptographic hash function H is truly random, in which case we call H a 'random oracle', then the ZK-NARK is sound and the the special honest verifier zero-knowledge property of the original Σ protocol transforms into full zero-knowledge [25].

4.1.4. Bilinear Groups

A Bilinear Group generator $\mathcal{BG}(1^\lambda)$ takes a security parameter as input and outputs $bp = (q, \mathbb{G}, \mathbb{G}_T, e, g)$ where:

- $\mathbb{G}$ and $\mathbb{G}_T$ are two cyclic groups of order q
- g is a generator of $\mathbb{G}$
- e is a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. It holds that
  - For all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$ and
  - $e(g, g)$ is a generator of $\mathbb{G}_T$.
- $\mathbb{G}$ is a bilinear group if there exists a group $\mathbb{G}_T$ and a bilinear map as above.

4.1.5. Function Extensions

Definition 4.1.5 (Function Extension). A polynomial h: $F^l \to \mathbb{F}$ is an extension of the function $V : \{0,1\}^l \to \mathbb{F}$ if $\forall b \in \{0,1\}^l : h(b) = V(b)$.

Informally, a polynomial that has the same input-output pairs as a funtion and some additional input-output pairs is an extension of that function.

Definition 4.1.6 (Multilinear Extension (MLE)). Let $V : \{0,1\}^l \to \mathbb{F}$ be a function. Then there exists a unique l-variate polynomial $\tilde{V} : \mathbb{F}^l \to \mathbb{F}$ called the multilinear extension of V, with the properties that:

1. $\tilde{V}$ has degree at most 1 in each variable and

2. $\tilde{V}(x) = V(x)$ for all $x \in \{0,1\}^l$.

Informally, an Multilinear Extension (MLE) is an extension of a function where every variable has degree at most 1. Every function has an unique MLE.

Example 6. Given a function $V : \{0,1\}^2 \to \mathbb{F}$ defined as

$$V(0,0) = 2$$
$$V(0,1) = 3$$
$$V(1,0) = 4$$
$$V(1,1) = 0$$

its MLE is

$$\tilde{V}(x_2, x_1) = 2 + x_1 + 2x_2 - 5x_1x_2$$

We can see that:

$$\tilde{V}(0,0) = V(0,0) = 2$$
$$\tilde{V}(0,1) = V(0,1) = 3$$
$$\tilde{V}(1,0) = V(1,0) = 4$$
$$\tilde{V}(1,1) = V(1,1) = 0$$

Definition 4.1.7 (Equality predicate). A μ-variate equality predicate is a function $eq : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}$ where $eq(a,b) = 1$ if $a = b$, otherwise $eq(a,b) = 0$.

If we are given the values a and b, then we can compute $\tilde{eq}(a,b)$ in $O(\mu)$ time as $\prod_{i=1}^{\mu}(1 - a_i - b_i + 2a_ib_i)$, where a_i and b_i are the i-th value of a and b respectively.

An important lemma that we will be using in constructing proofs for correct ML operations is the following:

Lemma 1 ([[26] Lemma 3.2.1)**.**] For any polynomial $h : \mathbb{F}^l \to \mathbb{F}$ extending $V : \{0,1\}^l \to \mathbb{F}$, it holds:

$$\tilde{V}(X) = \sum_{b \in \{0,1\}^l} \tilde{eq}(X,b) \cdot h(b)$$

Example 7. Given the function $V : \{0,1\}^1 \to \mathbb{F}$ defined as:

$$V(0) = 2$$
$$V(1) = 3$$

, an extension of V

$$h(x_1) = 2 + x_1^2$$
$$h(0) = V(0) = 2$$
$$h(1) = V(1) = 3$$

and the MLE of a 1-variate equality predicate

$$\tilde{eq}(x_2, x_1) = 1 - x_1 - x_2 + 2x_1x_2$$

it can be seen that:

$$\tilde{V}(X) = \sum_{b \in \{0,1\}^l} \tilde{eq}(X, b) \cdot h(b)$$
$$2 + X = \tilde{eq}(X, 0) \times h(0) + \tilde{eq}(X, 1) \times h(1)$$
$$2 + X = (1 - X) \cdot 2 + X \cdot 3$$
$$2 + X = 2 + X$$

4.2. Polynomial Encoding

In order to implement PolyCom, we have to represent polynomials with common programming data structures like vectors. We encoded an l-variate polynomial as a 2^l vector. Let's denote with $i \in \{0, ..., 2^l - 1\}$ the index of an element in the vector. The index i can be written as an l-bit binary number. The bits represent the variables of the polynomial in the way that the least significant bit represents the first variable, the second least significant bit represents the second variable and so forth. The vector element at index i is the coefficient that belongs to the polynomial term that contains the variables that are equal to 1 in the binary representation of i.

Example 8. The polynomial $p(x_1, x_2) = 2 + x_1 + 2x_2 - 5x_1x_2$ is encoded as $[2, 1, 2, -5]$ since

index	binary index	variables	coefficient
0	00	/	2
1	01	x_1	1
2	10	x_2	2
3	11	x_2x_1	-5

This encoding scheme allows us to easily perform operations with polynomials. We have created the Polynomial Evaluation Algorithm for evaluating a polynomial encoded in the vector V with an input represented in the vector T of length l where the first element in T is the value of the highest-index variable. The intuition behind the algorithm is that the highest-index variable is contained in the terms whose coefficients are stored in the last half of the vector, therefore we can multiply those elements with the value of t_l. If we split the vector into two parts then for each part the same holds for the variable t_{l-1}. We can repeat this procedure until we cannot split the vector any more.

Algorithm 1 Polynomial Evaluation Algorithm

```
procedure EVAL(V, T)
    n = ⌈log_2 len(V)⌉                     ▷ Number of variables of the polynomial
    if len(T) ≠ n then                     ▷ We need a value for every variable
        return error
    end if
    M = EVALMONOMIALS(V, T)
    return sum of elements of vector M
end procedure

procedure EVALMONOMIALS(V, T)
    if len(V) = 1 then
        return V
    end if
    t = T[0]                               ▷ Variable with the highest index
    remove T[0] from T
    n = ⌈log_2 len(V)⌉                     ▷ Number of variables of the polynomial
    m = 2^n                                ▷ Maximum number of terms of the polynomial
    V[m/2 :] = V[m/2 :] · t                ▷ Multiply the last half of the vector with t
    L = V[0 : m/2]
    R = V[m/2 :]
    outL = EVAL(L,T)
    outR = EVAL(R,T)
    return concat(outL, outR)
end procedure
```

Example 9. In Figure 4.1 we can see an example execution of the polynomial evaluation algorithm for the polynomial $p(x_1, x_2) = 2 + x_1 + 2x_2 - 5x_1x_2$ on $T = (t_2 = 3, t_1 = 2)$. At the end we have to sum the values $2 + 2 + 6 - 30 = -20$.

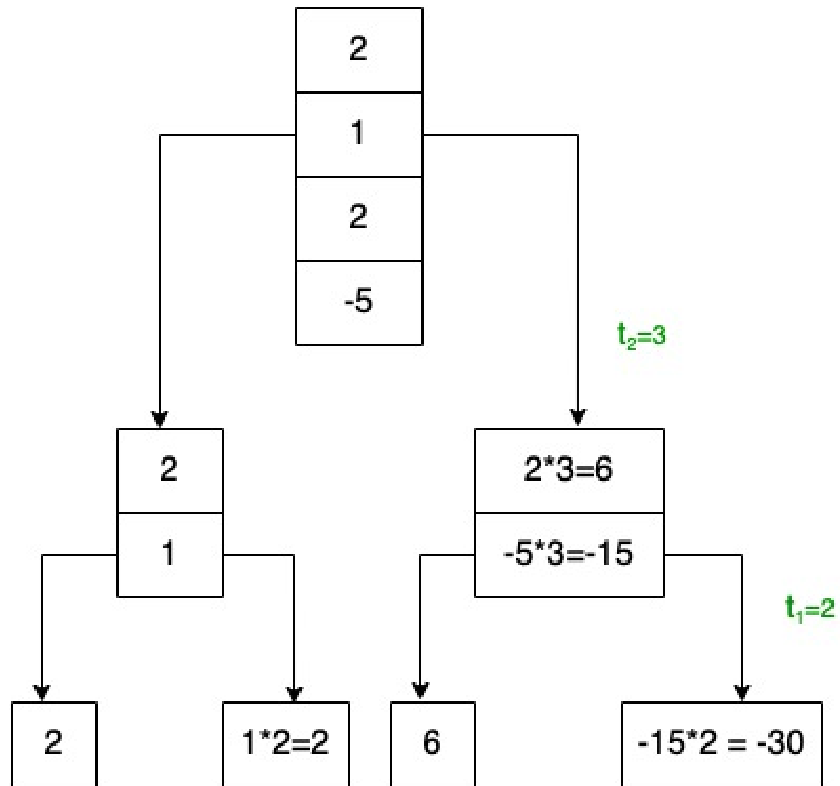


Figure 4.1.: Example execution of the polynomial evaluation algorithm

4.3. PolyCom

PolyCom is a commitment scheme *Com* (see Section 4.1.1), that has the additional properties of:

1. **Linear Homomorphism**: It is required that there exist an algorithm **Eval**$(ck, c_1, ..., c_n, x_1, ..., x_n) \to c_l$ that on an input of n valid commitments where $\text{Commit}(ck, u_i) \to (c_i, o_i)$ and n coefficients $x_1, ...x_n \in Z_q$ outputs a new commitment c_l such that $\text{VerCommit}(ck, c_l, \sum_1^n x_i u_i, o_l) \to 1$, where o_l is calculated as a function of the openings $(o_1, ..., o_n)$ and coefficients $(x_1, ..., x_n)$.
2. **Extractability**: The commitment scheme should be extractable which means that it should not be possible to output a valid commitment without knowing a corresponding pre-image.
3. There should be a **Zero-knowledge argument for commitment-pre-image equality** $ZK_{eq}(u, o_1, o_2, c_1, c_2) \to b \in \{0, 1\}$ for proving that two commitments produced with *Com* have the same pre-image u.
4. There should be a **Zero-knowledge argument for product of pre-images** $ZK_{prod}(u_1, u_2, , o_1, o_2, o_3, c_1, c_2, c_3) \to b \in \{0, 1\}$ for proving that the pre-image of c_3 is the product of pre-images of c_1 and c_2.
5. It should be possible to **commit polynomials**.

In the following subsections we will show how we can implement PolyCom by adapting the Pedersen commitment scheme (see Section 4.1.2).

4.3.1. Linear Homomorphism

The algorithm **Eval**$(ck, c_1, ..., c_n, x_1, ..., x_n) \to c_l$ can be implemented by calculating c_l as:

$$c_l = \sum_1^n c_i^{x_i} = (g^{u_i} h^{o_i})^{x_i} = g^{\sum_{i=1}^n u_i x_i} h^{\sum_{i=1}^n o_i x_i}$$

Futhermore we will create another algorithm **EvalOpening**$(ck, o_1, ..., o_n, x_1, ..., x_n) \to o_l$ with which the opening for c_l can be computed as:

$$o_l = \sum_1^n o_i x_i$$

We can see that $VerCommit(ck, c_l, \sum_1^n x_i u_i, o_l) \to 1$.

4.3.2. Extractability

The Pedersen commitment scheme is not extractable since an adversary can create a valid Pedersen commitment without knowing the pre-image of the commitment by simply generating a random element $c \in_R G$ and sending c as a commitment.

We will modify the Pedersen Commitment Scheme in the following way:

1. In the setup algorithm, we will run a bilinear group generator $\mathcal{BG}(1^\lambda) \rightarrow bp = (q, \mathbb{G}, \mathbb{G}_T, e, g)$ and add bp to the commitment parameters ck. Moreover, a secret value $\beta \in_R Z_q$ will be generated and the values g^β and h^β will be added to ck. It is important that β stays secret.

2. In the commitment algorithm, in addition to generating the value $c^{(1)} = g^u h^o$, the value $c^{(2)} = g^{\beta u} h^{\beta o} = (g^u h^o)^\beta$ will be generated and the output will be $(c = (c^{(1)}, c^{(2)}), o)$.

3. A new algorithm **Check**$(ck, c) \rightarrow b \in \{0, 1\}$ will be introduced with which anyone could check if the committer knows the pre-image of the commitment. The algorithm returns 1 if $e(c^{(1)}, g^\beta) = e(c^{(2)}, g)$ else returns 0. It should be noted, that if an adversary created a commitment by randomly choosing $c^{(1)} \in_R G$, it is not possible to generate $c^{(2)} = c^{(1)\beta}$ since the value β cannot be extracted from the commitment key because the discrete log problem is hard in $\mathbb{G}$.

4.3.3. $\mathrm{ZK_{eq}}$

In Figure 4.2 we presented a non-interactive zero-knowledge argument for Pedersen commitment pre-image equality. The protocol is based on the sigma protocol presented in [25]. We have used the Fiat-Shamir heuristic to convert the interactive sigma protocol into a ZK-NARK.

The original Σ protocol satisfies the notions of completeness, special soundness and special honest verifier zero knowledge as can be seen in Appendix B. If H is a 'random oracle', ZK_{eq} is sound and the the special honest verifier zero-knowledge property of the original sigma protocol transforms into full zero-knowledge [25].

We have created a variation of $\mathrm{ZK_{eq}}$ called $\mathrm{ZK_{eq}}'$. $\mathrm{ZK_{eq}}'$ is a protocol which can be used to prove that two commitments with different bases g_1 and g_2 have the same pre-image. The only difference is that we are using these bases instead of g. The protocol can be seen in Figure 4.3. This protocol is useful because $g_2 = g_1^d$, for some $d \in Z_q$, which means that if the pre-image of c_1 is g_1^u, then the pre-image of c_2 is $g_2^u = g_1^{du}$. Moreover, if we set $g_2 = c_1$, then $c_2 = c_1^u = g^{u^2}$. We can use this protocol to show that the pre-image of c_2 is a square of the pre-image of c_1.

$x = (c_1 = g^u h^{o_1}, c_2 = g^u h^{o_2})$		
Prover		**Verifier**
$w = (o_1, o_2, u)$		
$o_3, o_4, o_5 \in_R Z_q$ $a = g^{o_3} h^{o_4}$ $b = g^{o_3} h^{o_5}$ $c = H(c_1, c_2, a, b)$ $z_1 = c \cdot u + o_3$ $z_2 = c \cdot o_1 + o_4$ $z_3 = c \cdot o_2 + o_5$		
	$\xrightarrow{(a,b,z_1,z_2,z_3)}$	
		$c = H(c_1, c_2, a, b)$ Accept if and only if: $a, b \in \mathbb{G}$ $z_1, z_2, z_3 \in \mathbb{Z}_q$ $a \cdot c_1^c =^? g^{z_1} h^{z_2}$ $b \cdot c_2^c =^? g^{z_1} h^{z_3}$

Figure 4.2.: ZK_{eq}

$x = (g_1, g_2, c_1 = g_1{}^u h^{o_1}, c_2 = g_2{}^u h^{o_2})$		
Prover		**Verifier**
$w = (o_1, o_2, u)$		
$o_3, o_4, o_5 \in_R Z_q$ $a = g_1{}^{o_3} h^{o_4}$ $b = g_2{}^{o_3} h^{o_5}$ $c = H(c_1, c_2, a, b)$ $z_1 = c \cdot u + o_3$ $z_2 = c \cdot o_1 + o_4$ $z_3 = c \cdot o_2 + o_5$		
	$\xrightarrow{(a,b,z_1,z_2,z_3)}$	
		$c = H(c_1, c_2, a, b)$ Accept if and only if: $a, b \in \mathbb{G}$ $z_1, z_2, z_3 \in \mathbb{Z}_q$ $a \cdot c_1^c =^? g_1{}^{z_1} h^{z_2}$ $b \cdot c_2^c =^? g_2{}^{z_1} h^{z_3}$

Figure 4.3.: $ZK_{eq}{}'$

4.3.4. ZK$_{\text{prod}}$

In Figure 4.4 we presented a non-interactive zero-knowledge argument for product of Pedersen commitment-pre-images. The protocol is based on the sigma protocol presented in [25]. We have used the Fiat-Shamir heuristic to convert the interactive sigma protocol into a ZK-NARK.

$x = (c_1 = g^{u_1}h^{o_1}, c_2 = g^{u_2}h^{o_2}, c_3 = g^{u_1 u_2}h^{u_3})$		
Prover		**Verifier**
$w = (u_1, u_2, o_1, o_2, o_3)$		
$d, e, o_4, o_5, o_6 \in_R Z_q$		
$c_d = g^d h^{o_4}$		
$c_e = g^e h^{o_5}$		
$c'_d = c_2^d h^{o_6} = g^{du_2} h^{do_2 + o_6}$		
$c = H(c_1, c_2, c_3, c_d, c_e, c'_d)$		
$f_1 = u_1 \cdot c + d$		
$f_2 = u_2 \cdot c + e$		
$z_1 = o_1 \cdot c + o_4$		
$z_2 = o_2 \cdot c + o_5$		
$z_3 = c \cdot o_3 - c \cdot u_1 \cdot o_2 + o_6$		
	$\xrightarrow{(c_d, c_e, c'_d, f_1, f_2, z_1, z_2, z_3)}$	
		$c = H(c_1, c_2, c_3, c_d, c_e, c'_d)$
		Accept if and only if:
		$c_d, c_e, c_d \in \mathbb{G}$
		$f_1, f_2, z_1, z_2, z_3 \in \mathbb{Z}_q$
		$g^{f_1} h^{z_1} = c_1^c \cdot c_d$
		$g^{f_2} h^{z_2} = c_2^c \cdot c_e$
		$c_2^{f_1} h^{z_3} = c_3^c \cdot c'_d$

Figure 4.4.: ZK_{prod}

The Σ protocol satisfies the notions of completeness, special soundness and special honest verifier zero knowledge as can be seen in Appendix C. If H is a 'random oracle', ZK_{prod} is sound and the the special honest verifier zero-knowledge property of the original sigma protocol transforms into full zero-knowledge [25].

4.3.5. Polynomial Commitment

Now we will show how a user can commit to a polynomial. The main idea is that a TTP will chose values for the variables of the polynomial, the user evaluates the polynomial with those values without learning what those values are and what the result of the evaluation is and commits that evaluation.

There will be an additional input, a positive integer l, in the *Setup* algorithm, which will be used to generate the proving key $\mathbb{P}$. To generate $\mathbb{P}$, the TTP first generates l secret values $x_j \in_R Z_q$ for all $j \in \{1,..,l\}$ and a secret value $\alpha \in_R Z_q$. Mark with W the power set (the set of all subsets) of $\{x_j\}_{j\in\{1,...,l\}}$. W has 2^l elements. Every subset can be uniquely identified with the index $i \in \{0,..,2^l - 1\}$in the same way as described in Section 4.2. Denote with $w_i \in W$ the element of W identified by i. For every $i \in \{0,..,2^l - 1\}$ compute:

$$P_i^{(1)} = g^{\prod_{x\in w_i} x}$$
$$P_i^{(2)} = g^{\alpha \prod_{x\in w_i} x}$$

Add the proving key $\mathbb{P} = \{P_i^{(1)}, P_i^{(2)}\}_{i\in\{1,..,l\}}$ and h^α to the commitment key ck.

To commit a polynomial, generate an opening $o \in_R Z_q$ and exponentiate every proving key P_i with the according coefficient v_i of the polynomial.

$$c^{(1)} = (\prod_{i=0}^{2^l-1} P_i^{(1)v_i})h^o$$
$$c^{(2)} = (\prod_{i=0}^{2^l-1} P_i^{(2)v_i})h^{\alpha o} = c^{(1)\alpha}$$

To verify a polynomial commitment, the user reveals the polynomial and the opening o, and the verifier can see if they have been used to create the commitment c.

Example 10. Now we will show how to commit a polynomial $p(x_2, x_1) = 2 + x_1 + 2x_2 - 5x_1x_2$ if the secret values are $x_1 = 2$ and $x_2 = 3$.

$$\begin{aligned}
c^{(1)} &= (P_0^{(1)})^2 \cdot (P_1^{(1)})^1 \cdot (P_2^{(1)})^2 \cdot (P_3^{(1)})^{-5} \cdot h^o \\
&= (g)^2 \cdot (g^2)^1 \cdot (g^3)^2 \cdot (g^6)^{-5} \cdot h^o \\
&= g^{2+2+6-30}h^o \\
&= g^{-20}h^o \\
c^{(2)} &= (P_0^{(2)})^2 \cdot (P_1^{(2)})^1 \cdot (P_2^{(2)})^2 \cdot (P_3^{(2)})^{-5} \cdot h^o \\
&= (g^\alpha)^2(g^{\alpha 2})^1(g^{\alpha 3})^2(g^{\alpha 6})^{-5}h^{\alpha o} \\
&= g^{\alpha 2+\alpha 2+\alpha 6-\alpha 30}h^{\alpha o} \\
&= (g^{-20}h^o)^\alpha
\end{aligned}$$

It can be seen that we committed $p(3, 2) = -20$.

