PRIVACY-PRESERVING XGBOOST INFERENCE WITH HOMOMORPHIC ENCRYPTION

by
ŞEYMA SELCAN MAĞARA

Submitted to the Graduate School of Sabancı University
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
October 2022

ABSTRACT

PRIVACY-PRESERVING XGBOOST INFERENCE WITH HOMOMORPHIC ENCRYPTION

ŞEYMA SELCAN MAĞARA

Computer Science and Engineering M.Sc. THESIS, October 2022

Thesis Supervisor: Prof. Erkay Savaş

Keywords: Privacy-Preserving Machine Learning, Homomorphic Encryption, XGBoost

There is a plethora of applications that incorporate Machine Learning (ML) in a wide variety of fields. Many sectors, such as healthcare and business, provide ML-based prediction services. However, data privacy concerns may prevent using machine learning services at their full potential. Enabling ML algorithms to perform under highly secure encryption schemes would grant the chance to overcome the data privacy issue. Running ML algorithms on homomorphically encrypted data is a promising way to eliminate data privacy issues. Thus, the compatibility of existing machine learning models with homomorphic encryption is crucial for data privacy. We chose to work on combining widely used XGBoost inference algorithm and homomorphic encryption to provide a post-quantum security level for query data owners. This work proposes and implements a privacy-preserving XGBoost inference method that performs operations on encrypted data. The prediction and time performance of the method is evaluated using various XGBoost models. Moreover, security and privacy analysis are provided for both query owners and model owners. In conclusion, a practical and effective homomorphic encryption solution for XGBoost inference is presented.

ÖZET

HOMOMORFIK ŞIFRELEME ILE GIZLILIK KORUMALI XGBOOST TAHMIN ALGORITMASI

ŞEYMA SELCAN MAĞARA

BİLGİSAYAR BİLİMLERİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, EKİM 2022

Tez Danışmanı: Prof. Dr. ERKAY SAVAŞ

Anahtar Kelimeler: Gizlilik Korumalı Makine Öğrenmesi, Homomorfik Şifreleme, XGBoost

Farklı alanlarda Makine Öğrenimini (ML) içeren çok sayıda uygulama kullanılmaktadır. Sağlık ve iş dünyası gibi birçok sektör, makine öğrenimi tabanlı tahmin hizmetleri sunmaktadır. Öte yandan, veri gizliliği endişeleri makine öğrenimi hizmetlerinin tam potansiyeliyle kullanılmasını engelleyebilir. Makine öğrenmesi algoritmalarının yüksek güvenlikli şifreleme şemaları altında çalışmasını sağlamak, veri gizliliği sorununun üstesinden gelinmesine olanak tanıyacaktır. Makine öğrenimi algoritmalarını homomorfik olarak şifrelenmiş veriler üzerinde çalıştırmak, veri gizliliği sorunlarını ortadan kaldırmanın oldukça etkili bir yoludur. Bu nedenle, mevcut makine öğrenimi modellerinin homomorfik şifreleme ile kullanılabilirliği veri gizliliği için çok önemlidir. Sorgu veri sahipleri için kuantum sonrası bir güvenlik seviyesi sağlamak amacıyla yaygın olarak kullanılan XGBoost çıkarım algoritması ile homomorfik şifrelemeyi birleştirme üzerine çalışmayı seçtik. Bu çalışma, şifrelenmiş veriler üzerinde işlem yapabilen, gizliliği koruyan bir XGBoost çıkarım yöntemi önermekte ve gerçeklemektedir. Yöntemin tahmin ve zaman performansı çeşitli XGBoost modelleri kullanılarak değerlendirilmiştir. Ayrıca, hem sorgu sahipleri hem de model sahipleri için güvenlik ve gizlilik analizleri sunulmaktadır. Özet olarak, XGBoost çıkarımı için pratik ve etkili bir homomorfik şifreleme çözümü sunulmuştur.

ACKNOWLEDGEMENTS

First, I'd like to thank my thesis supervisor, Prof. Erkay Savaş, for his invaluable support and continuous patience throughout my master's studies. His guidance and insight help me to improve my knowledge in many fields.

I would like to extend my sincere thanks to the thesis jury members Prof. Albert Levi and Prof. Uğur Sezerman, for their time and valuable feedback.

Moreover, I'd like to thank all my friends from CISEC (FENS 2014), Anes, Ceren, Utku, Fatih, and Berk, for creating such a friendly environment. They've always supported and helped me throughout my master's. It was a blessing to work with all of them.

Special thanks to my best friend Berke for always supporting me and believing in me more than I did.

Most importantly, I am sending special thanks to my parents, Halime and Halil, my brothers, Ahmet and Barbaros, and my great family for their endless support, love, and motivation. Knowing that you'd be there for me whenever I need you is the most incredible privilege in the world.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ART Adversarial Robustness Toolbox 54

BFV Brakerski-Gentry-Vaikunthanathan HE scheme 8

CKKS Cheon-Kim-Kim-Song scheme 8

CNN Convolutional Neural Networks 54, 55

FHE Fully Homomorphic Encryption 7

HE Homomorphic Encryption 2

ML Machine Learning 1, 19

MLaaS Machine Learning as a Service 18

PPML Privacy-Preserving Machine Learning 1

RLWE Ring Learning with Errors 7

SIMD Single Instruction Multiple Data 10, 28, 29, 30, 34

SWHE Somewhat Homomorphic Encryption 7

XGBoost eXtreme Gradient Boosting 1

LIST OF NOTATIONS

Notation	Description	Page List
E_f	Encoding of the query data's feature f	22
E_s	Encoding of the split point s	21
N	The dimension of the ring $\boldsymbol{R}_q$	11, 30, 41
S_f	Set of the split points of feature f	20, 21
Z_j	The ciphertext containing the comparison result of the j^{th} nodes.	33
Δ	Scaling factor of CKKS scheme	10, 11, 40, 41
$\circ$	The symbol stands for homomorphic bitwise multiplication.	10, 32
λ	Security level of CKKS scheme.	12, 40, 41
$\overrightarrow{M}$	Vector representation of the model. Contains bit string corresponding of each node	30
$\overrightarrow{Z}$	Comparison results	33, 34
d_{max}	Maximum depth of the trees in the model	14, 27, 28, 29, 30, 37, 38, 39, 41, 42, 69, 70, 71, 72, 73, 74, 75
m	Total number of Features	31, 42

Notation	Description	Page List
m'	The number of features used in the model	38, 39, 42, 62, 63, 74
n_C	Total number of classes in a dataset	14
n_E	Length of the encoding	29, 30, 31, 33, 42, 62, 63
n_T	Number of gradient boosted trees, a parameter to XGBoost training algorithm.	14, 15, 38, 39, 46, 69, 70, 71, 72, 73, 74, 75
n_f	Number of split points for a particular feature f	21, 22, 23
n_{TT}	Total number of trees in a model	14, 30, 31, 32, 41, 42
p_i	Path with index i	26
q	Ciphertext modulus	11, 41, 43
t_ℓ	learning rate of the XGBoost training algorithm	14, 38, 39, 69, 70, 71, 72, 73, 74
z_i	Ciphertext that contains the comparison result of the node i	25
B	Buckets	21, 22