4.3.6. Overview of PolyCom

PolyCom is a tuple of algorithms (Setup,Commit,VerCommit,Eval,EvalOpening,Check). The definition of these algorithms can be seen in Figure 4.5. For security proofs of PolyCom we refer the reader to [15] and [23].

<table>
<tr><td>Setup$(1^\lambda, l)$</td><td>Commit$(ck, u, type)$</td></tr>
<tr><td>1. $\mathcal{BG}(1^\lambda) \to bp = (q, \mathbb{G}, \mathbb{G}_T, e, g)$
2. Generate h such that no one knows $dlog_g h$
3. Chose $\alpha, \beta, x_1, ..., x_l \in_R Z_q$
4. Compute $\mathbb{P} = \{P_0^{(1)}, ..., P_{2^l-1}^{(1)}, P_0^{(2)}, ..., P_{2^l-1}^{(2)}\}$
• $P_i^{(1)} = g^{\prod_{x \in w_i} x}$
• $P_i^{(2)} = g^{\alpha \prod_{x \in w_i} x}$
output: $ck = (bp, h, \mathbb{P}, h^\alpha, g^\beta, h^\beta)$</td><td>$o \in_R Z_q$
If $type = v$:
• $c^{(1)} = g^u h^o$
• $c^{(2)} = g^{\alpha u} h^{\alpha o}$
Else if $type = p$:
• $c^{(1)} = (\prod_{i=0}^{2^l-1} P_i^{(1)v_i}) h^o$
• $c^{(2)} = (\prod_{i=0}^{2^l-1} P_i^{(2)v_i}) h^{\alpha o}$
output: $(c = (c^{(1)}, c^{(2)}), o)$</td></tr>
<tr><td>Eval$(ck, c_1, ..., c_n, x_1, ..., x_n)$</td><td>VerCommit$(ck, c, u, o, type)$</td></tr>
<tr><td>$c_l^{(1)} = \sum_1^n c_i^{(1)x_i}$
$c_l^{(2)} = \sum_1^n c_i^{(2)x_i}$
output: $(c_l = (c_l^{(1)}, c_l^{(2)}))$</td><td>If $type = v$:
output 1 if:
$c^{(1)} = g^u h^o$
else output 0
If $type = p$:
output 1 if:
$c^{(1)} = (\prod_{i=0}^{2^l-1} P_i^{(1)v_i}) h^o$
else output 0</td></tr>
<tr><td>EvalOpening$(ck, o_1, ..., o_n, x_1, ..., x_n)$</td><td>Check$(ck, c, type)$</td></tr>
<tr><td>$o_l = \sum_1^n o_i x_i$
output: (o_l)</td><td>If $type = v$:
output 1 if:
$e(c^{(1)}, g^\beta) = e(c^{(2)}, g)$
else output 0
If $type = p$:
output 1 if:
$e(c^{(1)}, g^\alpha) = e(c^{(2)}, g)$
else output 0</td></tr>
</table>

Figure 4.5.: PolyCom

4.4. Converting a Vector into a Polynomial

When performing ML inference, the computation is done using vectors and matrices, however PolyCom only allows for commitment of polynomials. In this section we will explain how we can transform vectors and matrices into polynomials to be able to commit them.

Viewing a vector as a function

If a vector has size n, then we can set $l = \lceil \log_2 n \rceil$. We can think about a vector as a function that on an input of an l-bitstring $\{0,1\}^l$ returns the element on the position $\{0,1\}^l$ if we interpret the input as a binary number. If the element does not exist then the function returns 0.

Example 11. Given a vector $x = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$, of size $n = 3$ the value of l can be computed as $l = \lceil \log_2 3 \rceil = 2$ and we can view the vector as a function $V : \{0,1\}^2 \rightarrow \mathbb{F}$ that is defined as :

$$
\begin{aligned}
V(00) &= 2 \\
V(01) &= 3 \\
V(10) &= 4 \\
V(11) &= 0
\end{aligned}
$$

Viewing a matrix as a function

If a matrix has size $n \times m$, then we can set $l_n = \lceil \log_2 n \rceil$ and $l_m = \lceil \log_2 m \rceil$. We can think about a matrix as a function that on an input of an $(l_n + l_m)$-bitstring $\{0,1\}^{(l_n + l_m)}$ returns the element in the row l_n and the column l_m if we interpret l_n and l_m as binary numbers. If the element does not exist then the function returns 0.

Example 12. Given a matrix $x = \begin{bmatrix} 2 & 5 & 4 \\ 3 & 6 & 7 \end{bmatrix}$, of size 2×3 the value of l_n can be computed as $l_n = \lceil \log_2 2 \rceil = 1$, the value of l_m can be computed as $l_m = \lceil \log_2 3 \rceil = 2$ and we can view the matrix as a function $V : \{0,1\}^3 \rightarrow \mathbb{F}$ that is defined as :

$$
\begin{aligned}
V(000) &= 2 & V(100) &= 3 \\
V(001) &= 5 & V(101) &= 6 \\
V(010) &= 4 & V(110) &= 7 \\
V(011) &= 0 & V(111) &= 0
\end{aligned}
$$

Converting a function into a polynomial

We view vectors and matrices as functions $V : \{0,1\}^\mu \to \mathbb{F}$. We know that every such function has a unique MLE polynomial. Therefore, to commit a vector or matrix, we first calculate its MLE and then commit the MLE.

We have created the MLE Algorithm that given a function encoded in V outputs an MLE.

Algorithm 2 MLE Algorithm

```
procedure MLE(V)
    if len(V) = 1 then
        return V
    end if
    n = ⌈log_2 len(V)⌉                    ▷ Number of variables of the polynomial
    m = 2^n                               ▷ Maximum number of terms of the polynomial
    L = V[0 : m/2]
    R = V[m/2 :]
    outL = MLE(L)
    outR = MLE(R)
    outR = outR − outL
    return concat(outL, outR)
end procedure
```

Example 13. An example execution of the MLE algorithm for generating the MLE of the function V defined in the example 11 is shown in the Figure 4.6

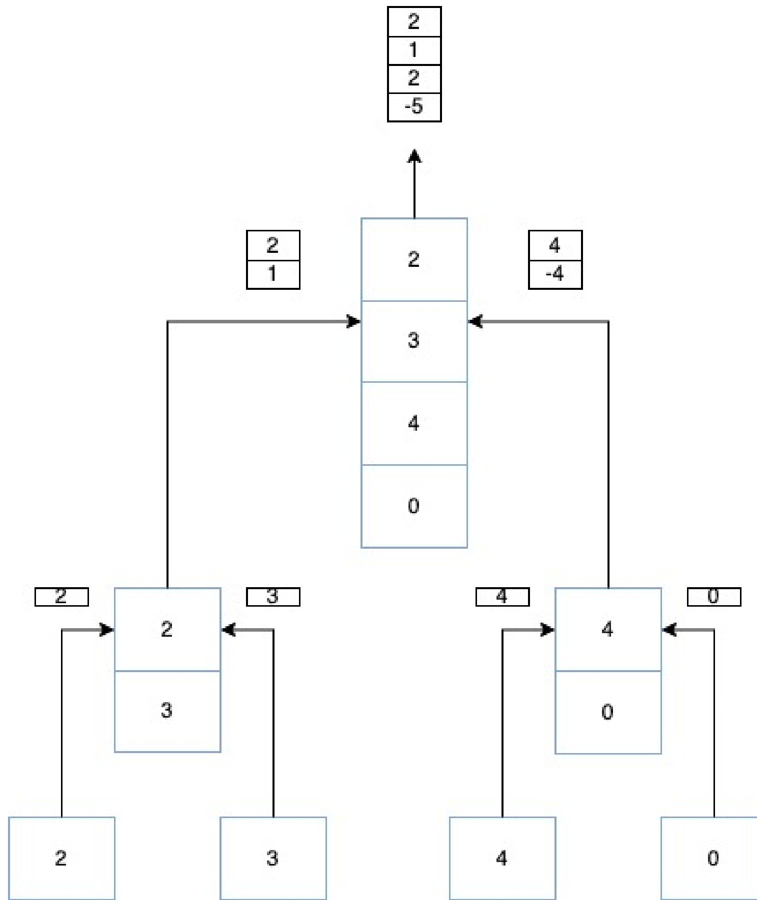


Figure 4.6.: The MLE algorithm

5. Proving correct ML inference computation

In this chapter we will show how a service provider can generate a proof of correct ML inference. To create these proofs we have used the LegoSNARK framework. In Section 5.1 we give an overview of LegoSNARK, in Section 5.2 we show a table which contains all verifiable computation schemes that we have used. In Section 5.3 we present the two fundamental schemes from LegoSNARK that are at the core of all other schemes we have used. In Section 5.4 we present the verifiable computation schemes for operations defined in Section 3.1. In Section 5.5 we show our ML inference verifiable computation schemes. In Section 5.6 we discuss the complexity of our schemes and in Section 5.7 we discuss our implementation and experimental evaluation.

5.1. LegoSNARK

Definition 5.1.1 (ZK-SNARK). A ZK-SNARK is a ZK-NARK(see Definition 4.1.4) that has an additional property that the running time of VerProof is $poly(\lambda)(\lambda + |x| + \log|w|)$ and the proof size is $poly(\lambda)(\lambda + \log|w|)$. This property is called succinctness.

Informally, ZK-SNARKs are special ZK-NARKs that have short and efficiently verifiable proofs. ZK-SNARKs support general computations in the class NP, however, according to Campanelli et al in [15], this generality comes at the expense of performance. They further note that to achieve generality, ZK-SNARKs assume one single unifying representation of the program. However, a program tends to consists of different sub computations that are of different nature. Therefore, general ZK-SNARKs miss the opportunity to optimize these different sub computations. The framework that they have built in [15], called LegoSNARK allows its users to build a ZK-SNARK for a general computation by linking smaller, specialized ZK-SNARKs called gadgets. Gadgets are basic building blocks that can be composed and reused. To make this approach possible, LegoSNARK is based on the Commit-and-Prove (CP) methodology.

Definition 5.1.2 (CP-SNARK). A Commit-and-Prove Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (CP-SNARK) is a ZK-SNARK that allows users to prove statements about committed values. Then the witness domain can be split in two subdomains $\mathcal{D}_w = \mathcal{D}_u \times \mathcal{D}_\omega$ that is $w = (u, \omega)$ where u is the committed portion of the witness and ω is the non-committed portion. Note that not every commitment-based relation has the uncommitted witness portion. A CP-SNARK is a ZK-SNARK that can prove a knowledge of (x, u, ω) such that $R(x, u, \omega) = 1$ and u opens a commitment c_u. Moreover, the committed witness domain can be split in l subdomains called commitment slots, we write $D_u = D_1 \times ... \times D_l$. Then a prover proves knowledge of $(x, u_1, ..., u_l, \omega)$ such that $R(x, u_1, ..., u_l, \omega) = 1$ and $u_1, ..., u_l$

opens the commitments $c_1, ..., c_l$ respectively. More formally, given a family of relations $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$ and a commitment scheme Com whose input space $\mathcal{D}$ is such that $\mathcal{D}_i \subset \mathcal{D}$ for every $i \in \{1, ..., l\}$, a CP-SNARK is a ZK-SNARK for a family of relations $\{\mathcal{R}_\lambda^{Com}\}_{\lambda \in \mathbb{N}}$ such that:

- Every $\mathbf{R} \in \mathcal{R}^{Com}$ is a pair (ck, R) where $ck \leftarrow Setup(1^\lambda)$ and $R \in \mathcal{R}_\lambda$
- $\mathbf{R}$ is over pairs (X, W) where
  - $X := (x, (c_j)_{j \in [l]}) \in \mathcal{D}_x \times \mathcal{C}^l$. That is X consists of a public statement and l commitments.
  - $W := ((u_j)_{j \in [l]}, (o_j)_{j \in [l]}, \omega) \in \mathcal{D}_1 \times ... \times \mathcal{D}_l \times \mathcal{O}^l \times \mathcal{D}_\omega$ That is W consists of inputs and openings of the l commitments and of an uncommitted witness ω.
- R holds if $\wedge_{j \in [l]} VerCommit(ck, c_j, u_j, o_j) \wedge R(x, (u_j)_{j \in [l]}, \omega)$

Definition 5.1.3 (Conjunctions of CP-SNARKs). Let $\{\mathcal{R}_\lambda^{(0)}\}_{\lambda \in \mathbb{N}}$ and $\{\mathcal{R}_\lambda^{(1)}\}_{\lambda \in \mathbb{N}}$ be two families of relations such that for every $\lambda \in \mathbb{N}$ the domain of the relation $R_0 \in \{\mathcal{R}_\lambda^{(0)}\}_{\lambda \in \mathbb{N}}$ can be split into $D_u^{(0)} = D_0 \times D_2$ and the domain of the relation $R_1 \in \{\mathcal{R}_\lambda^{(1)}\}_{\lambda \in \mathbb{N}}$ can be split into $D_u^{(1)} = D_1 \times D_2$ That is the two relations share a commitment slot u_2 called the shared slot. $\{\mathcal{R}_\lambda^{\wedge}\}_{\lambda \in \mathbb{N}}$ is defined as a family of relations where for every $\lambda \in \mathbb{N}$, $\mathcal{R}_\lambda^{\wedge} = \{R_{R_0,R_1}^{\wedge} : R_0 \in \mathcal{R}_\lambda^{(0)}, R_1 \in \mathcal{R}_\lambda^{(1)}\}$ and the relation $R_{R_0,R_1}^{\wedge}(x_0, x_1, u_0, u_1, u_2, (w_0, w_1)) = 1$ if $R_0(x_0, u_0, u_2, w_0) = 1 \wedge R_1(x_1, u_1, u_2, w_1) = 1$. Let Com be a commitment scheme and let CP_b for $b \in \{0, 1\}$ be a CP-SNARK for Com and $\{\mathcal{R}_\lambda^{(b)}\}_{\lambda \in \mathbb{N}}$. We can build a CP-SNARK $CP^{\wedge}$ for Com and $\{\mathcal{R}_\lambda^{\wedge}\}_{\lambda \in \mathbb{N}}$, since it is enough to use CP_b to prove and verify the two statements $(x_b, u_b, u_2, w_b)_{b \in \{0,1\}}$ using the same commitment to u_2. This approach can be applied several times to build a CP-SNARK for composition of several relations. For the proof and and full description we refer the reader to the original paper [15].

5.2. Overview of CP-SNARKs

We have built CP-SNARKs for proving the correct inference of a linear ML model, a linear-kernel SVM model and a polynomial-kernel SVM model by composing CP-SNARKs for matrix multiplication, extended hadamard product, matrix exponentiation, column summation and scalar addition. These CP-SNARKs are built by composing two fundamental CP-SNARK for polynomial evaluation and sumcheck. An overview of all of these CP-SNARKs can be seen in Table 5.1. The CP-SNARKs marked with red color have been constructed by us and the others are part of the LegoSNARK framework [15]. In Section 5.3 we will present the fundamental CP-SNARKs, in Section 5.4 we will present the CP-SNARKs for the ML operations and finally, in Section 5.5 we will present the CP-SNARKs for ML inference.

CP-SNARK	R	X	W
Fundamental CP-SNARKs			
CP-Poly	$R^{poly}(t, f, y) = 1 \iff f(t) = 1$	(t, c_f, c_y)	(f, y, o_f, o_y)
CP-Sumcheck	$R^{sc}(f, y) = 1 \iff \sum_{b_1 \in \{0,1\}} \ldots \sum_{b_l \in \{0,1\}} f(b_1, \ldots, b_l) = y$	(c_f, c_y)	(f, y, o_f, o_y)
CP-SNARKs for ML operations			
CP-MM	$R^{mm}(A, B, C) = 1 \iff mm(A, B) = C$	(c_A, c_B, c_C)	(A, B, C, o_A, o_B, o_C)
CP-EHad	$R^{ehad}(A, B, C) = 1 \iff ehad(A, B) = C$	(c_A, c_B, c_C)	(A, B, C, o_A, o_B, o_C)
CP-Expo	$R^{expo}(d, A, B) = 1 \iff expo(A, d) = B$	(d, c_A, c_B)	(A, B, o_A, o_B)
CP-ColumnSum	$R^{cs}(A, B) = 1 \iff cs(A) = B$	(c_A, c_B)	(A, B, o_A, o_B)
CP-ScalarAdd	$R^{sa}(A, B, C) = 1 \iff sa(A, C) = B$	(c_A, c_B, c_C)	(A, B, C, o_A, o_B, o_C)
CP-SNARKs for ML inference			
CP-Linear	$R^{linear}(W, B, Z, O) = 1 \iff f_{linear}(W, B, Z) = O$	c_W, c_B, c_Z, c_O	(W, B, Z, O, o_W, o_B, o_Z, o_O)
CP-SVMLinear	$R^{SVMLinear}(X, Y, B, Z, O) = 1 \iff f_{SVMLinear}(X, Y, B, Z) = O$	(c_X, c_Y, c_B, c_Z, c_O)	(X, Y, B, Z, O, o_X,o_Y, o_B, o_Z, o_O)
CP-SVMPoly	$R^{SVMPoly}(d, X, Y, B, C, Z, O) = 1 \iff f_{SVMPoly}(d, X, Y, B, C, Z) = O$	(d,c_C, c_X, c_Y, c_B, c_Z, c_O)	(C, X, Y, B, Z, O, o_C, o_X,o_Y, o_B, o_Z, o_O)

Table 5.1.: Overview of CP-SNARKs

5.3. Fundamental CP-SNARKs

5.3.1. CP-Poly

CP-Poly is a fundamental CP-SNARK presented in the LegoSNARK framework and is a building block of many other CP-SNARKs. Suppose that a prover has an l-variate polynomial $f(x_1, ..., x_l)$ and evaluates that polynomial on some public input $(t_1, ..., t_l)$ and obtains the value $y = f(t_1, ..., t_l)$. He then commits the polynomial PolyCom.Commit$(ck, f, p) \rightarrow (c_f, o_f)$ and the evaluation
PolyCom.Commit$(ck, y, v) \rightarrow (c_y, o_y)$ using the previously described commitment scheme PolyCom. CP-Poly allows a prover to convince a verifier, given only the polynomial commitment c_f, the evaluation commitment c_y and the public input $(t_1, ..., t_l)$ that the pre-image of c_y is the evaluation of the polynomial which is a pre-image of c_f on the public input $(t_1, ..., t_l)$. The protocol relies on the polynomial decomposition theorem.

Lemma 2 (Polynomial Decomposition). Let $f : \mathbb{F}^l \rightarrow \mathbb{F}$ be a polynomial. For all $t \in \mathbb{F}^l$ there exist efficiently computable polynomials $q_1, ..., q_l$ such that: $f(x) - f(t) = \sum_{i=1}^{l}(x_i - t_i)q_i(x)$ where t_i is the i-th element of t.

The prover computes and commits the polynomials q_i using the proving key $\mathbb{P}$ from the commitment scheme and sends the commitments to the verifier as proof. The proving algorithm can be seen in Figure 5.1 and the verification algorithm in Figure 5.2.

Public statement: (t, c_f, c_y)

Prover
Witness: (f, y, o_f, o_y)
 	
1. Compute $q_i(x), \forall i \in \{1, ..., l\}$ such that:
  • $f(x) - f(t) = \sum_{i=1}^{l}(x_i - t_i)q_i(x)$
 	
2. PolyCom.Commit$(ck, q_i, p) = (c_{q_i}, o_{q_i}), \forall i \in \{1, ..., l\}$
 	
3. Compute $c_{q_o} = (q_0^{(0)}, q_0^{(1)})$
  • compute $q_0^{(1)} = \frac{g^{o_f - o_y}}{\prod_{i=1}^{l}(P_{2i-1}^{(1)} g^{-t_i})^{o_{q_i}}} = g^{o_f - o_y - \sum_{i=1}^{l}(x_i - t_i)o_{q_i}}$
  • compute $q_0^{(2)} = \frac{(g^{\alpha})^{o_f - o_y}}{\prod_{i=1}^{l}(P_{2i-1}^{(2)} g^{-\alpha t_i})^{o_{q_i}}} = g^{\alpha(o_f - o_y - \sum_{i=1}^{l}(x_i - t_i)o_{q_i})}$
 	
output: $\pi = (c_{q_0}, ..., c_{q_l})$

Figure 5.1.: Prover algorithm for CP-Poly

Public statement: (t, c_f, c_y)
Verifier Input: $\pi = (q_0, c_{q_1}, ..., c_{q_l})$ 1. Compute $A = e(g, \frac{c_f}{c_y})$ • Note that $A = e(g,g)^{f(x)-y+x(o_f-o_y)}$ 2. Compute $B = \prod_{i=1} e(P^{(1)}_{2^{i-1}} g^{-t_i}, c_{q_i})$ • Note that $B = e(g,g)^{\sum_{i=1}^{l}(x_i-t_i)q_i(x)+x(\sum_{i=1}^{l}(x_i-t_i)o_{q_i})}$ 3. Compute $C = e(q_0, h)$ • Note that $C = e(g,g)^{x(o_f-o_y)-x(\sum_{i=1}^{l}(x_i-t_i)o_{q_i})}$ Output 1 if and only if: • $A = B \cdot C$ • PolyCom.Check$(ck, c_f, p) = 1$ • PolyCom.Check$(ck, c_y, v) = 1$ • $\forall i \in \{0, ..., l\}$, PolyCom.Check$(ck, c_{q_i}, p) = 1$, otherwise output 0.