1. INTRODUCTION

Machine Learning (ML) applications have a substantial role in various areas such as finance (Goodell, Kumar, Lim & Pattnaik (2021); Renault (2020)), healthcare (Callahan & Shah (2017); Pattnayak & Panda (2021)), face recognition, and spam detection. For instance, we can predict stock market prices and the start of a disease within a patient body and craft machine learning models to provide cybersecurity-related services (Dasgupta, Akhtar & Sen (2022); Dua & Du (2016); Handa, Sharma & Shukla (2019)). Moreover, with the recent growth in the adoption of cloud services, machine learning has reached extensive levels of usage in many fields. However, the widespread use of the ML has brought privacy concerns since the data used by the ML applications might be sensitive. Therefore, the increase in privacy concerns raised the need and interest in the field of Privacy-Preserving Machine Learning (PPML).

Particularly in the health sector, data is usually considered highly sensitive. Breaching or disclosing data related to patients of various diseases can put them through different harmful situations, economically (e.g., lose their health insurance or put their jobs at risk), psychologically (e.g., being known as an HIV patient), and socially (e.g., depressions and isolation from society). Most critically, it may lead to identity impersonation or theft. As a result, data owners have serious security concerns about sharing their data openly. This hesitation prevents the machine learning algorithms from being used at their highest capacity. PPML algorithms offer a solution to this predicament.

XGBoost is one of the most prevalent machine learning algorithms. It provides efficient and well-performing models since it can handle various data types and provides fast and accurate predictions (Jayaraman, Forkan, Morshed, Haghighi & Kang (2020); Ma, Ma, Miao, Liu, Choo, Yang & Wang (2020); Sharma & Verbeke (2020)). However, in real-life situations, where highly sensitive data needs to be processed, the data owners may want to keep their data as encrypted so that no one else accesses the data. Thus, the XGBoost algorithm needs to be made suitable to work on encrypted data to benefit from its power in a secure and privacy-preserving manner.

The biggest obstacle for the XGBoost algorithm to work on encrypted data is that the XGBoost inference algorithm requires comparison operation at each node of the model trees. Although the comparison is straightforward on plain data, comparing encrypted data is a costly operation. Therefore, it is not feasible to use the current homomorphic comparison operation methods in XGBoost inference applications. In this thesis, a practical and low-cost encoding method is presented for the required comparison operations. Moreover, the homomorphic XGBoost inference algorithm is implemented using the proposed method and the applicability of this method is demonstrated by experimenting with various data sets. We also investigated the security and privacy implications of using XGBoost on encrypted data in a use case scenario, in which data owners send their encrypted data to the model owner, who homomorphically apply the model on the encrypted data.

The proposed encoding method forms possible data intervals, which are named as buckets, using the decision rules of the XGBoost model. Then, both inputs of the comparison operation (i.e., query data and the split value, where the comparisons happen) are encoded into binary numbers using the buckets so that the digit-by-digit multiplication of the two side's encoded values yields the comparison result. Using encoded inputs makes the comparison operation on encrypted data more concise and requires fewer operations than conventional methods. Hence, the proposed method ensures a feasible and practical XGBoost inference algorithm as a natural consequence of low-cost comparison operations.

The proposed method is implemented using the PALISADE homomorphic encryption library and tested on three different data sets. The method's computation time and prediction performance are evaluated on different XGBoost model setups. Also, the security aspect of the proposed method is analyzed, and the possible attack scenarios are investigated. Regarding the experiments and analysis results, the method gives high accuracy and good timing performance while protecting privacy.

In conclusion, the proposed encoding method enables performing homomorphic operations on a machine learning inference algorithm. Specifically, it proposes a privacy-preserving XGBoost inference method using homomorphic encryption (HE). Thus, a data owner can encrypt his/her data, have it homomorphically processed and get the prediction result without revealing any sensitive data.

The work is organized as follows: Section 2 explores the literature about applying privacy-preserving techniques in machine learning. It also provides the necessary knowledge about homomorphic encryption schemes and their main primitives, the machine learning algorithm used in this study, and details about the inference operations. Next, the methodology is described in Section 3. Then, the experimental

setup and results are given and discussed in Section 4 followed by a security analysis in Section 5. Finally, the concluding remarks of the thesis are presented and potential future work is discussed in Section 6.

2. BACKGROUND AND RELATED WORK

This chapter is organized into three subsections. Section 2.1 analyzes the related works and gives an overview of the current literature. In Section 2.2, the fundamentals of homomorphic encryption are explained in detail, including various HE schemes and HE implementations. Next, Section 2.3 provides an overview of the XGBoost model structure and explains the XGBoost inference algorithm in detail. Finally, Section 2.4 concludes with a discussion on the challenges of adopting the XGBoost algorithm to the HE setting.

2.1 Related Work

With the fast pace of development in artificial intelligence, big data, machine learning, and federated learning technologies, there comes a threat that hinders the evolving process of such technologies, which is violation and loss of data privacy. There is a trade-off between performance of data processing and privacy of data being processed. For instance, the best prediction accuracy in the developed machine learning model may be achieved when security and privacy are not taken into account. Major data leakage threats may occur when the model is shared or used or when sending the data to a third party to apply the model and deliver the results back. These data exchanges cannot be held in public. Otherwise, it would lead to serious privacy issues ((De Cristofaro, 2021; Zhang, Xu, Vijayakumar, Sharma & Ghosh, 2022)).

Many works in the healthcare field use machine learning algorithms to build models for different purposes, such as privacy-preserving genome-related studies, ancestry and paternity testing (Wood, Najarian & Kahrobaei, 2020), classification ML models for DNA sequencing (Yang, Niehaus, Walker, Iqbal, Walker, Wilson, Peto, Crook, Smith, Zhu & others, 2018), etc. The common security concern that these applications address is data privacy. Data has to be shared between parties to apply

the developed models. The core issue is the potential disclosure of the data to an unauthorized or malicious party. Since the effectiveness of the machine learning techniques are promising, various methods were applied to overcome this privacy challenge.

A plethora of machine learning models and protocols, which have been developed under the umbrella of healthcare research, propose several techniques for protection of privacy. For example, Paul, Annamalai, Ming, Al Badawi, Veeravalli & Aung (2021) proposed a learning protocol that encrypts the feature activations and uses a homomorphically encrypted logistic regression model for training. The protocol uses the Cheon-Kim-Kim-Song (CKKS) scheme (Cheon, Kim, Kim & Song (2017a)) since the data comprises fixed-point arithmetic operations. Furthermore, for a low-cost ubiquitous system, Kwabena, Qin, Zhuang & Qin (2019) built a multi-scheme deep learning system called *MSCryptoNet* that guarantees the privacy of the clients' data. This system allows the training of a collaborative deep neural network model of ciphertext encrypted with different homomorphic encryption (HE) schemes or have various keys. Moreover, it offers an acceptable security level.

Some of the current literature that use HE-based deep learning algorithms perform the training process in a federated form, which means data owners participate in the process by training partial models on their private data and then aggregating the global one in the last training stage. HE-based protection techniques are especially useful as the privacy concerns and the potential of data leakage are pronounced in a federated learning scenario, which includes several parties and their private inputs. The works such as (Ku, Susilo, Zhang, Liu & Zhang (2022); Ma, Naas, Sigg & Lyu (2022); Singh, Rathore, Alfarraj, Tolba & Yoon (2022)) are examples of building algorithms for privacy-preserving federated learning for healthcare-related research purposes.

Another way to enhance data privacy in healthcare is by including the blockchain technology. One of the primary roles of blockchain in federated learning models is the decentralization of the local gradient aggregation process, where the specialized (local) models are combined to form the global one by not disclosing the partial models' privacy. Therefore, ensuring better personal data confidentiality and integrity since there is no further need to send the data to a centralized server or a trusted third party to perform this operation (Kasyap & Tripathy (2021); Passerat-Palmbach, Farnan, McCoy, Harris, Manion, Flannery & Gleim (2020); Singh et al. (2022)).

Last but not least, Meng & Feigenbaum (2020) proposed a privacy preserving XG-Boost inference algorithm and implemented it empirically on AWS Sagemaker. They

take advantage of the Order Preserving Encryption (OPE) and hyperparameters of XGBoost as well as homomorphic encryption to provide privacy. They are using XGBoost hyperparameters to train different models for each user. OPE supports the comparison of model and query data in an efficient way. Although they offer a different and practical solution, the method is quite limited. They mentioned that OPE is used to support comparison operations due to the high cost of comparison on the semantically encrypted data. They also mentioned that the comparison part is still open to further investigation.

In this work, we propose an encoding method for query data and the XGBoost model that allows us to implement a comparison function for homomorphic encryption. The proposed method allows for a semantically secure solution unlike the work in Meng & Feigenbaum (2020); its security properties is discussed in Section 5.2

2.2 Homomorphic Encryption

Encryption is an essential cryptographic tool to preserve the privacy of data owners. However, conventional encryption methods do not allow working directly on encrypted data (Acar, Aksu, Uluagac & Conti (2018)). This limitation forces data owners to either unveil their data or refrain from processing their data lest it is exposed to unauthorized parties. Homomorphic Encryption (HE) helps out with this predicament since it is a cryptographic technique that makes it possible to perform mathematical computations directly on encrypted data without decryption.

There are some cryptographic algorithms that are inherently homomorphic over a particular operation. However, the term "homomorphism" was first used deliberately in 1978 by Rivest & Dertouzos (1978). Since then, the technique has been evolving. Today we define homomorphic encryption as in Definition 2.2.1.

Definition 2.2.1. An encryption technique E is called homomorphic if it satisfies the following equation for any possible input m:

$$D(f(E(m))) = f(m), \tag{2.1}$$

where D is the decryption function and f is an arbitrary primitive operation (e.g., multiplication, addition).

There are different types of HE schemes as illustrated in Figure 2.1. Partially ho-

momorphic encryption schemes support only one homomorphic operation, which are usually either addition or multiplication, but not both. For example, a very well-known partially homomorphic encryption scheme is Paillier public key cryptographic algorithm, which supports only addition (Paillier, 1999). Therefore, their applicability is very limited as oftentimes we need both operations even for basic data processing.

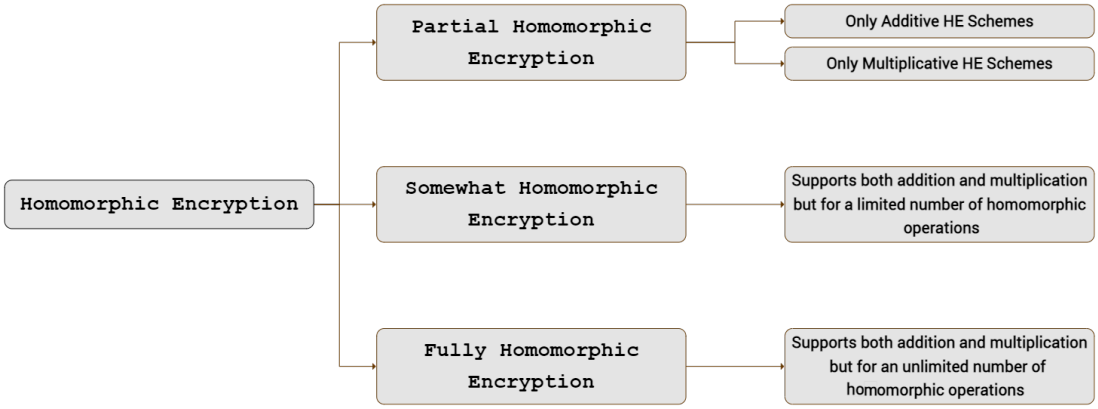


Figure 2.1 Homomorphic Encryption types and divisions

Fully homomorphic encryption scheme that supports both addition and multiplication had remained one of the most important unsolved problems in cryptography until the time Gentry (2009) published his seminal work. Gentry introduced a technique called "bootstrapping" to perform as many homomorphic operation over encrypted data as needed. Bootstrapping is needed to decrease the the so-called "error" or "noise" in the ciphertext, which can render it undecipherable if it exceeds certain threshold as a result of homomorphic operations. As bootstrapping is an expensive operation an alternative version known as "somewhat homomorphic encryption" (SWHE) is proposed, where the depth of the circuit, over which homomorphic operations are evaluated, is limited. SWHE schemes, on the other hand, are very popular as they are efficient for practical applications. From now on, we will use the short form HE to refer to mostly SWHE schemes, but also for FHE as SWHE schemes used in this thesis also support bootstrapping. Nonetheless, bootstrapping operation will not be used here for efficiency reasons.