Figure 5.2.: Verifier algorithm for CP-Poly

Implementation

We have created the Polynomial Decomposition Algorithm to decompose the polynomial. The main idea is that we first divide the l-variate polynomial with $(x_l - t_l)$. The quotient of the division is the polynomial q_l and then we divide the $l-1$-variate remainder with $(x_{l-1} - t_{l-1})$ to obtain q_{l-1}. We continue until we obtain the polynomial q_1.

Example 14. An example execution of the Decomposition algorithm for generating the polynomials q_i of the function $f(x_1, x_2) = 2 + x_1 + 2x_2 - 5x_1x_2$ for the public input $(t_1 = 2, t_2 = 3)$ is shown in Figure 5.3. It can be seen that the verification will hold.

$$f(x_1,x_2) - f(t_1,t_2) = \sum_{i=1}^{l}(x_i - t_i)q_i(x)$$
$$22 + x_1 + 2x_2 - 5x_1x_2 = -14(x_1 - 2) + (x_2 - 3)(2 - 5x_1)$$
$$22 + x_1 + 2x_2 - 5x_1x_2 = -14x_1 + 28 + 2x_2 - 5x_1x_2 - 6 + 15x_1$$
$$22 + x_1 + 2x_2 - 5x_1x_2 = 22 + x_1 + 2x_2 - 5x_1x_2$$

Algorithm 3 Polynomial Decomposition Algorithm

```
procedure DecomposePoly(V, T, y)
    out = []                                   ▷ The polynomials q_i
    V[0] = V[0] − y                            ▷ subtract the evaluation from the polynomial
    while len(V) > 1 do
        t = T[0]                               ▷ Variable with the highest index
        remove T[0] from T
        (Q, R) = Divide(V,t)
        V = R
        append Q to out
    end while
    return out
end procedure

procedure Divide(V, t)
    n = ⌈log_2 len(V)⌉                         ▷ Number of variables of the polynomial
    m = 2^n                                    ▷ Maximum number of terms of the polynomial
    L = V[0 : m/2]                             ▷ Terms that do not contain the last variable
    R = V[m/2 :]                               ▷ Terms that contain the last variable
    Q = R                                      ▷ Quotient
    R = L + R · t                              ▷ Remainder
    return (Q, R)
end procedure
```

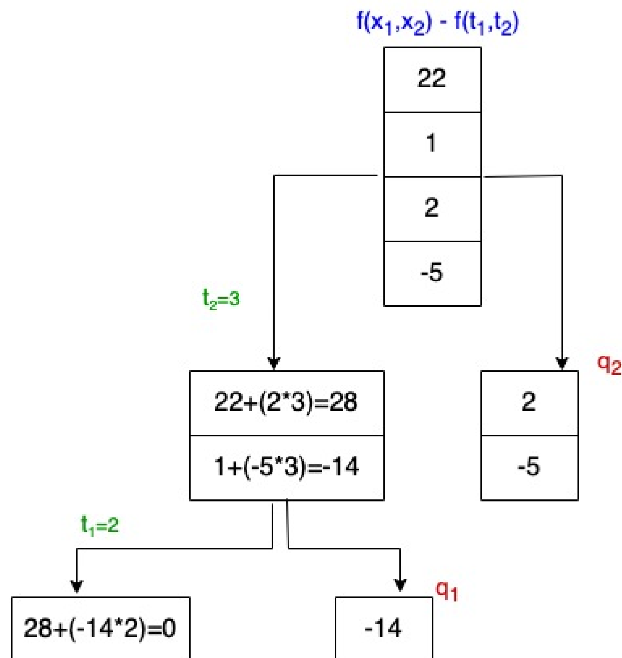


Figure 5.3.: Example Execution of the Polynomial Decomposition Algorithm

5.3.2. CP-Sumcheck

CP-Sumcheck is another fundamental CP-SNARK presented in the LegoSNARK framework and is a building block of many other CP-SNARKs. We will first define what a sumcheck is:

Definition 5.3.1 (Sumcheck). Given an l-variate polynomial $f(x_1, ..., x_l)$, the value $y = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_l \in \{0,1\}} f(b_1, ..., b_l)$ is called the sumcheck.

Suppose that a prover has a l-variate polynomial $f(x_1, ..., x_l)$ and obtains the sumcheck value y. Note that if the polynomial $f(x_1, ..., x_l)$ is an MLE of a vector, then y is the sum of elements of the vector. He then commits the polynomial PolyCom.Commit$(ck, f, p) \rightarrow (c_f, o_f)$ and the sumcheck PolyCom.Commit$(ck, y, v) \rightarrow (c_y, o_y)$ using the previously described commitment scheme PolyCom. CP-Sumcheck allows a prover to convince a verifier, given only the polynomial commitment c_f and the sumcheck commitment c_y that the pre-image of c_y is the sumcheck of the polynomial which is a pre-image of c_f. The prover algorithm can be seen in Figure 5.4 and the verifier algorithm can be seen in Figure 5.5.

When using CP-Sumcheck, there can be a need to prove some additional properties of the function $f(x_1, ..., x_l)$. Therefore we have created CP-Sumcheck'. The proving algorithm is the same as in CP-Sumcheck, the only difference is that in CP-Sumcheck', there is no computation of π_{poly} and the proof consists only of π_{sc}. Additionally, the prover has to memorize the values of (c_l, o_l, r) to use them in the proving of additional properties of $f(x_1, .., x_l)$. The verification algorithm is the same as in CP-Sumcheck, we only do not perform the last step since we do not have π_{poly}. The verifier also has to memorise (c_l, r) to use in further verifications. The public statement of CP-Sumcheck' consists only of c_y.

Public statement: (c_f, c_y)

Prover

Witness: (f, y, o_f, o_y)

Set:

$c_0 = c_y$

$o_0 = o_y$

$r = \{\}$

For every round $i \in \{1, .., l\}$:

1. $g_i(x) = \sum_{b_{i+1},...,b_l \in \{0,1\}} f(r_1, ..., r_{l-1}, x, b_{i+1}, ..., b_l)$
   - Note that $g_i(0) + g_i(1) =$ input of c_{i-1}
   - Define m as the number of coefficients of $g_i(x)$
   - $\forall j \in \{0, ..., m-1\}$ define a_j as the j-th coefficient of $g_i(x)$
2. $\forall j \in \{0, ..., m-1\}, (c_{a_j}, o_{a_j}) \leftarrow Com(ck, a_j)$
   - Note that $g_i(0) + g_i(1) =$ input of $c_{a_0} \prod_{j=0}^{m-1} c_{a_j}$
   - Define $\mathbb{K}_i = \{c_{a_j}\}_{j \in \{0,...,m-1\}}$
3. $\text{ZK}_{\text{eq}}.\text{Prove}(u, o_1, o_2, c_1, c_2) \rightarrow \pi_i$ where:
   - $c_1 = c_{i-1}$
   - $o_1 = o_{i-1}$
   - $c_2 = c_{a_0} \prod_{j=0}^{m-1} c_{a_j}$
   - $o_2 = \sum_{j=0}^{m-1} o_{a_j}$
   - $u = g_i(0) + g_i(1)$
4. $r_i = H(\mathbb{K}_i)$
   - Append r_i to r
5. $c_i = Eval(ck, c_{a_0}, ..., c_{a_m}, 1, r_i, ..., r_i^m)$
   - Note that $g_i(r_i) =$ input of c_i
6. $o_i = \sum_{0}^{m-1} o_{a_j} r_i^j$

$\pi_{poly} = \text{CP-Poly.Prove}(x = (r, c_f, c_l), w = (f, o_f, f(r), o_l))$
- $f(r) =$ input of c_l

$\pi_{sc} = (\mathbb{K}_1, ..., \mathbb{K}_l, \pi_1, ..., \pi_l)$

output $\pi = (\pi_{poly}, \pi_{sc})$

Figure 5.4.: Prover algorithm for CP-Sumcheck

Public statement: (c_f, c_y)

Verifier
input $\pi = (\pi_{poly}, \pi_{sc})$

Set:
$c_0 = c_y$
$r = \{\}$

For every round $i \in \{1, .., l\}$:

1. $\mathrm{ZK_{eq}.Verify}(c_{i-1}, c_{a_0} \prod_{j=0}^{m-1} c_{a_j}, \pi_i)$
   - $c_{a_j} \in \mathbb{K}_i$
2. $r_i = H(\mathbb{K}_i)$
   - Append r_i to r
3. $c_i = Eval(ck, c_{a_0}, ..., c_{a_m}, 1, r_i, ..., r_i^m)$

CP-Poly.Verify$(\pi_{poly}, x = (r, c_f, c_l))$

output 1 if all verifications succeed, else 0

Figure 5.5.: Verifier algorithm for CP-Sumcheck

5.4. CP-SNARKs for ML Operations

5.4.1. CP-MM

Given three commitments c_A, c_B and c_C, CP-MM is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_C is a matrix multiplication of the pre-images of c_A and c_B.

More formally, a CP-MM is a CP-SNARK for relation $\mathbf{R} = (ck, R^{mm})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (c_A, c_B, c_C)$, $\mathbf{W} = (A, B, C, o_A, o_B, o_C)$ and $R^{mm}(A, B, C) = 1 \iff mm(A, B) = C$. The protocol is presented in [15] and we will provide its explanation. The definition of matrix multiplication is given in Definition 3.1.1.

The intuition behind the proof is the following observation: if $\tilde{A}$ is an MLE of an $m \times n$ matrix A, $\tilde{B}$ an MLE of a $n \times k$ matrix B, $\tilde{C}$ an MLE of $mm(A, B)$ and $\mu = \lceil \log_2 n \rceil$ then the following holds $\forall R \in \mathbb{F}^{\lceil \log_2 m \rceil} \wedge \forall C \in \mathbb{F}^{\lceil \log_2 k \rceil}$:

$$\tilde{C}(R, C) = \sum_{b \in \{0,1\}^{\mu}} \tilde{A}(R, b) \times \tilde{B}(b, C)$$

We will show this observation in an example.

Example 15. Given

$$A \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix}, B \begin{bmatrix} 3 \\ 3 \end{bmatrix} \text{ and } mm(A,B) = C \begin{bmatrix} 18 \\ 33 \end{bmatrix}$$

and their corresponding MLEs:

$$\tilde{A}(x_2, x_1) = 2 + 2x_1 + 3x_2 - x_1x_2$$
$$\tilde{B}(x_2, x_1) = 3 - 3x_1$$
$$\tilde{C}(x_2, x_1) = 18 - 18x_1 + 15x_2 - 15x_1x_2$$

We can see that the following equation holds:

$$\tilde{C}(x_2, x_1) = \sum_{b \in \{0,1\}^\mu} \tilde{A}(x_2, b) \times \tilde{B}(b, x_1)$$
$$\tilde{C}(x_2, x_1) = \tilde{A}(x_2, 0) \times \tilde{B}(0, x_1) + \tilde{A}(x_2, 1) \times \tilde{B}(1, x_1)$$
$$\tilde{C}(x_2, x_1) = (2 + 3x_2) \times (3 - 3x_1) + (4 + 2x_2) \times (3 - 3x_1)$$
$$\tilde{C}(x_2, x_1) = 18 - 18x_1 + 15x_2 - 15x_1x_2$$

To prove the correct matrix multiplication we can generate a public input by using a random oracle $H(c_A, c_B, c_C) \rightarrow I, J$. We can evaluate the MLE of C on that input $\tilde{C}(I, J) \rightarrow c$ and partially evaluate the MLEs of A and B: $\tilde{A}(I, X) \rightarrow \tilde{A}'(X)$ and $\tilde{B}(X, J) \rightarrow B'(X)$. Then we get the following equation:

$$\tilde{C}(I, J) = \sum_{b \in \{0,1\}^\mu} \tilde{A}(I, b) \times \tilde{B}(b, J)$$
$$c = \sum_{b \in \{0,1\}^\mu} \tilde{A}'(b) \times \tilde{B}'(b)$$

We can see that c is a sumcheck (see Definition 5.3.1) of the function $f(x) = \tilde{A}'(x) \times \tilde{B}'(x)$. We can use CP-Sumcheck' since we want to prove that c is the sumcheck of $f(x)$ and the additional property of $f(x)$ that it is a multiplication of $\tilde{A}'(x)$ and $\tilde{B}'(x)$.

For the CP-Sumcheck' protocol we need a commitment of the sumcheck c. We first need to create the commitment of that value c_c and use CP-Poly to prove that the value committed in c_c is an evaluation of the polynomial committed in c_C on the public input (I, J).

In the CP-Sumcheck protocol the commitment c_l, from the round l, has the same pre-image as $f(r)$ where r is the random input generated in the protocol. In CP-MM, $f(r) = \tilde{A}'(r) \times \tilde{B}'(r)$, therefore the pre-image of c_l is equal to $\tilde{A}'(r) \times \tilde{B}'(r)$.

We can evaluate $\tilde{A}'(r) = \tilde{A}(I, r) \rightarrow a$ and $\tilde{B}'(r) = \tilde{B}(r, J) \rightarrow b$, create commitments c_a and c_b which are commitments of a and b respectively, and use CP-Poly to prove that the pre-images of c_a, c_b are evaluations of polynomials which are pre-images of c_A, c_B on public inputs (I, r) and (r, J) respectively.

Finally, we can use ZK_{prod} to prove that the pre-image of c_l is a product of pre-images of c_a and c_b. The complete proving and verification algorithm can be seen in Appendix D.

5.4.2. CP-EHad

Given three commitments c_A, c_B, c_C, CP-EHad is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_C is the extended Hadamard product of the pre-images of c_A and c_B.

More formally, a CP-EHad is CP-SNARK for relation $\mathbf{R} = (ck, R^{ehad})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (c_A, c_B, c_C)$, $\mathbf{W} = (A, B, C, r_A, r_B, r_C)$ and $R^{ehad}(A, B, C) = 1 \iff ehad(A, B) = C$. We created this protocol by modifying CP_{had} from [15], which is a CP-SNARK for Hadamard Products. The definition of the extended hadamard product is given in Definition 3.1.2.

The intuition behind the proof is that if $\tilde{A}$ is an MLE of an $m \times n$ matrix A, $\tilde{B}$ an MLE of a vector B of length m, $C = ehad(A, B)$, $\mu = \lceil \log_2 m \rceil$, $\nu = \lceil \log_2 n \rceil$, $R \in \mathbb{F}^{\mu}$ and $C \in \mathbb{F}^{\nu}$ then the function $f(R, C) = \tilde{A}(R, C) \times \tilde{B}(R)$ is an extension of C.

Example 16. Given:

$$A = \begin{bmatrix} 3 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 2 & 5 \end{bmatrix}, C = \begin{bmatrix} 6 & 4 \\ 15 & 20 \end{bmatrix}$$

We can notice that:

$$\tilde{A}(x_2, x_1) = 3 - x_1 + 2x_1x_2$$
$$\tilde{B}(x_2) = 2 + 3x_2$$
$$f(x_2, x_1) = \tilde{A}(x_2, x_1) \times \tilde{B}(x_2)$$

$$f(0,0) = 3 \times 2 = 6$$
$$f(0,1) = 2 \times 2 = 4$$
$$f(1,0) = 3 \times 5 = 15$$
$$f(1,1) = 4 \times 5 = 20$$

Then according to Lemma 1, the following holds:

$$\begin{aligned} \tilde{C}(R, C) &= \sum_{b_r \in \{0,1\}^{\mu}} \sum_{b_c \in \{0,1\}^{\nu}} eq(R, C, b_r, b_c) \times f(b_r, b_c) \\ &= \sum_{b_r \in \{0,1\}^{\mu}} \sum_{b_c \in \{0,1\}^{\nu}} eq(R, C, b_r, b_c) \times \tilde{A}(b_r, b_c) \times \tilde{B}(b_r) \end{aligned}$$

To prove the correct extended hadamard product computation we can generate a public input by using a random oracle $H(c_A, c_B, c_C) \to I, J$. We can evaluate the MLE of C on that input $\tilde{C}(I, J) \to c$ and we can partially evaluate the $\mu + \nu$- variate equality predicate

$eq(I, J, X_r, X_c) \to \tilde{eq}'(X_r, X_c)$. Then we get the following equation:

$$\tilde{C}(I, J) = \sum_{b_r \in \{0,1\}^\mu} \sum_{b_c \in \{0,1\}^\nu} eq(I, J, b_r, b_c) \times \tilde{A}(b_r, b_c) \times \tilde{B}(b_r)$$
$$c = \sum_{b_r \in \{0,1\}^\mu} \sum_{b_c \in \{0,1\}^\nu} \tilde{eq}'(b_r, b_c) \times \tilde{A}(b_r, b_c) \times \tilde{B}(b_r)$$

We can see that c is a sumcheck (see Definition 5.3.1) of the function $f(x_r, x_c) = \tilde{eq}'(x_r, x_c) \times \tilde{A}(x_r, x_c) \times \tilde{B}(x_r)$. We can use CP-Sumcheck' since we want to prove that c is the sumcheck of $f(x_r, x_c)$ and the additional property of $f(x_r, x_c)$ that it is a multiplication of $\tilde{eq}'(x_r, x_c)$, $\tilde{A}(x_r, x_c)$ and $\tilde{B}(x_r)$.

For the CP-Sumcheck' protocol we need a commitment of the sumcheck c. We first need to create the commitment of that value c_c and use CP-Poly to prove that the value committed in c_c is an evaluation of the polynomial committed in c_C on the public input (I, J).

In the CP-Sumcheck protocol the commitment c_l, from the round l, has the same pre-image as $f(r_r, r_c)$ where (r_r, r_c) is the random input generated in the protocol. In CP-Ehad, $f(r_r, r_c) = \tilde{eq}'(r_r, r_c) \times \tilde{A}(r_r, r_c) \times \tilde{B}(r_r)$, therefore the pre-image of c_l is equal to $\tilde{eq}'(r_r, r_c) \times \tilde{A}(r_r, r_c) \times \tilde{B}(r_r)$.

We can evaluate $\tilde{A}(r_r, r_c) \to a$ and $\tilde{B}(r_r, r_c) \to b$, create commitments c_a and c_b which are commitments of a and b respectively, and use CP-Poly to prove that the pre-images of c_a, c_b are evaluations of polynomials which are pre-images of c_A, c_B on the public input (r_r, r_c). Furthermore we will evaluate $\tilde{eq}'(r_r, r_c) = \tilde{eq}(I, J, r_r, r_c) \to e$.

Finally, we can use ZK_{prod} to prove that the pre-image of c_l is a product of pre-images of c_a, c_b and the value e. The complete proving and verification algorithm can be seen in Appendix E.

5.4.3. CP-ColumnSum

Given two commitments c_A, c_B, CP-ColumnSum is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_B is a column summation of the matrix A which is a pre-image of c_A.