As all commonly-used FHE and SWHE schemes are lattice-based they are also quantum secure. In particular, the security of the most practicable HE schemes based on the computational problem called "Ring Learning with Errors" (RLWE) (Lyubashevsky, Peikert & Regev, 2013). RLWE can be reduced to most difficult problems over lattices that are considered secure against quantum computers (Regev, 2005), (Peikert, 2009)

The RLWE hardness assumption is defined upon the following parameters: we have a set of pairs of $(a_i; a_i \times s + e_i)$ such that a_i and $s \in \boldsymbol{R}_q$ where $\boldsymbol{R}_q$ represents the polynomial rings, namely $\boldsymbol{R}_q = \mathbb{Z}_r/\Phi(x)$. Here, $\Phi(x)$ is the degree N cyclotomic polynomial. The main operation in $\boldsymbol{R}_q$ is polynomial arithmetic, where the reduction is performed by the cyclotomic polynomial. Therefore, all the polynomials are of degree $N-1$ or less. The arithmetic operations over the polynomial coefficients are performed modulo q. Also, $e_i \in \boldsymbol{D}_{\boldsymbol{R},\sigma}$ is random polynomial $\boldsymbol{R}$ drawn from a Gaussian distribution with 0 mean and relatively small standard deviation, σ.

Given these polynomials we have; $b_i = a_i \times s + e_i$ and we define the 'Search' and 'Decision' versions of the hard problems, on which the security of HE schemes are based:

1.1 Search RLWE: given pairs of a_i and b_i it is hard to extract the secret s.

1.2 Decision RLWE: given pairs of a_i and b_i it is hard to decide whether $b_i = a_i \times s + e_i$ or b_i is randomly drawn from R_q.

In HE schemes used in this thesis, s is secret key whereas the $(b_i,\ a_i)$ are public keys.

2.2.1 SWHE Schemes: BFV and CKKS

There are various HE schemes with different features, such as BFV (Brakerski, Gentry & Vaikuntanathan, 2014) and CKKS (Cheon, Kim, Kim & Song, 2017b). Brakerski/Fan-Vercauteren (BFV) encryption scheme is one of the second-generation HE schemes based on the RLWE hardness assumption. The scheme is adapted for homomorphic operations on integers. Since real-world applications, such as machine learning algorithms usually work with non-integer values, this scheme has a limited field of applications.

On the other hand, the Cheon-Kim-Kim-Song (CKKS) scheme is suitable for homomorphic operations over fixed-point numbers. This scheme is first introduced in (Cheon et al. (2017a)) as an approximate arithmetic operation technique for homomorphic encryption. Since CKKS's working domain is more extensive and includes rational numbers, it is more suitable for general use. The details of the CKKS scheme is given in Section 2.2.2.

2.2.2 CKKS Homomorphic Encryption Scheme

This section covers the main primitives of the CKKS scheme, including CKKS encoding, encryption, and batching. An overview of the CKKS scheme is given in Figure 2.2. The CKKS algorithm works over the complex numbers, $\mathbb{C}$. The input fed to the scheme is of float data type. Then, the scheme converts the input to vector of complex numbers of dimension $N/2$, where N is the dimension of the ring $\boldsymbol{R}_q$. Thus, for the message we can write $msg \in \mathbb{C}$, which is, then, encoded into a plaintext polynomial $p(X) \in \boldsymbol{R}$. Later, the plaintext polynomial $p(x)$ is encrypted, resulting in a pair of polynomials as ciphertext in $\boldsymbol{R}_q$. For encryption, both secret and public keys can be used. However, this work assumes that the encryption is performed using the public key.

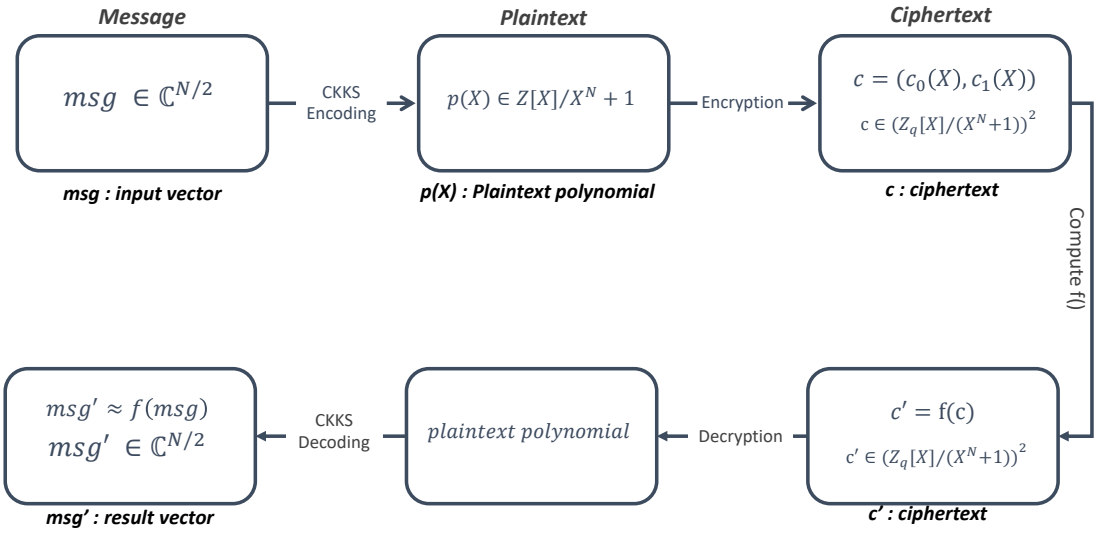


Figure 2.2 High level view of the CKKS scheme

The CKKS scheme uses the error e (or noise) introduced by the RLWE-based HE schemes for security purposes. Hence, the message is combined with some small random error during the encryption operation. Consequently, the decryption of ciphertext c in Figure 2.2 is not exactly msg but an approximation to msg. Similarly, the result of function $f(c)$, is decrypted to $f(msg) + e'$ for a small error e'.

Due to the error introduced by the encryption, the error increases every time an operation is performed on ciphertexts. The homomorphic addition operation is low-cost since the error increases linearly per addition. On the other hand, ciphertext multiplication operations increase the error quadratically. Therefore, only a limited

number of multiplication operations can be performed without completely distorting the message.

Messages of small magnitude are prone to be corrupted by the added error in the CKKS scheme, which addresses this issue by scaling up the messages with the **scaling factor** Δ. After the operations, the result is scaled back. The scaling factor is also accepted as the indicator of precision in results.

2.2.2.1 Batching in Homomorphic Encryption Schemes

HE schemes, including CKKS and BFV, support the encryption of a vector of plaintexts into a single ciphertext. Here, the term "*slot*" is used to designate the virtual ciphertext location, into which each component of the plaintext vector is encoded and encrypted. The same operation is independently executed over all components when a homomorphic operation is performed on the ciphertext. As a result, homomorphic operations in these schemes can be classified as Single Instruction Multiple Data (SIMD) type operations. This feature is also called *batching* in the context of HE. Batching is a powerful data encoding technique to improve the performance of HE applications.

The number of elements in the batching depends on the ring dimension. If the ring dimension is N, then the CKKS algorithm converts a vector of up to $N/2$ elements into a ciphertext (or a plaintext). For instance, in Figure 2.2 the input vector msg is encrypted into a single ciphertext c. If $msg = [msg_0, msg_1, ..., msg_{N/2-1}]$, then the result vector will be $msg' \approx [f(msg_0), f(msg_1), ..., f(msg_{N/2-1})]$.

In this thesis, all homomorphic operations are batch operations, unless otherwise stated. In other words, multiplication and addition operations on ciphertexts correspond to vector multiplication and vector additions on raw data. Additionally, the $\circ$ symbol represents the batch multiplication operation in this work.

Also, when the algorithms are presented in the rest of this thesis, ciphertexts are represented as if they are plaintext vectors, and homomorphic operations are shown to be executed over those vectors without showing their encryption for the sake of simplicity.

2.2.3 Homomorphic Encryption Implementations

The Homomorphic Encryption Schemes are implemented with various software libraries. PALISADE (Polyakov, Rohloff & Ryan, 2017), SEAL (SEAL, 2022), and HElib (Halevi & Shoup, 2014) are some of the well-known and widely used open-source HE libraries. Although this work uses both SEAL and PALISADE for implementation and performance testing, the final version is implemented and tested on PALISADE Release v1.11.7 due to the convenience of use.

In the following, we introduce the terminology used throughout the thesis in the context of homomorphic encryption.

- *Homomorphic evaluation of a circuit*: For homomorphic computation, the operation over the ciphertext is expressed as an (arithmetic or logic) circuit whose inputs[1], outputs, and wires are all encrypted. Given encrypted input values, the computation of encrypted values of wires and outputs is referred to as the (homomorphic) evaluation of the circuit.

- *Multiplication Depth*: It represents the number of multiplications in the circuit that are sequentially executed. Multiplication is much costlier than addition in terms of noise. In other words, each multiplication operation multiplies the noise in the ciphertext much more than the addition operation. Therefore, a limited number of multiplications can be performed before the ciphertext noise gets too high for decryption.

- *Ciphertext modulus* (q): The parameter, which is the modulus used for the coefficients of polynomials in $\boldsymbol{R}_q$, determines the noise amount that can be tolerated. In other words, q is the most important factor determining the number of homomorphic operations that can be applied to ciphertext. In SWHE, this parameter is set depending on the circuit's multiplication depth to be homomorphically evaluated.

- *Ring dimension* (N): It is the dimension of the ring, $\boldsymbol{R}_q$, which is determined by the security level and the ciphertext modulus. N determines the size of plaintext vector in batching, designated by $batchSize$; in particular, $batchSize = N/2$ in CKKS.

- *Scaling factor* (Δ): This parameter determines the precision of the fixed-point numbers used in the CKKS scheme. The CKKS scheme scales up the in-

[1]Some of the inputs to the homomorphic encryption can be plaintext when they are not private. Here, we use the term input to refer to those that are encrypted.

put value by the scaling factor to minimize the effect of error growth on the input value. The number of homomorphic operations, especially the homomorphic multiplication, is proportional to the error accumulated in the ciphertext. Therefore, higher scaling factors are more suitable for circuits of high multiplication depth. The current version of the PALISADE library supports scaling factors up to 60-bit in fixed point representation.

- *Security Level* (λ): Determines the computational effort to break a cryptographic algorithm. The security level is usually given as the base-two logarithm of the so-called computational effort. 128, 192, and 256-bit security level options are used in practice, whereas 100-bit or higher security levels are considered secure.

Existing homomorphic encryption schemes such as BFV and CKKS allow performing several arithmetic operations on encrypted data, such as multiplication, addition, and subtraction, as well as logic operations such as rotation of the plaintext vector in the slots of a ciphertext. The HE libraries implement these fundamental HE operations. Additionally, more complex operations, such as vector rotation and batch summation, are implemented. The batch summation computes the sum of $batchSize$ elements in the vector. Initially, it replicates the ciphertext, rotates it to the left by $batchSize/2$ slots, then adds it to the original ciphertext homomorphically. Then, the operation is repeated on the resulting ciphertext, and the rotation amount is halved until it reaches the value 1. If the $batchSize$ is a power of 2, the batch summation operation is repeated $\log_2 batchSize$ times. Otherwise, $batchSize$ is decomposed in to powers of 2, then batch summation operation is repeated for each power. For instance, if the $batchSize$ is 24, the number of rotations needed is $\log_2 16 + \log_2 8 = 7$.

Currently, the HE libraries do not provide a comparison function, as there is no universal and practical solution to comparison operation yet (See Section 2.4). Some of the homomorphic operations used in this work are summarized in Table 2.1. These operations are given by the names defined in the PALISADE library.

Table 2.1 Some operations supported by PALISADE library

Operation	Description
EvalAdd	Adds two ciphertexts homomorphically
EvalMult	Multiplies two ciphertexts homomorphically
EvalSum	Computes sum of n elements in the encrypted vector; works better if n is a power of 2
EvalMultMany	Multiplies k ciphertexts using binary-tree multiplication method Total multiplication cost is $\log_2(k)$ instead of k

2.3 XGBoost

eXtreme Gradient Boosting (XGBoost) is one of the decision-tree ensemble machine learning models. It uses a gradient boosting framework to train its data. The "eXtreme" part comes from some algorithmic improvements and optimizations applied differently from the other gradient boosting ML methods, which are explained in detail in Chen & Guestrin (2016). The training algorithm of the XGBoost generates an ensemble of classification and regression trees (CART) as an ML model. CARTs are similar to the decision trees apart from the leaf nodes. In CART, each leaf node is associated with a score, which can be interpreted based on the application and depend on hyperparameters (Meng & Feigenbaum (2020)). The internal, non-leaf nodes contain decision rules which are simply comparison operations. It is checked whether a feature value of the query is less than a particular value. The values used for comparison in decision rules are called "split points". An example of an XGBoost tree ensemble is given in Figure 2.3, in which, as it can be seen, an internal node contains a decision rule, which is simply a less than operation. For instance, the root node of the first tree (i.e., Tree 0) compares the value of the feature f_1 of the query data with the integer 148. In this case, 148 is a split point for the feature f_1. Its left branch is selected if the query data satisfies the decision rule. Otherwise, it selects the right branch.