More formally, a CP-ColumSum is CP-SNARK for relation $\mathbf{R} = (ck, R^{cs})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (c_A, c_B)$, $\mathbf{W} = (A, B, o_A, o_B)$ and $R^{cs}(A, B) = 1 \iff cs(A) = B$. The definition of column summation is given in Definition 3.1.4

The intuition behind this proof is similar as in CP-MM. If $\tilde{A}(R, C)$ is the MLE of a $m \times n$ matrix A, $\tilde{B}(C)$ is the MLE of $cs(A)$ and $\mu = \lceil \log_2 m \rceil$, it can be seen that $\forall C \in \mathbb{F}^{\lceil \log_2 n \rceil}$:

$$\tilde{B}(C) = \sum_{b \in \{0,1\}^\mu} \tilde{A}(b, C)$$

Example 17. Given

$$A \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \text{ and } B \begin{bmatrix} 7 & 10 \end{bmatrix}$$

and their corresponding MLEs:

$$\tilde{A}(x_2, x_1) = 2 + 2x_1 + 3x_2 - x_1x_2$$
$$\tilde{B}(x_1) = 7 + 3x_1$$

We can see that the following equation holds:

$$\tilde{B}(x_1) = \sum_{b \in \{0,1\}^{\mu}} \tilde{A}(b, x_1)$$
$$\tilde{B}(x_1) = (2 + 2x_1) + (5 + 1x_1)$$
$$\tilde{B}(x_1) = 7 + 3x_1$$

To prove the correct column summation we can generate a public input by using a random oracle $H(c_A, c_B) \to J$. We can evaluate the MLE of B on that input $\tilde{B}(J) \to b$ and partially evaluate the MLE of A $\tilde{A}(X, J) \to \tilde{A}'(X)$ Then we get the following equation:

$$\tilde{B}(C) = \sum_{b \in \{0,1\}^{\mu}} \tilde{A}(b, C)$$
$$b = \sum_{b \in \{0,1\}^{\mu}} \tilde{A}'(b)$$

We can see that b is a sumcheck (see Definition 5.3.1) of the function $f(x) = \tilde{A}'(x)$. We can use the CP-Sumcheck protocol to prove this.

For the CP-Sumcheck protocol we need a commitment of the sumcheck b. We first need to create the commitment of that value c_b and use CP-Poly to prove that the value committed in c_b is an evaluation of the polynomial committed in c_B on the public input J. The full proving and verification algorithm can be seen Appendix H.

5.4.4. CP-Expo

Given two commitments c_A, c_B and an integer value d, CP-Expo is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_B is the matrix exponentiation of the matrix committed in c_A and the exponent d.

More formally, a CP-Expo is CP-SNARK for relation $\mathbf{R} = (ck, R^{expo})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (d, c_A, c_B)$, $\mathbf{W} = (A, B, o_A, o_B)$ and $R^{expo}(A, B, d) = 1 \iff expo(A, d) = B$. The definition of matrix exponentiation is given in Definition 3.1.3

The basic idea behind the proof is similar to CP-EHad. If $\tilde{A}$ is an MLE of a $m \times n$ matrix A, d a positive integer, $B = expo(A, d)$, $\mu = \lceil \log_2 m \rceil$, $\nu = \lceil \log_2 n \rceil$, $R \in \mathbb{F}^{\mu}$ and $C \in \mathbb{F}^{\nu}$ then the function $f(R, C) = \tilde{A}(R, C)^d$ is an extension of B.

Example 18. Given:

$$B = \begin{bmatrix} 9 & 4 \\ 9 & 16 \end{bmatrix}, A = \begin{bmatrix} 3 & 2 \\ 3 & 4 \end{bmatrix}, d = 2$$

We can notice that:

$$\tilde{A}(x_2, x_1) = 3 - x_1 + 2x_1x_2$$
$$f(x_2, x_1) = \tilde{A}(x_2, x_1)^2$$
$$f(0, 0) = 3^2 = 9$$
$$f(0, 1) = 2^2 = 4$$
$$f(1, 0) = 3^2 = 9$$
$$f(1, 1) = 4^2 = 16$$

Then according to Lemma 1, the following holds:

$$\tilde{B}(R, C) = \sum_{b_r \in \{0,1\}^{\mu}} \sum_{b_c \in \{0,1\}^{\nu}} \tilde{eq}(R, C, b_r, b_c) \times f(b_r, b_c)$$
$$\tilde{B}(R, C) = \sum_{b_r \in \{0,1\}^{\mu}} \sum_{b_c \in \{0,1\}^{\nu}} \tilde{eq}(R, C, b_r, b_c) \times (\tilde{A}(b_r, b_c))^d$$

To prove the correct matrix exponentation computation we can generate a public input by using a random oracle $H(c_A, c_B, c_C) \to I, J$. We can evaluate the MLE of B on that input $\tilde{B}(I, J) \to b$ and we can partially evaluate the $\mu + \nu$- variate equality predicate $eq(I, J, X_r, X_c) \to \tilde{eq}'(X_r, X_c)$. Then we get the following equation:

$$\tilde{B}(I, J) = \sum_{b_r \in \{0,1\}^{\mu}} \sum_{b_c \in \{0,1\}^{\nu}} eq(I, J, b_r, b_c) \times \tilde{A}(b_r, b_c)^d$$
$$b = \sum_{b_r \in \{0,1\}^{\mu}} \sum_{b_c \in \{0,1\}^{\nu}} \tilde{eq}'(b_r, b_c) \times \tilde{A}(b_r, b_c)^d$$

We can see that b is a sumcheck (see Definition 5.3.1) of the function $f(x_r, x_c) = \tilde{eq}'(x_r, x_c) \times \tilde{A}(x_r, x_c)^d$. We can use CP-Sumcheck' since we want to prove that b is the sumcheck of $f(x_r, x_c)$ and the additional property of $f(x_r, x_c)$ that it is a multiplication of $\tilde{eq}'(x_r, x_c)$ and the polynomial $\tilde{A}(x_r, x_c)$ raised to the power of d.

For the CP-Sumcheck' protocol we need a commitment of the sumcheck b. We first need to create the commitment of that value c_b and use CP-Poly to prove that the value committed in c_b is an evaluation of the polynomial committed in c_B on the public input (I, J).

In the CP-Sumcheck protocol the commitment c_l, from the round l, has the same pre-image as $f(r_r, r_c)$ where (r_r, r_c) is the random input generated in the protocol. In CP-Expo, $f(r_r, r_c) = \tilde{eq}'(r_r, r_c) \times \tilde{A}(r_r, r_c)^d$, therefore the pre-image of c_l is equal to $\tilde{eq}'(r_r, r_c) \times \tilde{A}(r_r, r_c)^d$.

We can evaluate $\tilde{A}(r_r, r_c) \to a$ create the commitment c_a which is a commitment of a and use CP-Poly to prove that the pre-image of c_a is an evaluation of the polynomial which is a pre-images of c_A, on the public input (r_r, r_c).

We will also create commitments of $(a^2, ..., a^d)$. For every $i \in \{2, .., d\}$, we can use ZK_{eq}' to prove that c_a and c_{a^i} contain the same pre-image a when the base of c_a is g and the base of c_{a^i} is $c_{a^{(i-1)}}$. This will allow us the following: Since we have proven that c_a contains a, we can prove that the pre-image of c_{a^2} is also a if the base is $c_{a^{2-1}} = c_a = g^a$ which makes the real pre-image of $c_{a^2} = (g^a)^a = g^{a^2}$. Then we prove that the pre-image of c_{a^3} is also equal to a if the base is c_{a^2}. We have already proven that the pre-image of $c_{a^2} = g^{a^2}$ which makes the real pre-image of $c_{a^3} = (g^{a^2})^a = g^{a^3}$.

Furthermore we will evaluate $\tilde{eq}'(r_r, r_c) = \tilde{eq}(I, J, r_r, r_c) \to e$.

Finally, we can use ZK_{eq} to prove that the pre-image of c_l is equal to the pre-image of c_{a^d} multiplied by the value e. The complete proving and verification algorithm can be seen in Appendix F.

5.4.5. CP-ScalarAdd

Given three commitments c_A, c_B, c_C, CP-ScalarAdd is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that every element of matrix B, which is the pre-image of c_B, is equal to the sum of the same element of matrix A, which is the pre-image of c_A, and the scalar C, which is the pre-image of com_C.

More formally, a CP-ScalarAdd is CP-SNARK for relation $\mathbf{R} = (ck, R^{sa})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (c_A, c_B, c_c)$, $\mathbf{W} = (A, B, C, o_A, o_B, o_C)$ and $R^{sa}(A, B, C) = 1 \iff sa(A, C) = B$. The definition of scalar addition is given in Definition 3.1.5

The basic idea of the proof is that if $\tilde{A}$ is an MLE of an $m \times n$ matrix A, C a scalar value and $\tilde{B}$ an MLE of $sc(A, C)$, then the following holds $\forall X \in \mathbb{F}^{\lceil \log_2 m \rceil + \lceil \log_2 m \rceil}$:

$$\tilde{B}(X) = \tilde{A}(X) + C$$

Example 19. Given

$$A = \begin{bmatrix} 3 & 4 & 2 & 6 \end{bmatrix}, B = \begin{bmatrix} 6 & 7 & 5 & 9 \end{bmatrix} \text{ and } C = 3,$$

it can be seen that:

$$\tilde{B}(X) = \tilde{A}(X) + C$$
$$6 + x_1 - x_2 + 3x_1x_2 = 3 + x_1 - x_2 + 3x_1x_2 + 3$$

The proof is based on the following observation:

$$\frac{c_B}{c_A} = g^{B(X)-A(X)}h^{o_B-o_A} = g^C h^{o_B-o_A}$$

The prover can use ZK_{eq} to prove that the commitment $\frac{c_B}{c_A}$ contains the same pre-image C as c_C, if the opening of $\frac{c_B}{c_A}$ is $o_B - o_A$. The full proving and verification algorithms can be seen in Appendix H.

CP-ScalarAdd has a limitation that we will address in future work. The proof only works if A is a matrix where the number of rows and columns are a power of 2.

Example 20. Given

$$A = \begin{bmatrix} 3 & 4 & 2 \end{bmatrix}, B = \begin{bmatrix} 6 & 7 & 5 \end{bmatrix} \text{ and } C = 3,$$

it can be seen that:

$$\tilde{A}(x_1, x_2) = 3 + x_1 - x_2 - 3x_1x_2$$
$$\tilde{B}(x_1, x_2) = 6 + x_1 - x_2 - 6x_1x_2$$

$$\tilde{A}(x_1, x_2) + C \neq \tilde{B}(x_1, x_2)$$

This is because the matrices actually look like:

$$A = \begin{bmatrix} 3 & 4 & 2 & 0 \end{bmatrix}, B = \begin{bmatrix} 6 & 7 & 5 & 0 \end{bmatrix}$$

As it can be seen, the fourth element of B is not equal to the fourth element of A + C

5.5. CP-SNARKs for ML Inference

5.5.1. CP-Linear

Using the property of CP-SNARK composition (see 5.1.3) we have created CP-Linear. Given 4 commitments c_W, c_B, c_Z and c_O, CP-Linear is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_O is a vector that contains all predictions of the dataset committed in c_Z evaluated with a linear model with parameters W and B committed in c_W and c_B respectively.

More formally, CP-Linear is a CP-SNARK for relation $\mathbf{R} = (ck, R^{linear})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (c_W, c_B, c_Z, c_O)$, $\mathbf{W} = (W, B, Z, O, o_W, o_B, o_Z, o_O)$ and $R^{Linear}(W, B, Z, O) = 1 \iff f_{linear}(model = (W, B), Z) = O$. The function f_{linear} is formally defined in Definition 3.2.1.

The proving and verification algorithms can be seen in Appendix I.

5.5.2. CP-SVMLinear

Using the property of CP-SNARK composition (see 5.1.3) we have created CP-SVMLinear. Given 5 commitments c_X, c_Y, c_B, c_Z and c_O, CP-SVMLinear is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_O is a vector that contains all predictions of the dataset committed in c_Z evaluated with a linear-kernel SVM model with parameters X, Y and B committed in c_X, c_Y, c_B respectively.

More formally, CP-SVMLinear is a CP-SNARK for relation $\mathbf{R} = (ck, R^{svmLinear})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (c_X, c_Y, c_B, c_Z, c_O)$, $\mathbf{W} = (X, Y, B, Z, O, o_X, o_Y, o_B, o_Z, o_O)$ and $R^{svmLinear}(X, Y, B, Z, O) = 1 \iff f_{SVMLinear}(model = (X, Y, B), Z) = O$. The function $f_{SVMLinear}$ is formally defined in Definition 3.3.1.

The proving and verification algorithms can be seen in Appendix J.

5.5.3. CP-SVMPoly

Using the property of CP-SNARK composition (see 5.1.3) we have created CP-SVMPoly. Given 6 commitments $c_X, c_C, c_Y, c_B, c_Z, c_O$ and a positive integer value d, CP-SVMPoly is a CP-SNARK that a prover can use to prove that he knows how to open the given commitments and that the pre-image of c_O is a vector that contains all predictions of the dataset committed in c_Z evaluated with a polynomial-kernel SVM model with parameters X, C, Y, B committed in c_X, c_C, c_Y, c_B respectively and the kernel degree d.

More formally, a CP-SVMPoly is a CP-SNARK for relation $\mathbf{R} = (ck, R^{svmPoly})$ on pairs $(\mathbf{X}, \mathbf{W})$, where $\mathbf{X} = (d, c_X, C_C, c_Y, c_B, c_Z, c_O)$, $\mathbf{W} = (X, C, Y, B, Z, O, o_X, o_Y, o_B, o_Z, o_O)$ and $R^{svmPoly}(d, X, C, Y, B, Z, O) = 1 \iff f_{SVMPoly}(model = (d, X, C, Y, B), Z) = O$. The function $f_{SVMPoly}$ is formally defined in Definition 3.3.2.

The proving and verification algorithms can be seen in Appendix K.

5.6. Complexity Analysis

The time and space complexity of our protocols are dependent on the complexities of the building blocks: CP-Poly, CP-Sumcheck, ZK_{eq} and ZK_{prod}. These complexities are shown in Table 5.2. The protocols ZK_{eq} and ZK_{prod} have constant space and time complexity.

The complexity of CP-Poly is dependent on the number of variables of the polynomial whose correct evaluation we are proving. We denote this quantity with l. The proving time is exponential in l and the verification is linear in l. To prove the correct computation we decompose the polynomial into l polynomials. Each of these polynomials gets committed and we also have to generate one additional commitment to make the verification possible. A commitment consists of two elements of the group $\mathbb{G}$, therefore the CP-Poly proof consists of $2 \cdot (l+1)$ elements of $\mathbb{G}$.

The complexity of CP-Sumcheck is dependent on the number of variables of the polynomial whose correct sumcheck we are proving. We denote this quantity with l. It is also dependent on the highest variable degree of that polynomial. We denote this quantity with m. The proving time is exponential in l and the verification is linear in l. The sumcheck protocol proceeds in l rounds. In every round we generate a univariate round polynomial

	Proving time	Verification time	Proof Size	Space
CP-Poly	$O(2^l)$	$O(l)$	$2 \cdot (l+1)\ \mathbb{G}$	$O(l)$
CP-Sumcheck	$O(2^l)$	$O(l)$	l ZK$_{EQ}$ + $l \cdot (m+1)\ \mathbb{G}$	$O(lm)$
ZK$_{eq}$	$O(1)$	$O(1)$	2 $\mathbb{G}$ and 3 Z_q	$O(1)$
ZK$_{prod}$	$O(1)$	$O(1)$	3 $\mathbb{G}$ and 5 Z_q	$O(1)$

Table 5.2.: Complexity: CP-Poly, CP-Sumcheck, ZK$_{eq}$, ZK$_{prod}$, l is the number of variables in the polynomial

	CP-Poly	l	CP-Sumcheck	l	m	ZK$_{eq}$	ZK$_{prod}$	Com
(1)	$\tilde{C} : \mathbb{F}^{r+c} \to \mathbb{F}$ $\tilde{A} : \mathbb{F}^{r+b} \to \mathbb{F}$ $\tilde{B} : \mathbb{F}^{b+c} \to \mathbb{F}$	r+c r+b b+c	$\tilde{A}'(b) \times \tilde{B}'(b)$	b	2		1	3
(2)	$\tilde{C} : \mathbb{F}^{r+c} \to \mathbb{F}$ $\tilde{A} : \mathbb{F}^{r+c} \to \mathbb{F}$ $\tilde{B} : \mathbb{F}^{r} \to \mathbb{F}$	r+c r+c r	$\tilde{eq}'(r,c) \times$ $\tilde{A}(r,c) \times$ $\tilde{B}(r)$	r+c	3		1	3
(3)	$\tilde{B} : \mathbb{F}^{r+c} \to \mathbb{F}$ $\tilde{A} : \mathbb{F}^{r+c} \to \mathbb{F}$	r+c r+c	$\tilde{eq}'(r,c) \times$ $\tilde{A}(r,c)^d$	r+c	d+1	d		d+1
(4)	$\tilde{B} : \mathbb{F}^{c} \to \mathbb{F}$ $\tilde{A} : \mathbb{F}^{r+c} \to \mathbb{F}$	c r+c	$\tilde{A}'(r)$	r	1	1		2
(5)						1		

Table 5.3.: Complexity: (1)CP-MM, (2)CP-EHad, (3)CP-Expo, (4)CP-ColumnSum, (5)CP-ScalarAdd

and we commit every coefficient of that polynomial. The number of coefficients depends on the degree of the round polynomial m and is equal to $m+1$. Note that the coefficient commitments only consist of one element of $\mathbb{G}$. Additionally, in every round we generate one ZK_{eq} proof. That makes the CP-Sumcheck proof consist of l ZK_{eq} proofs and $l \cdot (m+1)$ elements of $\mathbb{G}$.

To determine the complexity of CP-MM, CP-EHad, CP-Expo, CP-ColumnSum and CP-ScalarAdd, we have created the Table 5.3. For each of these proofs, it can be seen what are the building blocks and what are the complexity parameters of those building blocks. For CP-Poly we show the polynomial that gets evaluated and the number of variables l of that polynomial. For CP-Sumcheck we show the polynomial, the number of variables l and the highest variable degree m. We show how many ZK_{eq} and ZK_{prod} proofs are contained in each CP-SNARK and how many additional commitments are part of the proof.

5.7. Implementation and Experimental Evaluation

We have implemented all of the aforementioned protocols in the Go programming language. The source code is publicly available on `https://github.com/marianasamardzic/MLPSP`.

Next we will present the results of the experimental evaluation of our implementation of CP-SVMPoly, CP-SVMLinear and CP-Linear. The experiments are run on a 2,7 GHz Quad-Core Intel Core i7 processor with 16GB of RAM memory.

The runtime of CP-Linear.Prove and CP-Linear.Verify depends on the number of features of the linear ML model and on the size of the client's dataset. We have ran two experiments. In the first we fixed the number of features to 2^2 and varied the dataset size. In the second one we fixed the dataset size to 2^2 and varied the number of features. The results can be seen in Table 5.4.

The runtime of CP-SVMLinear.Prove and CP-SVMLinear.Verify depends on the number of features of the SVM model, the number of support vectors and on the size of the client's dataset. We have ran three experiments. In the first we fixed the number of features and support vectors to 2^2 and varied the dataset size. In the second one we fixed the dataset size and the number of support vectors to 2^2 and varied the number of features. In the third experiment we fixed the number of features and the dataset size to 2^2 and varied the number of support vectors. The results can be seen in Table 5.5.

The runtime of CP-SVMPoly.Prove and CP-SVMPoly.Verify depends on the number of features of the SVM model, the number of support vectors, the degree of the kernel and on the size of the client's dataset. We have ran four experiments. In the first we fixed the number of features and support vectors to 2^2 and the degree to 2 and varied the dataset size. In the second one we fixed the dataset size and the number of support vectors to 2^2 and the degree to 2 and varied the number of features. In the third experiment we fixed the number of features and the dataset size to 2^2, the degree to 2 and varied the number of support vectors. In the fourth experiment we fixed the number of features, support vectors and the dataset size to 2^2 and varied the kernel degree. The results can be seen in Table 5.6.

An interesting observation is that the number of features has a quite smaller influence on the runtime then the number of support vectors or the dataset size in SVM models. Another observation is that the degree of the polynomial kernel also does not influence the runtime very much.