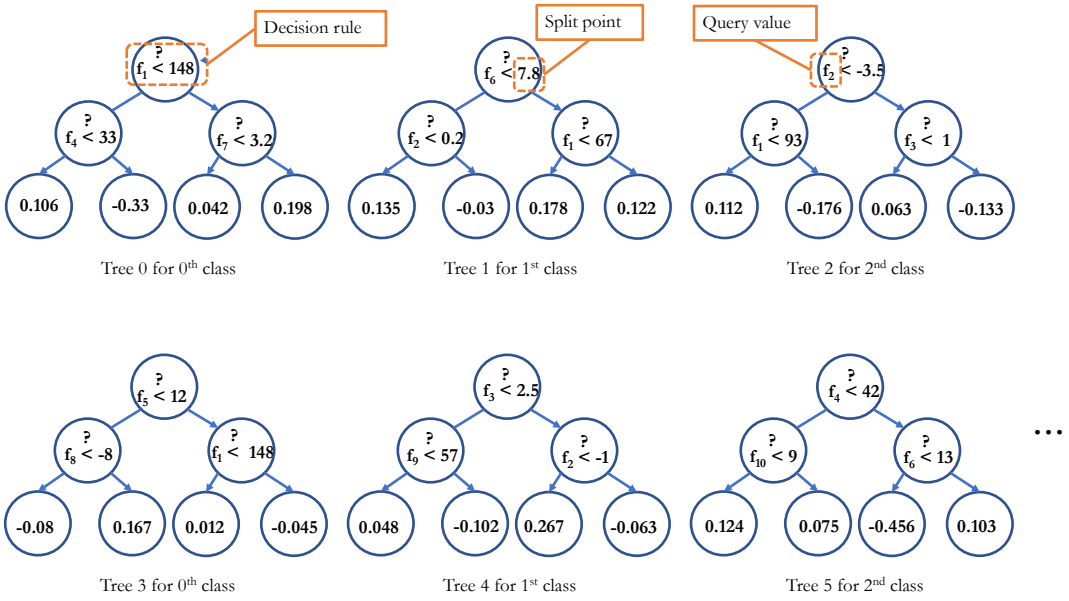


Figure 2.3 A section from an XGBoost tree ensemble for 3-class classification

2.3.1 Parameters

XGBoost algorithm is implemented and available as an open-source software library and provides a framework for different programming languages. One should set many parameters to use the XGBoost algorithm and generate a model. There are three types of parameters: general parameters, booster parameters, and task parameters. Most of these are training-related parameters that are not in the scope of this work. However, some of those parameters affect both the inference algorithm and the proposed encoding method. Thus the parameters needed in the inference part are summarized in this section.

- n_T: Number of estimators or number of gradient boosted trees.
- d_{max}: Maximum depth of the trees. The algorithm grows trees up to this depth. However, the trees may have less depth if early cutting or pruning is applied.
- n_C: The number of classes. It is not a parameter to the XGBoost object but an attribute of the data set. However, this affects the total number of trees in the model. If the classification is binary, the model consists of n_T trees. If it is a multi-class classification, the algorithm generates $(n_C \times n_T)$ trees.
- t_ℓ: The learning rate. A parameter used in boosting determines the learning speed of the model. The value must be between 0 and 1. The default is 0.3. We used this parameter to generate models with different levels of complexity.

The total number of trees in a model represented with n_{TT} is equal to n_T in the case of a binary class dataset. Otherwise, $n_{TT}= (n_C \times n_T)$ trees, in total.

2.3.2 Inference of XGBoost

For inference, the query data is evaluated on the XGBoost model's trees, and each tree's score is obtained. The evaluation starts from the root node. The query data is checked for decision rules. If the evaluation of a decision rule is true, the left branch is followed; otherwise, it selects the right one. This evaluation process continues until a leaf node is reached, which contains a score. The score of the tree corresponds to the score of the leaf node reached during the evaluation. Based on the classification type, the trees' scores are summed, then the results are interpreted accordingly.

The XGBoost algorithm takes the number of the gradient boosted trees, n_T, as a parameter. The training algorithm generates a single ensemble of size n_T if the model is formed exclusively for binary classification. The query input is evaluated on the model, and the final score is the summation of the trees' scores. If the final score exceeds 0, the query input is classified into the first class. Otherwise, it is classified as the other class.

On the other hand, the training algorithm generates an ensemble of n_T trees for each class if the model is a multi-class classifier. Thus, each ensemble consists of n_T trees. The official XGBoost software library outputs these trees of the model in modular order. In other words, the score of the 1^{st} tree in the model is added to the score of the 1^{st} class, and the score of the 2^{nd} tree is summed to the 2^{nd} class, and so forth. The names of the classes (i.e., 1st class, 2nd class, ...) are based on the training dataset labels. The query data is evaluated on each ensemble. The summation of the tree scores of an ensemble forms the score of the corresponding class. The query data is classified into the class with the highest score.

2.4 Challenges

The inference stage of the XGBoost algorithm requires two fundamental operations: comparison and addition. The features of the query data are evaluated in the nodes by simply comparing them with the split points as stipulated by the model. The addition operation is needed to compute the sum of the scores of the trees. Thus, both comparison and addition should be supported by the homomorphic encryption scheme used to implement XGBoost inference over encrypted data. However, the comparison operation on encrypted numbers is costly in terms of multiplication depth, which is the bottleneck of the HE. Two important methods are proposed for comparing encrypted fixed-point data. One of them uses the sign function $sign()$ and its polynomial approximation (Lee, Lee, No & Kim (2020), Cheon, Kim & Kim (2019)). The other method iteratively computes the comparison result using the approximation $comp(a, b) = \frac{a^k}{a^k+b^k}$ (Cheon, Kim, Kim, Lee & Lee (2019)). However, the accuracy of the method depends on the iteration count. As high-precision results require many iterations, which cause a high level of noise in the ciphertext and longer execution time, this approach is unsuitable for homomorphic XGBoost inference.

In conclusion, the homomorphic comparison of fixed-point numbers is still an open

problem since the proposed solutions are not efficient enough to use in practice (Babenko, Tchernykh, Golimblevskaia, Pulido-Gaytan & Avetisyan (2020)). We propose an efficient query and model encoding method suitable for the comparison operation in the XGBoost inference algorithm. The proposed method allows comparison operations to be performed with multiplication operations.

3. METHODOLOGY

Most mainstream machine learning algorithms, such as neural networks or tree-based learning models, require comparison operation in at least one of the training and inference stages. The inference stage on tree-based models depends mainly on comparison operations. Therefore, performing this operation efficiently on encrypted data is crucial for adapting machine learning models to homomorphic encryption. However, performing comparison operations on encrypted data is expensive and one of the biggest obstacles to using machine learning models in the homomorphic encryption setting, where there are two participants; data owner and model owner. In a typical scenario, the data owner encrypts its private data using its key and sends the resulting ciphertext to the model owner for classification (inference).

The impracticality of the homomorphic comparison operation is due to the prohibitively high computation overhead imposed by extra multiplication operations and pre-calculated values required for each comparison. Although there are some proposals to overcome this challenging operation at a relatively low cost, such as (Cheon et al. (2019), Cheon et al. (2019), Lee et al. (2020)), they are infeasible for other applications. To overcome this problem, we propose a new encoding method for inference/classification based on the XGBoost machine learning algorithm that enables comparison operation using relatively few multiplication and addition operations and with low multiplicative depth. The encoding takes advantage of the fact that the comparisons are merely "less than" operations at the XGBoost tree nodes.