In our implementation we provide code for benchmarking so all interested users can test the runtime with their desired configurations.

number of data instances	proving runtime (s)	verification runtime (s)
2^1	0.068486743	0.22165685
2^2	0.09140313	0.250679938
2^3	0.148570304	0.282334357
2^4	0.25366345	0.314890437
2^5	0.442880971	0.341474642
2^6	0.860304684	0.378023662
2^7	1.781879743	0.405651067
2^8	3.96959471	0.438057254
2^9	9.77601659	0.454717729
2^{10}	27.087514313	0.477222708

number of features	proving runtime (s)	verification runtime (s)
2^1	0.054702478	0.240234194
2^2	0.098670574	0.251311838
2^3	0.151078259	0.26153616
2^4	0.261934845	0.275170184
2^5	0.457610367	0.285683859
2^6	0.889116608	0.295073782
2^7	1.857223815	0.306302478
2^8	4.234431977	0.321828342
2^9	10.672226001	0.32767693
2^{10}	31.052624167	0.337051496

Table 5.4.: CP-Linear Runtime analysis

number of data instances	proving runtime (s)	verification runtime (s)
2^1	0.128853601	0.22165685
2^2	0.182592318	0.250679938
2^3	0.266597005	0.282334357
2^4	0.452716289	0.314890437
2^5	0.807331104	0.341474642
2^6	1.553573792	0.378023662
2^7	3.103460944	0.405651067
2^8	6.592880845	0.438057254
2^9	15.004700498	0.454717729
2^{10}	37.827697086	0.477222708
number of features	**proving runtime (s)**	**verification runtime (s)**
2^1	0.158764115	0.240234194
2^2	0.177687881	0.251311838
2^3	0.211877876	0.26153616
2^4	0.266923349	0.275170184
2^5	0.38760932	0.285683859
2^6	0.592957529	0.295073782
2^7	1.021796911	0.306302478
2^8	1.942966487	0.321828342
2^9	3.980013094	0.32767693
2^{10}	8.597983797	0.337051496
number of support vectors	**proving runtime (s)**	**verification runtime (s)**
2^1	0.115141937	0.214385787
2^2	0.166888711	0.242103184
2^3	0.263351118	0.274611575
2^4	0.448066782	0.302725977
2^5	0.791859524	0.340152848
2^6	1.509397471	0.369433461
2^7	3.029344191	0.395631042
2^8	6.541912918	0.425473345
2^9	15.271164463	0.456970409
2^{10}	40.479785438	0.495786622

Table 5.5.: CP-SVMLinear Runtime analysis

number of data instances	proving runtime (s)	verification runtime (s)
2^1	0.180641079	0.2868852
2^2	0.270887896	0.329650168
2^3	0.425649604	0.372847961
2^4	0.714760426	0.414024308
2^5	1.276818524	0.455996926
2^6	2.480512374	0.497677561
2^7	4.924208207	0.539451493
2^8	10.55959713	0.582156291
2^9	24.67293386	0.622069595
2^{10}	64.751735881	0.633355032
number of features	proving runtime (s)	verification runtime (s)
2^1	0.253541382	0.318867734
2^2	0.274903163	0.330071496
2^3	0.30331878	0.34193528
2^4	0.363543703	0.352419487
2^5	0.464600598	0.363660277
2^6	0.693669642	0.37446447
2^7	1.117033945	0.384365613
2^8	2.043482119	0.400424928
2^9	4.036891361	0.408353101
2^{10}	8.690006365	0.417617568
number of support vectors	proving runtime (s)	verification runtime (s)
2^1	0.175146175	0.270865071
2^2	0.266549149	0.316345114
2^3	0.412172509	0.358874139
2^4	0.685573339	0.403700077
2^5	1.233781813	0.44383008
2^6	2.351765819	0.492827398
2^7	4.762056232	0.523400062
2^8	10.388264175	0.569435159
2^9	24.546662798	0.61389956
2^{10}	65.893386607	0.64414548
degree	proving runtime (s)	verification runtime (s)
2	0.252276876	0.317186382
3	0.255662178	0.316767943
4	0.275670177	0.318416232
5	0.27714237	0.32323411
6	0.276638925	0.32589876
7	0.291484782	0.323438989
8	0.29988677	0.325664135
9	0.30289234	0.329766084
10	0.310534408	0.332323192

Table 5.6.: CP-SVMPoly Runtime analysis

6. ML Prediction Service Platforms

In this section we will present our two ML prediction service platforms: MLPSP and IP-MLPSP. The first one satisfies all requirements discussed in the introduction except of input privacy and the second one satisfies both the model and input privacy, but the outcome verifiability is probabilistic. Both platforms are based on the blockchain technology and IP-MLPSP uses homomorphic encryption to satisfy input privacy. First, we will give an overview of these two technologies.

6.1. Prerequisites

6.1.1. Blockchain

Blockchain is a distributed ledger where the transactions are stored in a chain of blocks. The committed transactions cannot be changed or deleted, and users can only view the transactions and create new ones. Key properties of blockchain are decentralization, persistency, anonymity and auditability and the core technologies that blockchain relies on are cryptographic hash functions, digital signatures and distributed consensus mechanism [27].

Blockchain technology can be used to implement smart contracts. A smart contract is a computer program that implements a contract and that will automatically be executed when conditions defined in the contract are met. Additionally, the smart contract cannot be modified once the smart contract has been installed on the blockchain [27].

6.1.2. Homomorphic Encryption

Homomorphic encryption is an encryption scheme that allows for certain computable functions to be executed on ciphertexts. For example, given an $Enc(m_1)$ and $Enc(m_2)$, a user can obtain $Enc(m_1 + m_2)$ without knowing the values of m_1 and m_2. Homomorphic encryption schemes can be classified in three groups depending on the number of operations that can be performed on ciphertexts. Partially Homomorphic Encryption allows one operation unlimited number of times. Somewhat homomorphic encryption allows some type of operations only a limited number of times. Fully Homomorphic Encryption allows an unlimited number of operations for an unlimited number of times [14].

6.2. MLPSP

In this section we will present the design for MLPSP that fulfills all of the security requirements discussed in Chapter 1 except of the input privacy. Users that have developed their own machine learning models (which can be linear models, SVM models with linear kernels or SVM models with polynomial kernels) can offer prediction services on that models using MLPSP. We call these users service providers. Users that want to use these prediction services can browse the available models, choose the ones that they would like to use and send their datasets to the model owners and get the predictions back in exchange for a financial compensation. We call these users clients. Both clients and service providers might act maliciously. The service provider might want to return random or false predictions back and the client might want to steal the ML models. There is one additional user called the platform owner. The platform owner's only task is to initialize the system and we assume that the platform owner is honest but curious, in other words, the platform owner might want to steal the secret ML models but will perform all operations required by the protocol honestly.

The platform consists of a blockchain where four data types can be stored. Since we are using blockchain all stored data is immutable: it cannot be deleted or changed. We also assume that for all uploaded data, it can be seen which user uploaded it. An overview of these data types can be seen in Figure 6.1. The first data type is a commitment key. It can only be uploaded by the platform owner during the platform initialization and there can only be one such key. The second data type is a model. Models are uploaded by service providers and every model has an unique model ID. A single service provider can upload multiple models. The third data type is a request. Requests are uploaded by clients and every request has an unique request ID. A single client can upload multiple requests. The final data type is the proof. Every proof contains an unique request ID, meaning that every request can only have one proof. The proofs with a certain request ID can only be uploaded by the service provider who has registered the model that has the same model ID as the model ID specified in the request with the proof's request ID.

One of the requirements for MLPSP is that we have fair payments. If a service provider outputs a valid proof, the client should pay the fee. Because of this requirement the client cannot do the verification. The client has the incentive to tell that the proof is not correct, even though it is, in order to not pay and get the predictions. Therefore, we need a third party that both the service provider and the verifier trust to do the verification. An ideal solution is a smart contract because its execution and business logic cannot be modified after being deployed on the blockchain. When the service provider uploads the proof of correct computation on the blockchain, the verification smart contract gets invoked. The smart contract verifies the proof using the data stored on the blockchain and if the proof is correct, the smart contract transfers the money from the client to the service provider.

To show the design of MLPSP, we will create a generic protocol CP-ML which takes as input the model type (linear, SVMLinear, SVMPoly) and based on the type will perform CP-Linear, CP-SVMLinear or CP-SVMPoly. With $\{param\}$ we will denote the set of all parameters of a model, with $\{c_{param}\}$ the set of all commitments of those parameters and with $\{o_{param}\}$ the set of all openings of those commitments. We have also created a generic function $f_{ML}(type, \{param\}, z)$ that based on the type performs either f_{linear}, $f_{SVMLinear}$ or $f_{SVMPoly}$. Moreover, in Figure 6.2, we have showed all methods that users can use when interacting with the platform. We assume that all service providers and clients have downloaded the commitment key when they joined the platform using the GetCK() method.

Commitment key ck
Model: • model ID • model type • model parameters commitments $\{c_{param}\}$ • prediction fee
Request: • request ID • model ID • dataset commitment c_Z
Proof: • request ID • proof π • output commitment c_O • output O • output opening o_O

Figure 6.1.: MLPSP Data Types

1. GetCK() / SetCK(ck)
   - returns(puts) the commitment key from(on) the blockchain
2. GetModel(model ID) / SetModel(Model)
   - returns(puts) a model from(on) the blockchain
3. GetRequest(request ID) / SetRequest(Request)
   - returns(puts) a request from(on) the blockchain
4. GetProof(request ID) / SetProof(Proof)
   - returns(puts) a proof from(on) the blockchain
5. Verify(request ID)
   - Invokes the verification smart contract

Figure 6.2.: MLPSP Methods

There are six distinct actions that can be performed on the platform:

1. Platform Setup
2. Model Registration
3. Request Registration
4. Proof Registration
5. Verification
6. Output Retrieval

In the following subsections we will give more information on each of these actions.

Platform Setup

The platform owner initializes MLPSP. The owner chooses the security parameter λ and the value l and creates the commitment key. The value l determines the maximum number of variables that a polynomial can have, which restricts the size of the ML models and client's datasets. The platform owner deletes the secret values used to generate the commitment key and then stores the commitment key on the blockchain. If the platform owner deletes the secret values, then we can be sure that the proofs cannot be forged. Otherwise the platform owner can collude with the service provider and generate fake proofs. The platform setup is performed only once and can be seen in Figure 6.3.

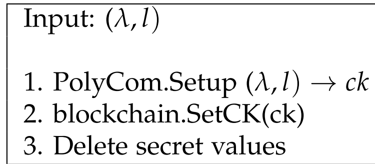


Figure 6.3.: MLPSP Platform Setup

Model Registration

A service provider may own multiple machine learning models. To be able to offer prediction services with these models, they have to be registered on the blockchain. The service provider keeps a list of registered models, denoted by the variable models. Every ML model gets registered only once, and can be used for multiple requests. The process of model registration can be seen in Figure 6.4.

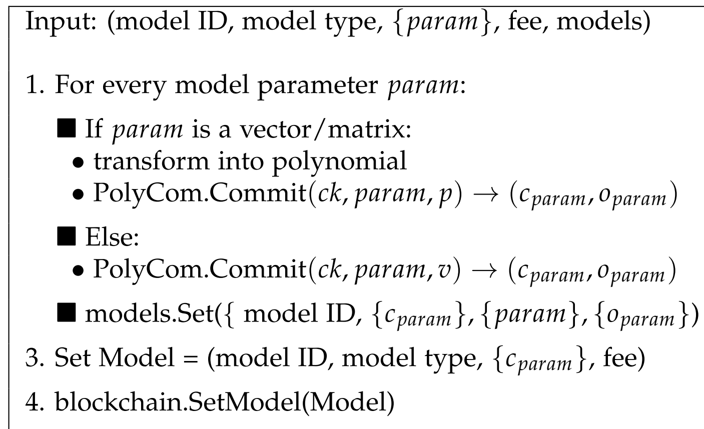
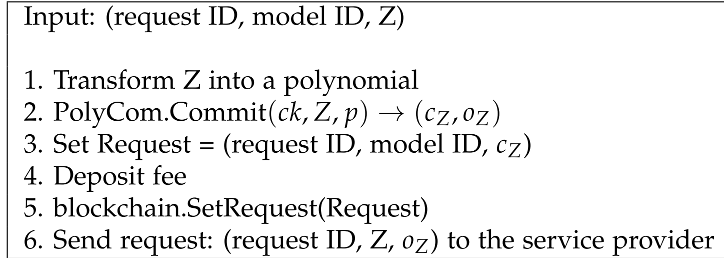


Figure 6.4.: MLPSP Model Registration

Request Registration

To create an ML prediction request, the client has to choose one of the registered ML models. The client has to have a suitable dataset, denoted by Z, that can be evaluated with the chosen ML model and enough money in the blockchain system. The process of creating a request can be seen in Figure 6.5.

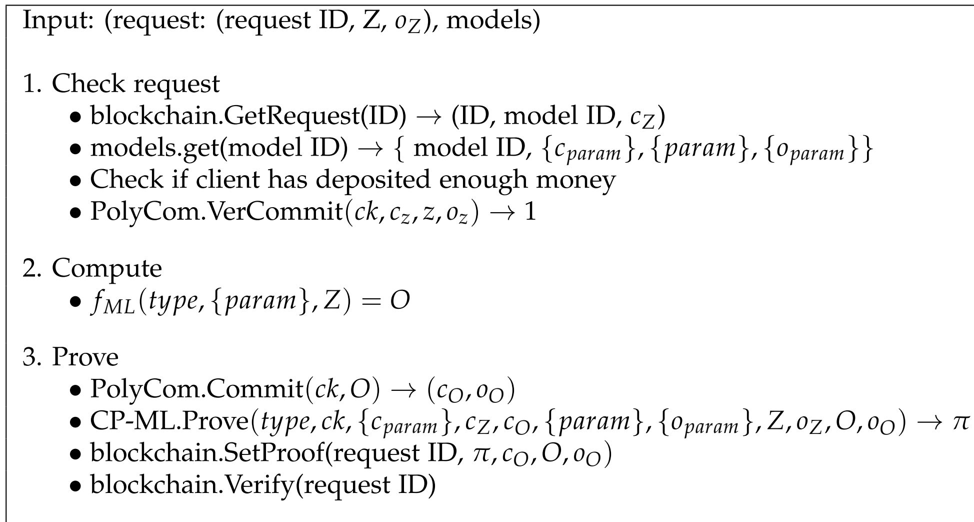

Input: (request ID, model ID, Z)

1. Transform Z into a polynomial
2. PolyCom.Commit$(ck, Z, p) \rightarrow (c_Z, o_Z)$
3. Set Request = (request ID, model ID, c_Z)
4. Deposit fee
5. blockchain.SetRequest(Request)
6. Send request: (request ID, Z, o_Z) to the service provider

Figure 6.5.: MLPSP Request Registration

Proof Registration

When the service provider gets a request from a client, the service provider has to check if the request is correctly registered on the blockchain. This involves checking if the requested model is owned by the service provider, if the client deposited enough money and if the received dataset is the one which is committed on the blockchain. Since PolyCom is binding, the client cannot commit one dataset and send another one to the service provider. If all checks are successful, then the service provider calculates the predictions and creates a proof of correct computation. The proof and the predictions get stored on the blockchain. The complete process can be seen in Figure 6.6.

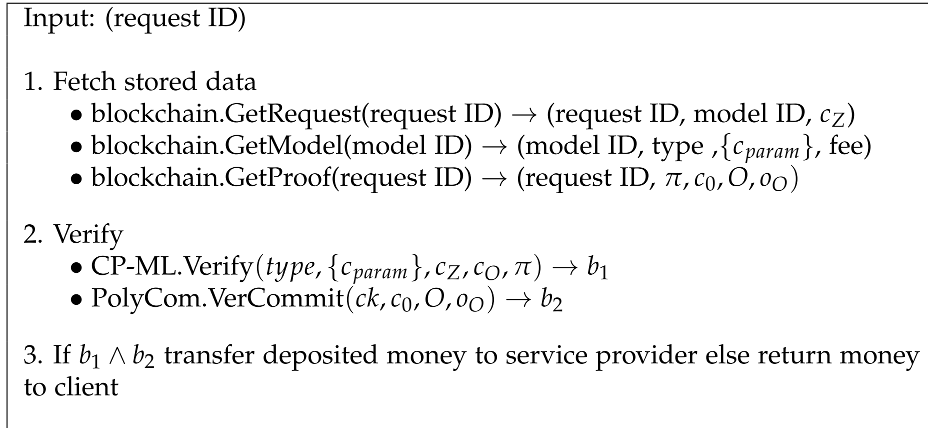

Input: (request: (request ID, Z, o_Z), models)

1. Check request
   - blockchain.GetRequest(ID) $\rightarrow$ (ID, model ID, c_Z)
   - models.get(model ID) $\rightarrow$ { model ID, $\{c_{param}\}, \{param\}, \{o_{param}\}$}
   - Check if client has deposited enough money
   - PolyCom.VerCommit$(ck, c_z, z, o_z) \rightarrow 1$
2. Compute
   - $f_{ML}(type, \{param\}, Z) = O$
3. Prove
   - PolyCom.Commit$(ck, O) \rightarrow (c_O, o_O)$
   - CP-ML.Prove$(type, ck, \{c_{param}\}, c_Z, c_O, \{param\}, \{o_{param}\}, Z, o_Z, O, o_O) \rightarrow \pi$
   - blockchain.SetProof(request ID, π, c_O, O, o_O)
   - blockchain.Verify(request ID)

Figure 6.6.: MLPSP Proof Registration

Verification

The verification is run by the smart contract. The smart contract is invoked by the service provider and the only input is the request ID. All the other data can be fetched from the blockchain by the smart contract using this request ID. The verification process is shown in Figure 6.7 and involves checking two things: the first one is checking if the proof is correct and the second one is checking if the returned predictions are committed in the the output commitment c_O used in the proof. Additionally, we assume there is a certain timeout. This timeout starts when the client uploads a request and if the smart contract does not get invoked in time, the request gets cancelled and the deposited money is returned back to the client.



Input: (request ID)

1. Fetch stored data
   - blockchain.GetRequest(request ID) $\rightarrow$ (request ID, model ID, c_Z)
   - blockchain.GetModel(model ID) $\rightarrow$ (model ID, type ,$\{c_{param}\}$, fee)
   - blockchain.GetProof(request ID) $\rightarrow$ (request ID, $\pi, c_0, O, o_O)$

2. Verify
   - CP-ML.Verify$(type, \{c_{param}\}, c_Z, c_O, \pi) \rightarrow b_1$
   - PolyCom.VerCommit$(ck, c_0, O, o_O) \rightarrow b_2$

3. If $b_1 \wedge b_2$ transfer deposited money to service provider else return money to client

Figure 6.7.: MLPSP Verification

Output Retrieval

Once the verification process is complete, the client can simply download the output O from the proof with the request's ID. The client can be sure that the output is correct.

6.2.1. Security Discussion

In Table 6.1 we have discussed how MLPSP satisfies the defined security requirements.

1.	Model Privacy	The model never leaves the service provider. Everyone can see the model commitment but since PolyCom is perfectly hiding no one can see what is committed, not even the platform owner. However, there is a possibility of a model extraction attack (more information in Chapter 7) if a single client performs many requests or if many clients collaborate. However, this attack is out of scope of this work.
2.	Input Privacy	There is no input privacy since the client's dataset is seen by the service provider.
2. a)	Output Privacy	There is no output privacy since the client's output is stored on the blockchain.
3.	Outcome Verifiability	If CP-ML satisfies the soundness property, then MLPSP satisfies outcome verifiability. To remind the reader, the soundness property assures that a correct proof cannot be generated if the user does not know the witness.
3. a)	Model Registration	The service provider has to register its model ahead of time. Since PolyCom is binding, it is not possible to find another model that will have the same commitment.
3. b)	Dataset Registration	The client creates a commitment of the dataset. The service provider cannot find another dataset with the same commitment since PolyCom is binding. The client cannot cheat since the client can also not find another dataset with the same commitment.
3. c)	Matching Output	The output commitment c_O is used in the proof verification. If the proof is correct, then the pre-image of c_O is a vector O that contains the correct predictions. The service provider reveals the pre-image O and the opening o_O and the smart contract can check if these two values have been used in generating c_o. Because of the binding property of PolyCom, the server cannot generate another vector O' and another opening o'_O that will match c_O. Therefore, if this verification succeeds, we can be sure that the returned output is correct.
4.	Batch Verification	It can be seen that there is one proof of correct predictions for the entire client's dataset.
5.	Fair Payments	The smart contract ensures that the server gets payed if and only if all outcome verifiability requirements are satisfied.
6.	Trustless system	We only rely on a TTP, the platform owner, to setup the platform. The platform owner does not have access to the model. If the platform owner does not delete the hidden values, then the platform owner can collude with the service provider and generate fake proofs. However, we assume that the platform owner is honest and deletes the secret values.

Table 6.1.: MLPSP Security Requirements

6.3. IP-MLPSP

6.3.1. Motivation

We have seen that MLPSP satisfies all security requirements except of input privacy. However, for some clients having input privacy is non-negotiable. If we did not care about outcome verifiability, a solution would be to use homomorphic encryption. The client would simply encrypt its dataset and send the encrypted dataset to the service provider, which can then evaluate the encrypted dataset without learning the plaintext value of the dataset and the output. However, we would also like to have outcome verifiability.

Our first idea for creating IP-MLPSP was modifying our CP-SNARKs to work with homomorphic encryption. For example, a server has a secret vector A and the client has a secret matrix B which gets encrypted (we will denote the encryption by Enc(B)) and sent to the server which then calculates the encryption of the extended hadamard product Enc(H). To create a CP-EHad proof, the service provider first needs to commit the ciphertext Enc(H) without knowing the underlying value H. The main difficulty of this approach was modifying PolyCom in such a way that it would be possible to commit ciphertexts. Working with encryptions instead of commitments is not a good idea since the client can see the intermediate computation values by decrypting them with the private key. In this example, if the client knows the values of B and H, then the client can extract A and we lose the model privacy. We also did not want the TTP to be the secret key owner since we want our system to be as trustless as possible. The core of the problem is that the ciphertexts are elements of the group $\mathbb{G}$ and not the field Z_q and the underlying Pedersen commitment scheme is made for committing elements from Z_q. We tried using the ElGamal commitment scheme [28] that is made for committing elements of the group $\mathbb{G}$. However, we have reached a paradox. For the ElGamal commitment scheme to work properly, the decisional Diffie-Hellman problem should be hard. However, to be able to commit polynomials we have to multiply two elements in $\mathbb{G}$: the encrypted polynomial coefficient and the proving key P_i from the commitment key ck. If we are able to do these multiplications then we could solve the decisional Diffie-Hellman problem, which would make the ElGamal commitment scheme not hiding. For this reason, we have abandoned this approach.

6.3.2. Homomorphic Encryption of the Dataset

In our second approach we require an additively homomorphic encryption scheme which encrypts a message $m \in Z_q$ using a randomness $r \in_R Z_q$. We write $Enc(m,r)$ to denote the encryption of m using the randomness r. The additively homomorphic encryption scheme should have the following properties:

1. Multiplication with a plaintext $a \in Z_q$

$$\mathbf{MultPlain}(Enc(m,r), a) = Enc(m,r)^a = Enc(a \cdot m, a \cdot r)$$

2. Addition with a plaintext $a \in Z_q$

$$\mathbf{AddPlain}(Enc(m,r), a) = Enc(m,r) \cdot Enc(a,a) = Enc(m + a, r + a)$$

3. Addition with a ciphertext

$$Enc(m_1, r_1) \cdot Enc(m_2, r_2) = Enc(m_1 + m_2, r_1 + r_2)$$

Such a homomorphic encryption scheme can be implemented using the exponential ElGamal encryption scheme for example.

Now we will show an interesting observation that we made while evaluating a dataset which is encrypted with the aforementioned homomorphic encryption scheme using the SVM model with a linear kernel. We can represent an SVM model with a linear kernel as a tuple $model = (X, Y, B)$. X is a $m \times n$ matrix where m is the number of support vectors and n is the number of features. Every row j of X represents the support vector x_j. The vector Y of length m is defined such that every element $Y_j = t_j \cdot \alpha_j$. The value B corresponds to the bias value. The dataset that should be evaluated is represented by a $n \times k$ matrix Z, where each column i represents a data instance z_i. The prediction of data instance z_i is equal to $f(z_i) = sign(\sum_{j=0}^{m-1} t_j \alpha_j (x_j^T z_i) + b)$. We have already shown that we can evaluate a dataset by computing 4 operations (see Section 3.3.1).

To encrypt the dataset Z, a user has to generate a matrix R of the same size as Z where every element is picked uniformly at random from $\mathbb{Z}_q$. We will write $Enc(Z, R)$ to denote the encrypted matrix and $Enc(Z, R)_{[j,i]}$ to denote the element at row j and column i. Every element $Enc(Z, R)_{[j,i]}$ of the encrypted dataset is equal to $Enc(Z_{j,i}, R_{j,i})$.

We have observed that for every operation f in $f_{SVMLinear}$ we can create a homomorphic version f' which has the following property:

$$f'(Enc(m, r), a) = Enc(f(a, m), f(a, r))$$

1. Homomorphic Matrix Multiplication

$$\begin{aligned} mm'(Enc(Z, R), X)_{[j,i]} &= \prod_{l=0}^{n-1} \textbf{MultPlain}(Enc(Z, R)_{[l,i]}, X_{[j,l]}) \\ &= Enc(\sum_{l=0}^{n-1} X_{[j,l]} \cdot Z_{[l,i]}, \sum_{l=0}^{n-1} X_{[j,l]} \cdot R_{[l,i]}) \\ &= Enc(mm(X, Z), mm(X, R))_{[j,i]} \end{aligned}$$

2. Homomorphic Extended Hadamard Product

$$\begin{aligned} ehad'(Enc(M, R), Y)_{[j,i]} &= \textbf{MultPlain}(Enc(M, R)_{[j,i]}, Y_j) \\ &= Enc(M_{[j,i]} \cdot Y_j, R_{[j,i]} \cdot Y_j) \\ &= Enc(ehad(M, Y), ehad(R, Y))_{[j,i]} \end{aligned}$$

3. Homomorphic Column Summation

$$\begin{aligned} columnSum'(Enc(H,R))_{[i]} &= \prod_{j=0}^{n-1} Enc(H,R)_{[j,i]} \\ &= Enc(\sum_{j=0}^{n-1} H_{[j,i]}, \sum_{j=0}^{n-1} R_{[j,i]}) \\ &= Enc(columnSum(H), columnSum(R))_{[i]} \end{aligned}$$

4. Homomorphic Scalar Addition

$$\begin{aligned} scalarAdd'(Enc(S,R),B)_{[i]} &= \mathbf{AddPlain}((Enc(S,R)_{[i]}, B) \\ &= Enc(S_i + B, R_i + B) \\ &= Enc(scalarAdd(S,B), scalarAdd(R,B))_{[i]} \end{aligned}$$

By combining these homomorphic functions, we can create homomorphic versions of inference functions.

$$f'_{SVMLinear}(Enc(Z,R),X,Y,B) = \\ Enc(f_{SVMLinear}(X,Y,B,Z), f_{SVMLinear}(X,Y,B,R))$$

$$f'_{Linear}(Enc(Z,R),W,B) = \\ Enc(f_{Linear}(W,B,Z), f_{Linear}(W,B,R))$$

For SVM with polynomial kernel we need a fully homomorphic scheme that can support d multiplications where d is the degree of the polynomial kernel. Finding such a scheme will be left for future work. We will only focus on linear and SVM with linear kernel models.

6.3.3. Main Idea

We assume that clients encrypt their datasets with such encryption scheme and that the encryption plaintext and randomness domains are subdomains of PolyCom's input domain.

The main idea of out solution is that a client has a dataset Z_{real} that should remain private. The client creates a fake dataset Z. Since the data is fake, this dataset can be revealed to the service provider. The dataset Z_ω is created by combining Z_{real} and Z. The data instances in Z_ω get permutated and the indices of the fake data instances in Z_ω are memorised in the set I.

The client generates a secret-public key pair and encrypts Z_ω using the private key in the way that was described in the previous subsection and sends $Enc(Z_\omega, R_\omega)$ to the service provider. The service provider executes $f'_{SVMLinear}(Enc(Z_\omega, R_\omega), X, Y, B)$ which is equal to $Enc(f_{SVMLinear}(X,Y,B,Z_\omega), f_{SVMLinear}(X,Y,B,R_\omega))$. The client then uses MLPSP and sends

Z and R to the service provider who runs the MLPSP protocol twice: once for Z and once for R and returns the values $f_{SVMLinear}(X, Y, B, Z)$ and $f_{SVMLinear}(X, Y, B, R)$ back. The smart contract verifies if those two values were correctly computed and if:

$$f'_{SVMLinear}(Enc(Z_\omega, R_\omega), X, Y, B)_{[I]} = Enc(f_{SVMLinear}(X, Y, B, Z), f_{SVMLinear}(X, Y, B, R))$$

If this does not hold then the service provider cheated. If this holds then there is a $\frac{Z}{Z_\omega}$ probability that the service provider did not cheat. This probability is tunable and determined by the client. In the following subsection we will describe in detail how the IP-MLPSP protocol works

6.3.4. Design

The protocol differs from MLPSP in two ways. First, when the client joins the platform, a private-public key pair has to be generated and published on the blockchain. Secondly, in MLPSP the server and client have one round of communication: the client registers the request and the server registers the proof. In IP-MLPSP the client and service provider have two rounds of communication. In the first round the client registers the request and then the service provider evaluates the encrypted dataset and registers the output from this round. In the second round, the client reveals the index set I and then the service provider evaluates the fake dataset and creates a proof of correct computation and registers the proof which contains the verified output from round two. In Figure 6.8 we have shown the datatypes that are stored on the blockchain. When compared to MLPSP, we have the following differences: (1) Every client stores its public key, (2) the request contains an additional field for the encryption randomness commitment, (3) there is a new data type called Index which contains a unique request ID and the set of all dummy data indices, (4) the proof now contains two proofs of correct computation, one for the dummy dataset Z and one for the encryption randomness of the dummy dataset R and the homomorphically evaluated predictions of the entire dataset.

<table>
<tr><td>Commitment key ck</td></tr>
<tr><td>Client's encryption public keys</td></tr>
<tr><td>Model:
• model ID
• model type
• model parameters commitments $\{c_{param}\}$
• prediction fee</td></tr>
<tr><td>Request:
• request ID
• model ID
• dataset commitment c_Z
• encryption randomness commitment c_R</td></tr>
<tr><td>Output:
• request ID
• hash of the output</td></tr>
<tr><td>Index:
• request ID
• set of all dummy data indices I</td></tr>
<tr><td>Proof:
• request ID
• for dataset Z
  • proof π
  • output commitment c_O
  • output O
  • output opening o_O
• for randomness R
  • proof π
  • output commitment c_O
  • output O
  • output opening o_O
• output</td></tr>
</table>

Figure 6.8.: IP-MLPSP Data Types

There are nine actions that can be performed on the platform:

1. Platform Setup
2. Model Registration
3. Client Registration
4. Request Registration
5. Output Registration
6. Index Registration
7. Proof Registration
8. Verification

9. Output Retrieval

The platform setup and model registration are the same as in MLPSP and for the other actions we will provide a more detailed explanation.

Client Registration

To join the platform, the client has to generate a public-private key pair and register the public key on the blockchain.

Request Registration

The client wants to evaluate its sensitive dataset Z_{real}. First, the client decides what the outcome verifiability probability should be and based on that generates the dummy dataset Z. The two datasets get merged into a dataset that we will denote with Z_ω and the indices of the dummy data points in Z_ω are kept in the set denoted with I. To encrypt Z_ω, the client generates the encryption randomness matrix R_ω of the same size as Z_ω. We will denote with R the randomnesses used for encrypting the dummy data points $R_{\omega[I]}$.

The client commits Z and R and creates a request which gets stored on the blockchain. Then the client sends $Enc(Z_\omega, R_\omega)$ to the service provider. The request registration process can be seen in Figure 6.9.

Input: (request ID, model ID, $Z_\omega = Z + Z_{real}$, I)

1. Generate encryption randomness $R_\omega = R + R_{real}$

2. For $X = \{Z, R\}$
   - Transform X into a polynomial
   - PolyCom.Commit$(ck, X, p) \rightarrow (c_X, o_X)$

3. Set Request = (request ID, model ID, c_Z, c_R)
4. blockchain.SetRequest(Request)
5. Deposit fee
6. Send request: (request ID, $Enc(Z_\omega, R_\omega)$) to the service provider

Figure 6.9.: IP-MLPSP Request Registration

Output Registration

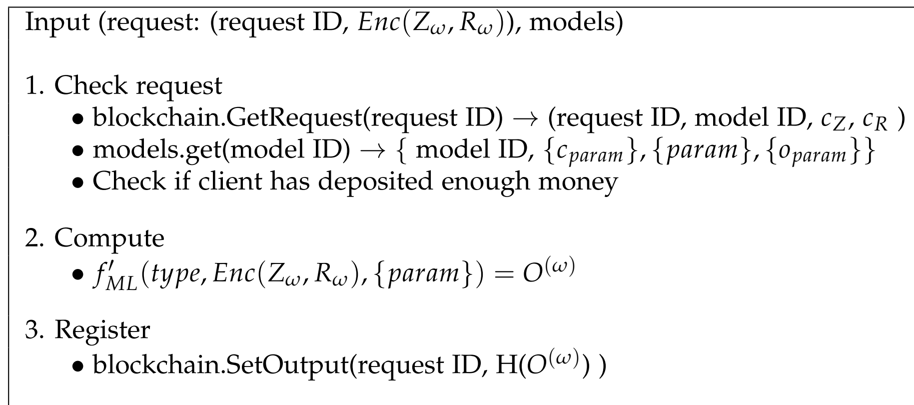
The service provider receives the encrypted dataset, checks if the request is correct which involves checking if the client deposited enough money and if the service provider owns the specified model, evaluates the encrypted dataset to obtain encrypted predictions and puts the hash of the encrypted predictions on the blockchain.

The service provider does not know which and how many data instances from that dataset are going to be checked later in the protocol. If the service provider gets caught cheating

there will be no payment. Some other penalties may be introduced for a cheating service provider. For example, the service provider might get removed from the system or be required to pay a fine.

The service provider is required to put the hash of the encrypted predictions on the blockchain to continue the protocol. This ensures that the service provider does the computation before finding out which data instances will be checked for correctness. At the end of the protocol, the service provider reveals the encrypted predictions and the smart contract will check if the hash is matching. In other worlds, the smart contract checks if the service provider performed the computation before learning which instances are fake.

We avoid putting the encrypted predictions on the blockchain at this moment to protect the honest service provider. If encrypted predictions were put on the blockchain, the client could decrypt them and obtain the correct predictions without paying. The operations to be performed can be seen in Figure 6.10.


Input (request: (request ID, $Enc(Z_\omega, R_\omega)$), models)

1. Check request
   - blockchain.GetRequest(request ID) $\rightarrow$ (request ID, model ID, c_Z, c_R)
   - models.get(model ID) $\rightarrow$ { model ID, $\{c_{param}\}, \{param\}, \{o_{param}\}$}
   - Check if client has deposited enough money

2. Compute
   - $f'_{ML}(type, Enc(Z_\omega, R_\omega), \{param\}) = O^{(\omega)}$

3. Register
   - blockchain.SetOutput(request ID, H($O^{(\omega)}$))

Figure 6.10.: IP-MLPSP Output Registration

Index Registration

After the service provider registers the output, the client has to reveal the dummy index set I. If the client does not upload the index in a pre-defined time interval since the service provider registered the output, the money will be transferred to the service provider. This is to assure that the client does not spam the server with many fake request in which case the service provider will do a lot of work without getting paid for it. The index registration process in shown in Figure 6.11. We have identified a limitation in the index registration action. The client could register the index on the blockchain and not send the request to the service provider which will make the service provider unable to generate the proof of correct computation and get paid. This problem will be left for future work.

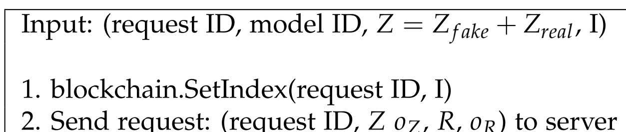
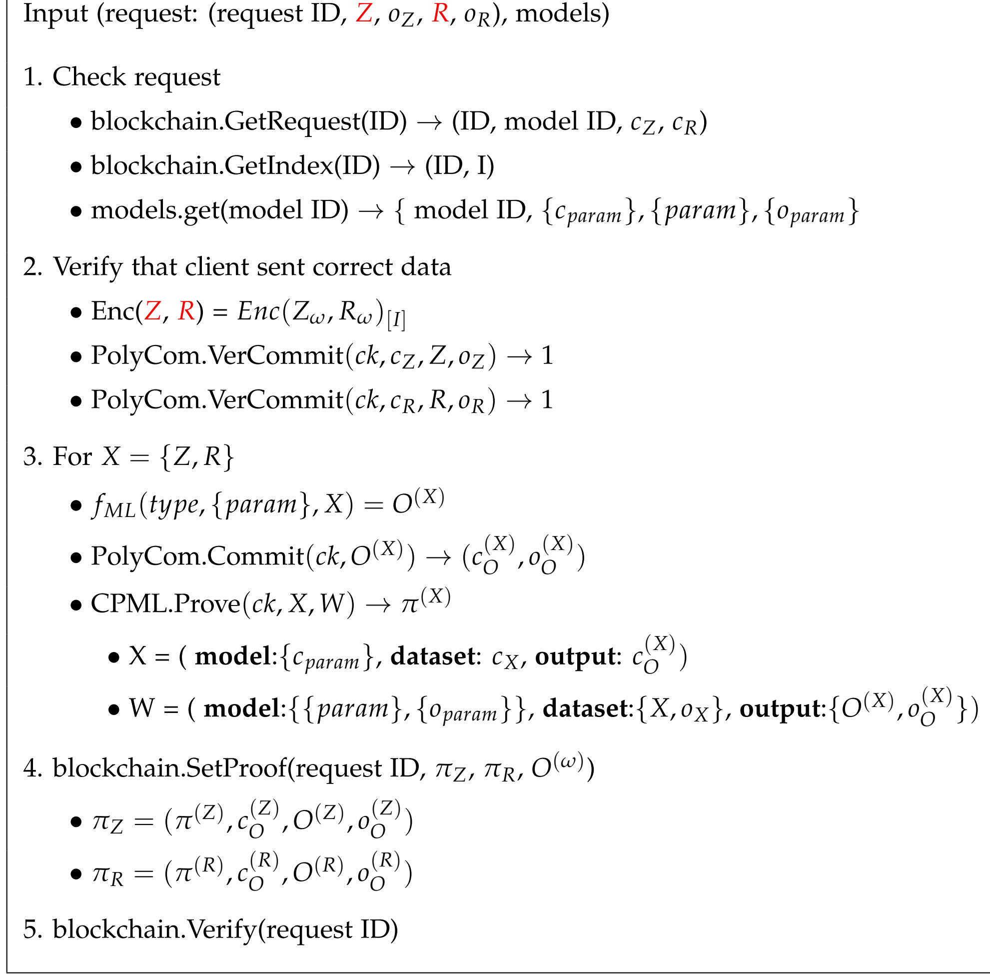


Figure 6.11.: IP-MLPSP Index Registration

Proof Registration

The service provider checks if the values Z and R correspond to the encrypted data instances at positions specified in I. Then, the service provider runs the MLPSP proof registration process twice, once with Z as input and once with R as input. Additionally, the service provider also puts the encrypted predictions calculated in the first round on the blockchain. The entire process can be seen in Figure 6.12.

Note that the encryption scheme is not binding for the secret key owner, in other words, the client could generate the pair (Z', R') such that $Enc(Z', R') = Enc(Z, R)$ and commit Z' and R'. However if $Enc(Z', R') = Enc(Z, R)$, then $Enc(O^{(Z)}, O^{(R)}) = Enc(O^{(Z')}, O^{(R')})$. Therefore, even if the client sends different values, but the ciphertexts are the same, the service provider will still be able to generate correct proofs.