Simply put, the encoding method converts both parties' inputs to the comparison operation, namely query data and decision rule in the machine learning model, into binary numbers. When these two binary numbers are multiplied bit-by-bit, the result would be a binary number representing the comparison result. With this method, comparison operations on encrypted data can be performed in a SIMD fashion, and the multiplication depth of the computation is just one. The proposed encoding method is explained in more detail in Section 3.2.

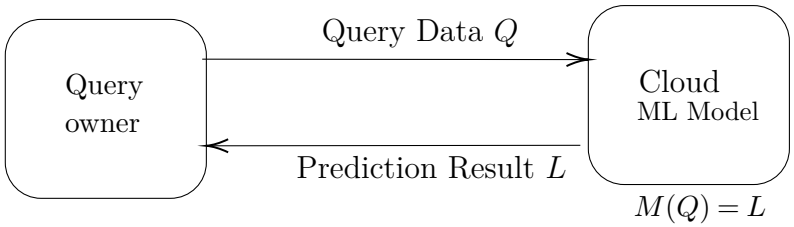
Furthermore, traversing a decision tree is another significant challenge in implement-

ing the homomorphic XGBoost inference operation. During the query evaluation on an XGBoost tree, the result of a comparison operation at a node determines the branch needs to be taken. Nevertheless, the results are also encrypted due to the homomorphic execution. However, this is desired as the selected branch must remain confidential and cannot be revealed to the server to preserve the query owner's privacy. Thus, our method calculates the scores of all possible paths from the root node to leaves in a privacy-preserving manner instead of computing only the correct path. The proposed tree score calculation method, which works with the presented encoding method, is explained in Section 3.3.

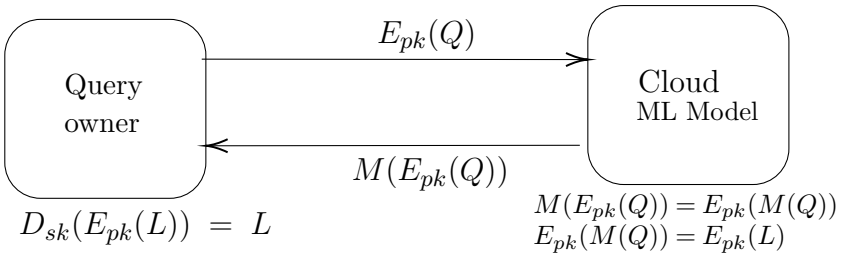
Finally, the proposed methods are combined to implement an XGBoost inference application in a client-server model scenario as visualised in Figure 3.2. In the scenario, the server keeps an XGBoost model; the client first encodes its input data according to the method explained in Section 3.2 and then homomorphically encrypts the encoded input data using the CKKS homomorphic encryption scheme. Later, the client queries the server with the resulting ciphertext. The server calculates the result homomorphically and sends back the ciphertext containing the result to the client. Since the query data is encrypted throughout this process, the client's data privacy is ensured. The implementation details of the algorithm are given in Section 3.4.

3.1 The System Architecture

Numerous cloud-based machine learning solutions emerged with the rapid growth of machine learning in different business areas. Machine Learning as a Service (MLaaS) is a blanket term that covers the wide range of machine learning tools offered by cloud computing services. The proffered MLaaS tools include data visualization, pre-processing, model training, prediction, and more. The prediction services host the trained ML model in the cloud and allow users to request predictions for their data. The users send their data to the cloud service, and the cloud infers the query data on the ML model and sends back the prediction results to the users. Figure 3.1.a demonstrates a machine learning prediction service where $M()$ represents inference operations and L is the prediction result. Although these prediction services were initially designed for plaintext data, they also support privacy-preserving implementations.



a) Basic Machine Learning Prediction Service



b)Privacy-Preserving Machine Learning Prediction Service

Figure 3.1 Demonstration of Machine Learning Prediction Services where $M()$, E_{pk} and D_{sk} represents the inference algorithm, public key encryption and secret key decryption respectively.

Homomorphic encryption is a promising method for privacy-preserving prediction services that aim to ensure the privacy of the query owner. The query owner encrypts the query data using his/her own HE public key and sends the resulting ciphertext to the ML cloud. Since the data is encrypted homomorphically, the ciphertext can be used directly in calculations without decryption. Also, compromising the query data is futile since the ciphertext can be decrypted only using the data owner's secret key. Hence, the query ciphertext is inferred on the ML model without revealing any information. The ciphertext containing the prediction result is sent back to the query owner for decryption.

As detailed in Section 2.4, the evaluation of query ciphertext, $E_{pk}(Q)$, on an XGBoost model is not practical with the current methods. Therefore the scenario shown in Figure 3.1.b can not be implemented efficiently. In order to address the issue of practicality while ensuring privacy preservation, this work investigates an alternative solution.

This thesis proposes an encoding method for the XGBoost predictions that makes it practical to infer encrypted data. The encoding step takes place before the encryption. As shown in Figure 3.2, the server (or cloud) sends the encoding rules to the client. Then, the client encodes the query data into binary sequences by applying encoding rules. The rest of the procedure is the same: the client encrypts the encoded values using his/her HE public key and sends the ciphertext to the model.

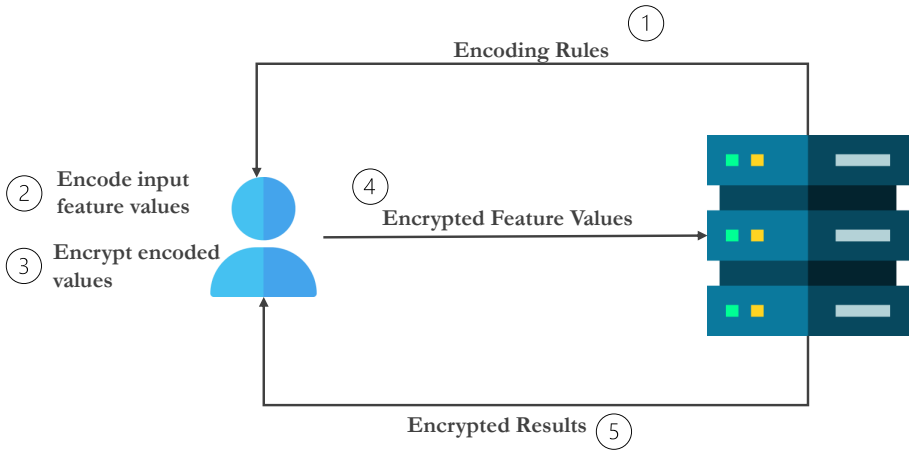


Figure 3.2 Client-Server model and the flow of the overall system

The latter evaluates the encrypted query and sends back the ciphertext the result.

The proposed encoding method provides an efficient solution while preserving the client's data privacy. The details of the encoding method and its application are given in the following subsections. Additionally, Section 5 inspects the overall security of the proposed method.