Input (request: (request ID, Z, o_Z, R, o_R), models)

1. Check request
   - blockchain.GetRequest(ID) $\rightarrow$ (ID, model ID, c_Z, c_R)
   - blockchain.GetIndex(ID) $\rightarrow$ (ID, I)
   - models.get(model ID) $\rightarrow$ { model ID, $\{c_{param}\}, \{param\}, \{o_{param}\}$ }
2. Verify that client sent correct data
   - Enc(Z, R) = $Enc(Z_\omega, R_\omega)_{[I]}$
   - PolyCom.VerCommit$(ck, c_Z, Z, o_Z) \rightarrow 1$
   - PolyCom.VerCommit$(ck, c_R, R, o_R) \rightarrow 1$
3. For $X = \{Z, R\}$
   - $f_{ML}(type, \{param\}, X) = O^{(X)}$
   - PolyCom.Commit$(ck, O^{(X)}) \rightarrow (c_O^{(X)}, o_O^{(X)})$
   - CPML.Prove$(ck, X, W) \rightarrow \pi^{(X)}$
     - X = (**model**:$\{c_{param}\}$, **dataset**: c_X, **output**: $c_O^{(X)}$)
     - W = (**model**:$\{\{param\}, \{o_{param}\}\}$, **dataset**:$\{X, o_X\}$, **output**:$\{O^{(X)}, o_O^{(X)}\}$)
4. blockchain.SetProof(request ID, π_Z, π_R, $O^{(\omega)}$)
   - $\pi_Z = (\pi^{(Z)}, c_O^{(Z)}, O^{(Z)}, o_O^{(Z)})$
   - $\pi_R = (\pi^{(R)}, c_O^{(R)}, O^{(R)}, o_O^{(R)})$
5. blockchain.Verify(request ID)

Figure 6.12.: IP-MLPSP Proof Registration

Verification

The verification smart contract gets invoked by the service provider. The only input is the request ID and all other data is fetched from the blockchain using that request ID. The smart contract verifies the MLPSP proof for Z and R and runs two more checks. First, the smart contract makes sure that the returned encrypted predictions have been generated in the first round, before the dummy index set was revealed. Secondly, the smart contract assures that the verified output for the dummy data is the same as the unverified output. To do this the smart contract checks if:

$$O_{[I]}^{(\omega)} = Enc(O^{(Z)}, O^{(R)})$$
$$f'_{ML}(Enc(Z, R), \{param\}) = Enc(f_{ML}(Z, \{param\}), f_{ML}(R, \{param\})$$

This equation should hold because of the homomorphic encryption property discussed before. If all checks have passed, the the smart contract transfers the client's deposited money to the service provider. Otherwise, the deposited money is returned back to the client. The verification process can be seen in Figure 6.13

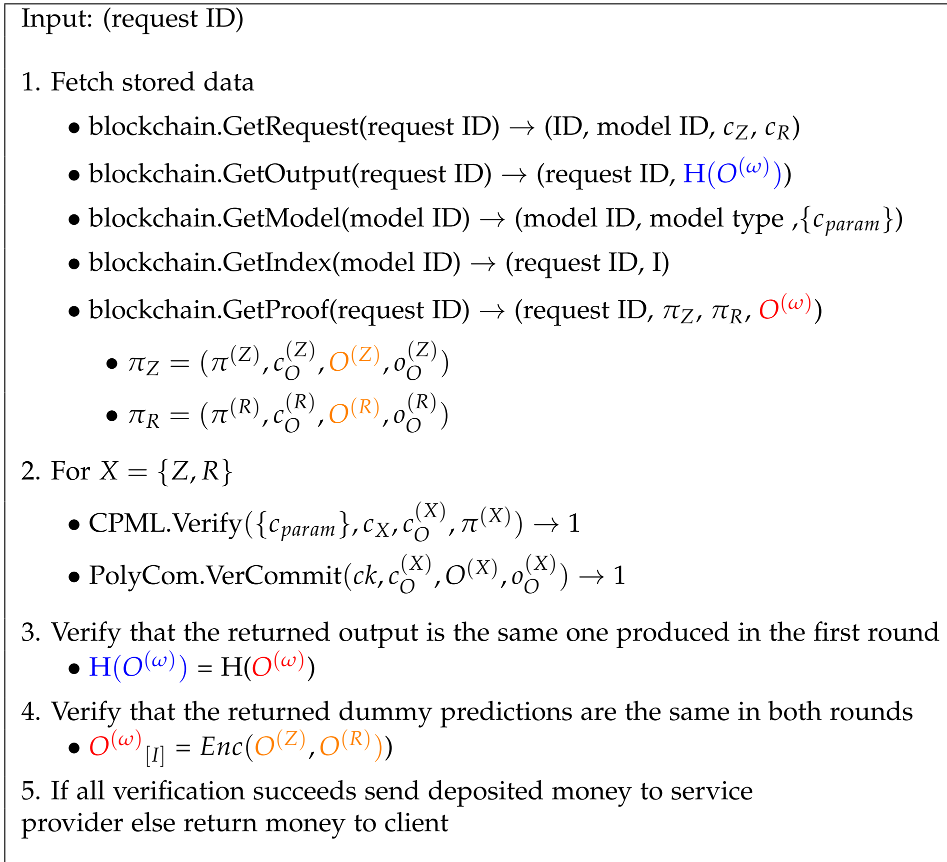


Figure 6.13.: IP-MLPSP Verification

Output Retrieval

Once the verification is complete, the client can download the encrypted predictions and use the secret key to decrypt them.

6.3.5. Security Discussion

The security discussion is shown in Table 6.2.

1.	Model Privacy	Same as in MLPSP
2.	Input Privacy	The client achieves input privacy since the dataset Z_{real} is never revealed in plaintext. No one can decrypt the data since the secret key does not leave the client
2. a)	Output Privacy	The client achieves output privacy since the predictions of Z_{real} are never revealed in plaintext. No one can decrypt the predictions since the secret key does not leave the client
3.	Outcome Verifiability	The outcome verifiability is probabilistic, and that holds for all sub-requirements as well.
4.	Batch Verification	It can be seen that there is one proof of correct predictions for the entire client's dataset.
5.	Fair Payments	The fair payments are probabilistic.
6.	Trustless system	Same as in MLPSP since the role of the platform owner is the same.

Table 6.2.: IP-MLPSP Security Requirements

7. Discussion

7.1. Comparison of IP-MLPSP with Related Work: Advantages

7.1.1. Trust Assumptions

One of our goals when building IP-MLPSP was to have a trustless system, a system where the users do not have to trust each other. That is the main reason why our scheme is probabilistic. The only trust assumption that we require in our system is that the platform owner deletes the secret values generated during the commitment key generation. If the platform owner fails to do so, the proofs can be forged, which negatively impacts the clients. MVP [12] has the same problem and they offer the following solution which can also be applied to IP-MLPSP: the commitment key can be generated collectively by the clients since they have no real incentive to leak the secret values. It should be noted that even if the platform owner leaks the key, the input and model privacy stay protected.

In MVP , the input privacy relies on the honesty of the TTP. The TTP owns the secret key which can decrypt the client's data and the TTP decrypts the client's predictions. In IP-MLPSP, client's secret dataset and predictions are not revealed to anyone.

The system model of [20] is a bit different then ours and we can analyze the trust assumptions from two viewpoints. A honest service provider has two goals: to protect its model from the client and to be sure that the cloud providers computed the inference correctly. Regarding the model privacy, the service provider sends the model to the untrusted cloud providers and assumes that they will not leak this model to the client or, more generally, to the world. Regarding the verifiability, the service provider does not have to rely on the honesty of the cloud providers. The client, on the other hand, wants to protect its dataset from the service provider and be sure that the service provider computed the predictions correctly. However, the dataset is only encoded, not encrypted, and if more than a certain number of servers from the first cloud provider collude they can retrieve the client's dataset and send it to the service provider. The output privacy is not protected at all, it can be seen by the third cloud provider. Regarding the verifiability, the service provider generates the commitment key and can forge proofs. Thus the client has to trust the service provider to be honest.

In VPMLP [21], the client assumes that the edge and cloud server will honestly compute the proof, that is, they will not cheat since the purpose of the proof of correct computation is to be sure that there were no external temperings with the data.

7.1.2. Model Extraction Attack

Every ML prediction service is susceptible to a model extraction attacks. A client that makes a certain number of request with different inputs will finally be able to recover the model. The simpler the model, less requests are needed. This issue was addressed in VPMLP [21] where they suggest that this problem can be solved by limiting the number of request that a client can make to one service provider.

We argue that polynomial-kernel MVP is more susceptible to the attack then polynomial kernel IP-MLPSP. In MVP, the service provider computes the dot product of every support vector x_j and of every data instance z_i and this intermediate computation value is seen by the client. If the number of data instances is equal to or greater then the number of support vectors the model can be extracted just like how we have shown in the following example:

Example 21. If the support vector matrix is $X = \begin{bmatrix} 2 & 4 \end{bmatrix}$ and the client knows the values of its dataset $\begin{bmatrix} 4 & 2 \\ 5 & 3 \end{bmatrix}$ and the dot products $\begin{bmatrix} 28 & 16 \end{bmatrix}$ then the client can extract the support vector matrix by solving the system of linear equations:

$$4 \cdot x_1 + 5 \cdot x_2 = 28$$
$$2 \cdot x_1 + 3 \cdot x_2 = 16$$

In IP-MLPSP, the client only sees the final prediction, which makes it much harder to extract the model.

7.1.3. Batch Verification

When it comes to batch verification, we argue that IP-MLPSP is better than the scheme from [20]. In their work there is one CP-MM proof for every server in the first cloud provider and one CP-Had proof for very data instance in the client's dataset. Additionally, they do not prove the correct computation of the column summation and scalar addition operation which we do.

7.2. Comparison of IP-MLPSP with Related Work: Disadvantages

The main disadvantage of IP-MLPSP compared to MVP, the scheme from [20] and VPMLP is that the verifiability is tunably probabilistic, whereas in the other schemes the verifiability is definite. Another disadvantage that follows from the first disadvantage is that we have to add dummy data to the client's dataset, thus the workload that the service provider has is higher then if we managed to add homomorphic encryption directly to MLPSP. The third potential disadvantage is that IP-MLPSP works for polynomial-kernel SVM models only under the assumption that there exists a leveled homomorphic scheme with property defined in Section 6.3.2. We have not yet found such a scheme, it will be left for future work.

7.3. Future Work

We have identified three main functionalities that we would like to add or improve. (1) The CP-ScalarAdd scheme should be extended to also support matrices where the number of rows and columns is not a power of two. (2) In IP-MLPSP we need a mechanism which can automatically detect if the client has sent the request to the service provider after registering the index set. (3) An adequate leveled homomorphic scheme should be found that we can use in IP-MLPSP for polynomial-kernel SVM.

After addressing these functionalities, there are some steps that we could undertake to improve our work. (1) We could evaluate CP-Linear, CP-SVMLinear and CP-Poly with real ML models and datasets instead of generating random values and compare the efficiency to the related work. (2) We could implement MLPSP and IP-MLPSP with a blockchain and homomorphic encryption scheme and measure the runtime and storage. (3) We could investigate which other ML models could be used in our platforms. (4) We could implement the distributed commitment key generation by the clients and remove the TTP completely.

8. Conclusion

In this research work, the goal was to create a ML prediction service platform where users that have managed to train powerful ML models could offer prediction services, meaning that that they will make predictions on client's datasets with their ML models in exchange for financial compensation. Our initial goal was to create a platform which simultaneously ensures (1) model privacy, meaning that the ML models stay secret, (2) input privacy which entails that the client's datasets and predictions stay secret, (3) outcome verifiability that ensures that the returned predictions are correctly computed, (4) batch verification, meaning that there is one proof for the entire dataset, (5) fair payments and (6) trustlessnes which we defined as our platform requiring less trust assumptions than related work.

To make our platform trustless and implement fair payments we have used the blockchain technology. For achieving input privacy we have used homomorphic encryption. For achieving model privacy, outcome verifiability and batch verification we have decided to use the LegoSNARK[15] framework. However, after many attempts to integrate homomorphic encryption with LegoSNARK, we claim that it is not possible to use a homomorphic encryption scheme together with LegoSNARK because of incompatibility of the polynomial commitment scheme and the homomorphic encryption scheme. Therefore, we have created two platforms: MLPSP which achieves all security requirements except of input privacy and IP-MLPSP which achieves input and model privacy but at the expense of outcome verifiability being tunably probabilistic. To create these two platforms we have created four schemes for proving basic operations on committed matrices, vectors and scalars. By combining these schemes we have created three more schemes: (1) CP-Linear which can be used to prove correct linear ML inference without revealing the model, (2) CP-SVMLinear which can be used to prove correct linear-kernel SVM inference without revealing the model and (3) CP-SVMPoly which can be used to prove correct polynomial-kerrnel SVM inference without revealing the model. These 7 schemes can be reused in other verifiable computation works.

A. Pedersen Commitment Scheme

1. **Correctness**: It is obviously correct.
2. **Binding**: If a malicious user found two pairs (u, o) and (u', o') such that $Commit(u, o) = Commit(u', o')$ then the discrete log problem $dlog_g h$ will be solved which is assumed to be hard in $\mathbb{G}$, in the following way:

$$\begin{aligned} Commit(u, o) &= Commit(u', o') \\ g^u h^o &= g^{u'} h^{o'} \\ g^u (g^x)^o &= g^{u'} (g^x)^{o'} \\ g^{u+xo} &= g^{u'+xo'} \\ u + xo &= u' + xo' \\ xo - xo' &= u' - u \\ x(o - o') &= u' - u \\ x &= \frac{u' - u}{o - o'} \end{aligned}$$

   Therefore, if the discrete log problem in $\mathbb{G}$ was not hard the binding property would not hold.
3. **Hiding**: Given a commitment c, there is a value $o \in Z_q$ for every possible message $u \in Z_q$ such that $c = g^u h^o$. Even if the discrete log problem in $\mathbb{G}$ was not hard the hiding property would still hold.

B. ZK$_{eq}$

The Σ protocol satisfies the notions of:

1. **Completeness**: If c_1 and c_2 contain the same pre-image and the prover knows the witness $w = (o_1, o_2, u)$ and executes the protocol correctly, then a honest verifier will accept the proof with probability 1.

$$a \cdot c_1^c =^? g^{z_1} h^{z_2} \qquad b \cdot c_2^c =^? g^{z_1} h^{z_3}$$
$$g^{o_3} h^{o_4} g^{cu} h^{co_1} = g^{cu} g^{o_3} h^{co_1} h^{o_4} \qquad g^{o_3} h^{o_5} g^{cu} h^{co_2} = g^{cu} g^{o_3} h^{co_2} h^{o_5}$$
$$g^{o_3+cu} h^{o_4+co_1} = g^{cu+o_3} h^{co_1+o_4} \qquad g^{o_3+cu} h^{o_5+co_2} = g^{cu+o_3} h^{co_2+o_5}$$

2. **Special Soundness**: Given two accepting transcripts with distinct challenges and same initial messages (a, b, c, z_1, z_2, z_3) and $(a, b, c', z_1', z_2', z_3')$, the witness $w = (o_1, o_2, u)$ can be extracted.

$$o_1 = \frac{(z_2' - z_2)}{c - c'} = \frac{o_1(c' - c)}{c' - c}$$
$$o_2 = \frac{(z_3' - z_3)}{c' - c} = \frac{o_2(c' - c)}{c' - c}$$
$$u = \frac{(z_1' - z_1)}{c' - c} = \frac{c' * u + o_3 - c * u - o_3}{c' - c} = \frac{u(c' - c)}{c' - c}$$

3. **Special Honest Verifier Zero-Knowledge**: Given a challenge from a public-coin honest verifier, a simulator can create an accepting transcript (a, b, c, z_1, z_2, z_3) without knowing the witness $w = (o_1, o_2, u)$

$$S(c) :=$$
$$z_1, z_2, z_3 \in_R \mathbb{G}$$
$$a = g^{z_1} h^{z_2} c_1^{-c}$$
$$b = g^{z_1} h^{z_3} c_2^{-c}$$

C. ZK$_{prod}$

The Σ protocol satisfies the notions of:

1. **Completeness**: If c_3 contains the same pre-image which is a product of pre-images of c_1 and c_2 and the prover knows the witness $w = (u_1, u_2, o_1, o_2, o_3)$ and executes the protocol correctly, then a honest verifier will accept the proof with probability 1.

$$\begin{aligned} g^{f_1}h^{z_1} &= c_1^c \cdot c_d \\ g^{u_1c+d}h^{o_1c+o_4} &= g^{u_1c}h^{o_1c}g^dh^{o_4} \\ g^{u_1c+d}h^{o_1c+o_4} &= g^{u_1c+d}h^{o_1c+o_4} \end{aligned}$$

$$\begin{aligned} g^{f_2}h^{z_2} &= c_2^c \cdot c_e \\ g^{u_2c}g^eh^{o_2c+o_5} &= g^{cu_2}h^{co_2}g^e + h^{o_5} \\ g^{u_2c+e}h^{o_2c+o_5} &= g^{cu_2+e}h^{o_2c+o_5} \end{aligned}$$

$$\begin{aligned} c_2^{f_1}h^{z_3} &= c_3^c \cdot c_d' \\ g^{u_2(u_1c+d)}h^{o_2(u_1c+d)}h^{co_3-cu_1o_2+o_6} &= g^{cu_1u_2}h^{co_3}g^{du_2}h^{dr_2+r_6} \\ g^{cu_1o_2+u_2d}h^{o_2d+co_3+o_6} &= g^{cu_1u_2+du_2}h^{co_3+do_2+o_6} \end{aligned}$$

2. **Special Soundness**: Given two accepting transcripts with distinct challenges and same initial messages $(c_d, c_e, c_d', c, f_1, f_2, z_1, z_2, z_3)$ and $(c_d, c_e, c_d', c', f_1', f_2', z_1', z_2', z_3')$, the witness $w = (u_1, u_2, o_1, o_2, o_3)$ can be extracted.

$$\begin{aligned} u_1 &= \frac{f_1 - f_1'}{c - c'} = \frac{u_1c + d - u_1c' - d}{c - c'} = \frac{u_1(c - c')}{c - c'} \\ u_2 &= \frac{f_2 - f_2'}{c - c'} = \frac{u_2c + e - u_2c' - e}{c - c'} = \frac{u_2(c - c')}{c - c'} \\ o_1 &= \frac{(z_1' - z_1)}{c - c'} = \frac{o_1c' + o_4 - o_1c - o_4}{c' - c} = \frac{o_1(c' - c)}{c' - c} \\ o_2 &= \frac{(z_2' - z_2)}{c - c'} = \frac{o_2c' + o_5 - o_2c - o_5}{c' - c} = \frac{o_2(c' - c)}{c' - c} \\ o_3 &= \frac{z_3' - z_3}{c - c'} + u_1o_2 \\ &= \frac{c(o_3 - c_1o_2) + o_6 - c'(o_3 - u_1o_2) - o_6}{c - c'} + u_1o_2 \\ &= \frac{(o_3 - u_1o_2)(c' - c)}{c' - c} + u_1o_2 \\ &= o_3 - u_1o_2 + u_1o_2 \end{aligned}$$

3. **Special Honest Verifier Zero-Knowledge**: Given a challenge from a public-coin honest verifier, a simulator can create an accepting transcript $(c_d, c_e, c'_d, c, f_1, f_2, z_1, z_2, z_3)$ without knowing the witness $w = (u_1, u_2, o_1, o_2, o_3)$

$$S(c) :=$$
$$f_1, f_2 \in_R \mathbb{Z}_q$$
$$z_1, z_2, z_3 \in_R \mathbb{G}$$
$$c_d = g^{f_1} h^{z_1} c_1^{-c}$$
$$c_e = g^{f_2} h^{z_2} c_2^{-c}$$
$$c'_d = c_2^{f_1} h^{z_3} c_3^{-c}$$

D. CP-MM

Public statement: (c_A, c_B, c_C)

Prover

Witness: (A, B, C, o_A, o_B, o_C)

1. Generate public input
   - $H(c_A, c_B, c_C) \to I, J$
2. Evaluate and prove C on (I, J)
   - $\tilde{C}(I, J) \to c$
   - PolyCom.Commit$(ck, c, v) \to (c_c, o_c)$
   - CP-Poly.Prove$(x = ((I, J), c_C, c_c), w = (C, c, o_C, o_c)) \to \pi_{poly_c}$
3. Run the sumcheck protocol
   - $f(X) = \tilde{A}(I, X) \cdot \tilde{B}(X, J)$

   CP-Sumcheck'.Prove$(x = (c_c), w = (c, o_c, f)) \to (R, c_l, o_l, \pi_{sc})$
4. Evaluate and prove A on (I, R)
   - $\tilde{A}(I, R) \to a$
   - PolyCom.Commit$(ck, a, v) \to (c_a, o_a)$
   - CP-Poly.Prove$(x = ((I, R), c_A, c_a), w = (A, a, o_A, o_a)) \to \pi_{poly_a}$
5. Evaluate and prove B on (R, J)
   - $\tilde{B}(R, J) \to b$
   - PolyCom.Commit$(ck, b, v) \to (c_b, o_b)$
   - CP-Poly.Prove$(x = ((R, J), c_B, c_b), w = (B, b, o_B, o_b)) \to \pi_{poly_b}$
7. Create a product proof
   - ZK_{prod}.Prove$(a, b, o_a, o_b, o_l, c_a, c_b, c_l) \to \pi_{prod}$

output: $\pi = (c_c, \pi_{poly_c}, \pi_{sc}, c_a, \pi_{poly_a}, c_b, \pi_{poly_b}, \pi_{prod})$

Figure D.1.: CP-MM.Prove

Public statement: (c_A, c_B, c_C)
Verifier Input: $\pi = (c_c, \pi_{poly_c}, \pi_{sc}, c_a, \pi_{poly_a}, c_b, \pi_{poly_b}, \pi_{prod})$ 1. Generate public input • $H(c_A, c_B, c_C) \to I, J$ 2. **Output** 1 if: • CP-Poly.Verify$(\pi_{poly_c}, (I, J), c_C, c_c) \to 1$ • CP-Sumcheck'.Verify$(\pi_{sc}, c_c) \to (1, R, c_l)$ • CP-Poly.Verify$(\pi_{poly_a}, (I, R), c_A, c_a) \to 1$ • CP-Poly.Verify$(\pi_{poly_b}, (R, J), c_B, c_b) \to 1$ • ZK_{prod}.Verify$(c_a, c_b, c_l, \pi_{prod}) \to 1$, else **output** 0

Figure D.2.: CP-MM.Verify

E. CP-EHad

Public statement: (c_A, c_B, c_C)

Prover
Witness: (A, B, C, o_A, o_B, o_C)

1. Generate public input
   - $H(c_A, c_B, c_C) \rightarrow I, J$
2. Evaluate and prove C on (I, J)
   - $\tilde{C}(I, J) \rightarrow c$
   - PolyCom.Commit$(ck, c, v) \rightarrow (c_c, o_c)$
   - CP-Poly.Prove$(x = ((I, J), c_C, c_c), w = (C, c, o_C, o_c)) \rightarrow \pi_{poly_c}$
3. Run the sumcheck protocol
   - $f(X_r, X_c) = \tilde{eq}(I, J, X_r, X_c) \times \tilde{A}(X_r, X_c) \times \tilde{B}(X_r)$
   - CP-Sumcheck'.Prove$(x = (c_c), w = (c, o_c, f)) \rightarrow ((R_r, R_c), c_l, o_l, \pi_{sc})$
4. Evaluate and prove A on (R_r, R_c)
   - $\tilde{A}(R_r, R_c) \rightarrow a$
   - PolyCom.Commit$(ck, a, v) \rightarrow (c_a, o_a)$
   - CP-Poly.Prove$(x = ((R_r, R_c), c_A, c_a), w = (A, a, o_A, o_a)) \rightarrow \pi_{poly_a}$
5. Evaluate and prove B on (R_r)
   - $\tilde{B}(R_r) \rightarrow b$
   - PolyCom.Commit$(ck, b, v) \rightarrow (c_b, o_b)$
   - CP-Poly.Prove$(x = (R_r, c_B, c_b), w = (B, b, o_B, o_b)) \rightarrow \pi_{poly_b}$
6. Create a product proof
   - $\tilde{eq}(I, J, R_r, R_c) \rightarrow e$
   - ZK_{prod}.Prove$(a, b \cdot e, o_a, o_b \cdot e, o_l, c_a, c_b^e, c_l) \rightarrow \pi_{prod}$

output: $\pi = (c_c, \pi_{poly_c}, \pi_{sc}, c_a, \pi_{poly_a}, c_b, \pi_{poly_b}, \pi_{prod})$

Figure E.1.: CP-EHad.Prove

Public statement: (c_A, c_B, c_C)
Verifier Input: $\pi = (c_c, \pi_{poly_c}, \pi_{sc}, c_a, \pi_{poly_a}, c_b, \pi_{poly_b}, \pi_{prod})$ 1. Generate public input • $H(c_A, c_B, c_C) \rightarrow I, J$ 3. **Output** 1 if: • CP-Poly.Verify$(\pi_{poly_c}, (I, J), c_C, c_c) \rightarrow 1$ • CP-Sumcheck'.Verify$(\pi_{sc}, c_c) \rightarrow (1, (R_r, R_c), c_l)$ • Compute $\tilde{eq}(I, J, R_r, R_c) \rightarrow e$ • CP-Poly.Verify$(\pi_{poly_a}, (R_r, R_c), c_A, c_a) \rightarrow 1$ • CP-Poly.Verify$(\pi_{poly_b}, R_r, c_B, c_b) \rightarrow 1$ • ZK_{prod}.Verify$(c_a, c_b^e, c_l, \pi_{prod}) \rightarrow 1$, else **output** 0

Figure E.2.: CP-EHad.Verify

F. CP-Expo

Public statement: (d, c_A, c_B)

Prover
Witness: (A, B, o_A, o_B)

1. Generate public input
   - $H(c_A, c_B) \rightarrow I, J$
2. Evaluate and prove B on (I, J)
   - $\tilde{B}(I, J) \rightarrow b$
   - PolyCom.Commit$(ck, b, v) \rightarrow (c_b, o_b)$
   - CP-Poly.Prove$(x = ((I, J), c_B, c_b), w = (B, b, o_B, o_b)) \rightarrow \pi_{poly_b}$
3. Run the sumcheck protocol
   - $f(X_r, X_c) = \tilde{eq}(I, J, X_r, X_c) \times (\tilde{A}(X_r, X_c))^d$
   - CP-Sumcheck'.Prove$(x = (c_b), w = (b, o_b, f)) \rightarrow ((R_r, R_c), c_l, o_l, \pi_{sc})$
4. Evaluate and prove A on (R_r, R_c)
   - $\tilde{A}(R_r, R_c) \rightarrow a$
   - PolyCom.Commit$(ck, a, v) \rightarrow (c_a, o_a)$
   - CP-Poly.Prove$(x = ((R_r, R_c), c_A, c_a), w = (A, a, o_A, o_a)) \rightarrow \pi_{poly_a}$
5. $\forall i \in \{2, ..., d\}$
   - PolyCom.Commit$(ck, a^i, v) \rightarrow (c_{a^i}, o_{a^i})$

     Note that: $c_{a^i} = (c_{a^{i-1}})^a h^{o_{a^i} - ao_{a^{(i-1)}}}$
   - ZK'_{eq}.Prove$(g_1, g_2, u, o_1, o_2, c_1, c_2) \rightarrow \pi_{a^i}$
     - $g_1 = g$ and $g_2 = c_{a^{i-1}}$
     - $u = a$
     - $o_1 = o_a$ and $o_2 = o_{a^i} - ao_{a^{(i-1)}}$
     - $c_1 = c_a$ and $c_2 = c_{a^i}$

6.Create a product proof
   - $\tilde{eq}(I, J, R_r, R_c) \rightarrow e$
   - ZK_{eq}.Prove$(e \cdot a^d, e \cdot o_{a^d}, o_l, (c_{a^d})^e, c_l) \rightarrow \pi_{eq}$

output: $\pi = (c_b, \pi_{poly_b}, \pi_{sc}, c_a, \pi_{poly_a}, c_{a^2}, ..., c_{a^d}, \pi_{a^2}, ..., \pi_{a^d}, \pi_{eq})$

Figure F.1.: CP-Expo.Prove

Public statement: (d, c_A, c_B)

Verifier

Input: $\pi = (c_b, \pi_{poly_b}, \pi_{sc}, c_a, \pi_{poly_a}, c_{a^2}, ..., c_{a^d}, \pi_{a^2}, ..., \pi_{a^d}, \pi_{eq})$

1. Generate public input
   - $H(d, c_A, c_B) \to I, J$
2. **Output** 1 if:
   - CP-Poly.Verify$(\pi_{poly_b}, (I, J), c_B, c_b) \to 1$
   - CP-Sumcheck'.Verify$(\pi_{sc}, c_b) \to (1, (R_r, R_c), c_l)$
     - Compute $\tilde{eq}(I, J, R_r, R_c) \to e$
   - CP-Poly.Verify$(\pi_{poly_a}, (R_r, R_c), c_A, c_a) \to 1$
   - $\forall i \in \{2, .., d\}$
     - ZK'_{eq}.Verify$(g_1, g_2, c_1, c_2, \pi_{a^i}) \to 1$
       - $g_1 = g$ and $g_2 = c_{a^{i-1}}$
       - $c_1 = c_a$ and $c_2 = c_{a^i}$
   - ZK_{eq}.Verify$((c_{a^d})^e, c_l, \pi_{eq}) \to 1$

, else **output** 0

Figure F.2.: CP-Expo.Verify

G. CP-ColumnSum

Public statement: (c_A, c_B)

Prover
Witness: (A, B, o_A, o_B)

1. Generate public input
   - $H(c_A, c_B) \to J$
2. Evaluate and prove B on J
   - $\tilde{B}(J) \to b$
   - PolyCom.Commit$(ck, b, v) \to (c_b, o_b)$
   - CP-Poly.Prove$(x = (J, c_B, c_b), w = (B, b, o_B, o_b)) \to \pi_{poly_b}$
3. Run the sumcheck protocol
   - $f(X) = \tilde{A}(X, J)$
   - CP-Sumcheck'.Prove$(x = (c_b), w = (b, o_b, f)) \to (R, c_l, o_l, \pi_{sc})$
4. Evaluate and prove A on (R, J)
   - $\tilde{A}(R, J) \to a$
   - PolyCom.Commit$(ck, a, v) \to (c_a, o_a)$
   - CP-Poly.Prove$(x = ((R, J), c_A, c_a), w = (A, a, o_A, o_a)) \to \pi_{poly_a}$

6.Create an equality proof
   - ZK_{eq}.Prove$(a, o_a, o_l, c_a, c_l) \to \pi_{eq}$

output: $\pi = (c_a, \pi_{poly_a}, \pi_{sc}, c_b, \pi_{poly_b}, \pi_{eq})$

Figure G.1.: CP-ColumnSum.Prove

Public statement: (c_A, c_B)
Verifier Input: $\pi = (c_a, \pi_{poly_a}, , \pi_{sc}, c_b, \pi_{poly_b}, \pi_{eq})$ 1. Generate public input • $H(c_A, c_B) \rightarrow J$ 2. **Output** 1 if: • CP-Poly.Verify$(\pi_{poly_b}, J, c_B, c_b) \rightarrow 1$ • CP-Sumcheck'.Verify$(\pi_{sc}, c_b) \rightarrow (1, R, c_l)$ • CP-Poly.Verify$(\pi_{poly_a}, (R, J), c_A, c_a) \rightarrow 1$ • ZK_{eq}.Verify$(c_a, c_l, \pi_{eq}) \rightarrow 1$, else **output** 0

Figure G.2.: CP-ColumnSum.Verify

H. CP-ScalarAdd

Public statement: (c_A, c_C, c_B)
Prover Witness: (A, C, B, o_A, o_C, o_B) ZK_{eq}.Prove$(C, o_C, o_B - o_A, c_C, \frac{c_B}{c_A}) \to \pi$ **output**: π

Figure H.1.: CP-ScalarAdd.Prove

Public statement: (c_A, c_C, c_B)
Verifier Input: π **Output** 1 if: ZK_{eq}.Verify$(c_C, \frac{c_B}{c_A}, \pi) \to 1$, else **output** 0

Figure H.2.: CP-ScalarAdd.Verify

I. CP-Linear

Public statement: $(c_X, c_Y, c_B, c_Z, c_O)$

Prover

Witness: $(X, Y, B, Z, O, o_X, o_Y, o_B, o_Z, o_O)$

1. Calculate the prediction
   - Compute $eHad(Z, W) = H$
   - Compute $columnSum(H) = S$
   - Note that: $scalarAdd(S, B) = O$
2. Commit computation values
   - PolyCom.Commit$(ck, H, p) \rightarrow (c_H, o_H)$
   - PolyCom.Commit$(ck, S, p) \rightarrow (c_S, o_S)$
3. CP-EHad.Prove$(x, w) \rightarrow \pi_{ehad}$
   - $x = (c_Z, c_W, c_H)$
   - $w = (Z, W, H, o_Z, o_W, o_H)$
4. CP-ColumnSum.Prove$(x, w) \rightarrow \pi_{columnSum}$
   - $x = (c_H, c_S)$
   - $w = (H, S, o_H, o_S)$
5. CP-ScalarAdd.Prove$(x, w) \rightarrow \pi_{scalarAdd}$
   - $x = (c_S, c_B, c_O)$
   - $w = (S, B, O, o_S, o_B, o_O)$

output: $\pi = (c_H, c_S, \pi_{eHad}, \pi_{columnSum}, \pi_{scalarAdd})$

Figure I.1.: CP-Linear.Prove

Public statement: (c_W, c_B, c_Z, c_0)
Verifier Input: $\pi = (c_H, c_S, \pi_{eHad}, \pi_{columnSum}, \pi_{scalarAdd})$ **Output** 1 if: • CP-EHad.Verify$(\pi_{eHad}, c_Z, c_W, c_H) \rightarrow 1$ • CP-ColumnSum.Verify$(\pi_{columnSum}, c_H, c_S) \rightarrow 1$ • CP-ScalarAdd.Verify$(\pi_{scalarAdd}, c_S, c_B, c_O) \rightarrow 1$, else **output** 0

Figure I.2.: CP-Linear.Verify

J. CP-SVMLinear

Public statement: $(c_X, c_Y, c_B, c_Z, c_O)$

Prover

Witness: $(X, Y, B, Z, O, o_X, o_Y, o_B, o_Z, o_O)$

1. Calculate the prediction
   - Compute $mm(X, Z) = M$
   - Compute $eHad(M, Y) = H$
   - Compute $columnSum(H) = S$
   - Note that: $scalarAdd(S, B) = O$
2. Commit computation values
   - PolyCom.Commit$(ck, M, p) \rightarrow (c_M, o_M)$
   - PolyCom.Commit$(ck, H, p) \rightarrow (c_H, o_H)$
   - PolyCom.Commit$(ck, S, p) \rightarrow (c_S, o_S)$
3. CP-MM.Prove$(x, w) \rightarrow \pi_{mm}$
   - $x = (c_X, c_Z, c_M)$
   - $w = (X, Z, M, o_X, o_Z, o_M)$
4. CP-EHad.Prove$(x, w) \rightarrow \pi_{ehad}$
   - $x = (c_M, c_Y, c_H)$
   - $w = (M, Y, H, o_M, o_Y, o_H)$
5. CP-ColumnSum.Prove$(x, w) \rightarrow \pi_{columnSum}$
   - $x = (c_H, c_S)$
   - $w = (H, S, o_H, o_S)$

5\. CP-ScalarAdd.Prove$(x, w) \rightarrow \pi_{scalarAdd}$
   - $x = (c_S, c_B, c_O)$
   - $w = (S, B, O, o_S, o_B, o_O)$

output: $\pi = (c_M, c_H, c_S, \pi_{mm}, \pi_{eHad}, \pi_{columnSum}, \pi_{scalarAdd})$

Figure J.1.: CP-SVMLinear.Prove

Public statement: $(c_X, c_Y, c_B, c_Z, c_0)$
Verifier Input: $\pi = (c_M, c_H, c_S, \pi_{mm}, \pi_{eHad}, \pi_{columnSum}, \pi_{scalarAdd})$ **Output** 1 if: • CP-MM.Verify$(\pi_{mm}, c_X, c_Z, c_M) \rightarrow 1$ • CP-EHad.Verify$(\pi_{eHad}, c_M, c_Y, c_H) \rightarrow 1$ • CP-ColumnSum.Verify$(\pi_{columnSum}, c_H, c_S) \rightarrow 1$ • CP-ScalarAdd.Verify$(\pi_{scalarAdd}, c_S, c_B, c_O) \rightarrow 1$, else **output** 0

Figure J.2.: CP-SVMLinear.Verify

K. CP-SVMPoly

Public statement: $(d, c_X, c_C, c_Y, c_B, c_Z, c_O)$

Prover

Witness: $(X, C, Y, B, Z, O, o_X, o_C, o_Y, o_B, o_C, o_Z, o_O)$

1. Calculate the prediction
   - Compute $mm(X, Z) = M$
   - Compute $scalarAdd(M, C) = \Gamma$
   - Compute $expo(\Gamma, d) = E$
   - Compute $eHad(E, Y) = H$
   - Compute $columnSum(H) = S$
   - Note that: $scalarAdd(S, B) = O$
2. Commit computation values
   - PolyCom.Commit$(ck, M, p) \rightarrow (c_M, o_M)$
   - PolyCom.Commit$(ck, \Gamma, p) \rightarrow (c_\Gamma, o_\Gamma)$
   - PolyCom.Commit$(ck, E, p) \rightarrow (c_E, o_E)$
   - PolyCom.Commit$(ck, H, p) \rightarrow (c_H, o_H)$
   - PolyCom.Commit$(ck, S, p) \rightarrow (c_S, o_S)$
3. CP-MM.Prove$(x, w) \rightarrow \pi_{mm}$
   - $x = (c_X, c_Z, c_M)$
   - $w = (X, Z, M, o_X, o_Z, o_M)$
4. CP-ScalarAdd.Prove$(x, w) \rightarrow \pi_{scalarAdd1}$
   - $x = (c_M, c_C, c_\gamma)$
   - $w = (M, C, \Gamma, o_M, o_C, o_\Gamma)$
5. CP-Expo.Prove$(x, w) \rightarrow \pi_{expo}$
   - $x = (d, c_\Gamma, c_E)$
   - $w = (\Gamma, E, o_\Gamma, o_E)$
6. CP-EHad.Prove$(x, w) \rightarrow \pi_{ehad}$
   - $x = (c_E, c_Y, c_H)$
   - $w = (E, Y, H, o_E, o_Y, o_H)$
7. CP-ColumnSum.Prove$(x, w) \rightarrow \pi_{columnSum}$
   - $x = (c_H, c_S)$
   - $w = (H, S, o_H, o_S)$
8. CP-ScalarAdd.Prove$(x, w) \rightarrow \pi_{scalarAdd2}$
   - $x = (c_S, c_B, c_O)$
   - $w = (S, B, O, o_S, o_B, o_O)$

output: $\pi = (c_M, c_\Gamma, c_E, c_H, c_S, \pi_{mm}, \pi_{scalarAdd1}, \pi_{expo}, \pi_{eHad}, \pi_{columnSum}, \pi_{scalarAdd2})$

Figure K.1.: CP-SVMPoly.Prove

Public statement: $(d, c_X, c_C, c_Y, c_B, c_Z, c_O)$

Verifier

Input: $\pi = (c_M, c_\Gamma, c_E, c_H, c_S, \pi_{mm}, \pi_{scalarAdd1}, \pi_{expo}, \pi_{eHad}, \pi_{columnSum}, \pi_{scalarAdd2})$

Output 1 if:

- CP-MM.Verify$(\pi_{mm}, c_X, c_Z, c_M) \to 1$
- CP-ScalarAdd.Verify$(\pi_{scalarAdd1}, c_M, c_C, c_\Gamma) \to 1$
- CP-Expo.Verify$(\pi_{expo}, d, c_\Gamma, c_E) \to 1$
- CP-EHad.Verify$(\pi_{eHad}, c_E, c_Y, c_H) \to 1$
- CP-ColumnSum.Verify$(\pi_{columnSum}, c_H, c_S) \to 1$
- CP-ScalarAdd.Verify$(\pi_{scalarAdd2}, c_S, c_B, c_O) \to 1$

, else **output** 0

Figure K.2.: CP-SVMPoly.Verify

Bibliography

[1] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[2] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[3] Lyn Thomas, Jonathan Crook, and David Edelman. *Credit scoring and its applications*. SIAM, 2017.

[4] Francesco Ricci, Lior Rokach, and Bracha Shapira. Recommender systems: introduction and challenges. *Recommender systems handbook*, pages 1–34, 2015.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.

[8] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[9] https://cloud.google.com/ai-platform.

[10] https://aws.amazon.com/marketplace.

[11] https://azuremarketplace.microsoft.com/en-us/marketplace/.

[12] Chaoyue Niu, Fan Wu, Shaojie Tang, Shuai Ma, and Guihai Chen. Toward verifiable and privacy preserving machine learning prediction. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1703–1721, 2020.

[13] Xixun Yu, Zheng Yan, and Athanasios V Vasilakos. A survey of verifiable computation. *Mobile Networks and Applications*, 22:438–453, 2017.

[14] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[15] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2075–2092, 2019.

[16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[17] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. *Advances in Neural Information Processing Systems*, 30, 2017.

[18] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network based on zk-snarks. *Cryptology ePrint Archive*, 2020.

[19] Lingchen Zhao, Qian Wang, Cong Wang, Qi Li, Chao Shen, and Bo Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2524–2540, 2021.

[20] Shadan Ghaffaripour and Ali Miri. Mutually private verifiable machine learning as-a-service: A distributed approach. In *2021 IEEE World AI IoT Congress (AIIoT)*, pages 0232–0239. IEEE, 2021.

[21] Xiong Li, Jiabei He, Pandi Vijayakumar, Xiaosong Zhang, and Victor Chang. A verifiable privacy-preserving machine learning prediction scheme for edge-enhanced hcpss. *IEEE Transactions on Industrial Informatics*, 18(8):5494–5503, 2021.

[22] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[23] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. A zero-knowledge version of vsql. *Cryptology ePrint Archive*, 2017.

[24] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology—CRYPTO'91: Proceedings*, pages 129–140. Springer, 2001.

[25] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, and Jens Groth. Efficient zero-knowledge proof systems. *Foundations of Security Analysis and Design VIII: FOSAD 2014/2015/2016 Tutorial Lectures 15*, pages 1–31, 2016.

[26] Guy N Rothblum. *Delegating computation reliably: paradigms and constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.

[27] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International journal of web and grid services*, 14(4):352–375, 2018.

[28] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class scrbook. The main font is Palatino.

TU Delft