3.2 Proposed Encoding Method

As the encoding method proposes a solution only for the inference algorithm, we assumed a properly trained XGBoost model is already given in this section. The model consists of binary trees, and each non-leaf (internal) node in the trees contains a decision rule. A decision rule in an XGBoost model is simply a comparison operation. The structure of the decision rules and split points (i.e., values the data features are compared with) used in those rules are explained in Section 2.3. The optimal splitting points and features are determined for each node during the training stage, which may result in many different split point values for a feature. The proposed method considers the multitude of split points to encode the model and the query data.

First, all split points used in the decision rules for each feature are extracted from the model. As described in Definition 3.2.1 all existing split points for a feature f forms the set S_f, where split points s_i are its elements. Sorting the splitting points

of the set in ascending order gives us a clear insight into the intervals, where data points can be contained. These intervals are called buckets in Definition 3.2.1.

Definition 3.2.1. Let $S_f = \{ s_0, s_1, ..., s_{n_F-1} \}$ be the set of split points for the feature f, where $s_i's$ are the split points values such that $s_i < s_{i+1}$ and n_f is the number of split points for the feature f. Then, the buckets to be used for the encoding are defined as

$$\underbrace{(min, s_0)}_{B_0}, \underbrace{[s_0, s_1)}_{B_1}, \underbrace{[s_2, s_3)}_{B_2}, ... \underbrace{[s_{n_f-2}, s_{n_f-1})}_{B_{n_f-1}}. \tag{3.1}$$

In this work, the buckets are defined as the intervals, whose boundaries are determined by (split points of features of data points of) the XGBoost Model, and B is the set of all buckets for the feature f; namely $B = \{B_0, B_1, \ldots, B_{n_F-1}\}$.

The query data and the model's comparison values, or split points, are encoded with the help of these buckets. The proposed method ensures that both the query data and the split point values are encoded as n_f-bit binary numbers, where n_f is the size of the corresponding split value set. However, the encoding methods of query data and split points are slightly different. The details are explained in the following subsections.

3.2.1 Encoding of the Model

This section explains the encoding of the split points in the decision rules of the XGBoost intermediate nodes. If there are n_f different split points for a particular feature in the trees, then the values to be compared can be encoded as n_f-bit binary numbers. The encoding method of the model, or more precisely split points in the model, is given in Algorithm 1.

Briefly speaking, the algorithm first creates a binary vector of all 0's of length n_f, E_s, for the encoding of the split value s, where n_f is the total number of buckets for the feature f. Each element of the vector corresponds to a bucket. The buckets or intervals that are smaller than the value of s are indicated with $1's$ in the vector. A sample encoding is demonstrated in Example 3.2.1.

Example 3.2.1. The heart model we trained using the *Cleveland Heart Disease* (Dua & Graff (2017)) data-set contains a feature named *thalac*. In the model, there are five different comparison values (i.e., split points) used in evaluation of the *thalac*

Algorithm 1 Decision Rule Encoding

```
1: procedure EncodeModel(s, B)  ▷ Encode the split point s using Buckets B
2:     E_s ←< 0,0,...,0 >                                   ▷ |E_s| = n_F
3:     for each bucket B_i in B do
4:         if s ≥ s_i then                                  ▷ B_i = [s_i, s_{i+1})
5:             E_s[i] ← 1
6:         end if
7:     end for
8:     return E_s                                           ▷ The encoding of value s
9: end procedure
```

feature. This means that there are five different intervals/buckets a *thalac* value can be in, and these buckets are determined as follows by a model we trained:

$$(min, 143), [143, 148), [148, 157), [157, 160), [160, 169).$$

Let us encode the split point 148 using these buckets. 148 is greater than the values in the first and second buckets. Therefore, the corresponding digits will be 1. The other digits are 0. So the encoding of the number 148 is <1, 1, 0, 0, 0> Similarly, the encoding of 157 is <1, 1, 1, 0, 0>.

3.2.2 Encoding of the Query Data

The query data is encoded using the same buckets in B obtained during the model encoding. Unlike the model encoding, a method similar to one-hot encoding is applied in the encoding of query data. The query value is represented with an n_f-bit binary number, where each bit corresponds to a bucket as defined in the Definition 3.2.1. This time, only the interval, bucket, containing the query data value is indicated with a high bit and the rest of bits are set to 0. The steps of the method used for query data encoding are given in Algorithm 2. The algorithm gets the feature value f to be encoded and the bucket set B and outputs the feature $f's$ encoding E_f as a binary vector. The Hamming weight of E_f is 1 and the position of the set bit indicates the bucket that contains the value of f.

An illustration of query data encoding is given in Example 3.2.2. The same feature *thalac* and its buckets defined in the previous example 3.2.1 are used for this example.

Example 3.2.2. Some of the *thalac* values in a query data (e.g. 150, 180, 157) are

Algorithm 2 Query Data Encoding

```
1: procedure EncodeQueryData(f, B)   ▷ Encode a feature value of a query data s using Buckets B
2:     E_f ←< 0, 0, ..., 0 >   ▷ where |E_f| = |B|
3:     for each bucket B_i in B do
4:         if f ∈ B_i then
5:             E_f[i] ← 1
6:             break
7:         end if
8:     end for
9:     return E_f   ▷ The encoding of feature value f
10: end procedure
```

encoded using the buckets in Example 3.2.1:

$$B = \{(min, 143), [143, 148), [148, 157), [157, 160), [160, 169)\}$$

150: <0, 0, 1, 0, 0> ▷ 150 falls into the 3rd bucket

180: <0, 0, 0, 0, 0> ▷ All 0, Since it doesn't fall into any buckets

157: <0, 0, 0, 1, 0> ▷ 157 falls into the 4^{th} bucket

3.2.3 Comparison Operation on Encoded Values

With this method, the model and query data are encoded as n_f-bit binary values as pointed out earlier. In the model encoding, set bits in the model corresponds to an entire range of intervals that are smaller than the split point. On the other hand, the set bit of the encoding of the query data represents the interval that includes it. Now, for the comparison of these two binary numbers simple bit-wise multiplication is performed on them. The resulting binary number contains one set bit if the query data is smaller than the split point; otherwise, all bits are zero. The reason is that the set bits of a split point's encoding represent the whole range of intervals smaller than the split point itself. Therefore, the proposed encoding ensures that if the query value is smaller than the split point, its single set bit overlaps with one of the set bits in the split point's encoding. Finally, all bits are summed and the result is accumulated to the bit at index 0. Thus, it will suffice to check the bit in the leftmost position to find out the result of the comparison. If it is 1, then the "less than" comparison holds; otherwise the data point is not smaller than the split point.

Here, we describe the comparison operation as the bit-wise multiplication of encoded values of query data and model, which are simply bit vectors. Although we do not explicitly state it, the vector of query data is homomorphically encrypted and each bit in the vector is placed in a separate slot of the underlying CKKS homomorphic encryption scheme. Also, the result is also encrypted that can only be decrypted by the data owner that sends the encrypted query.

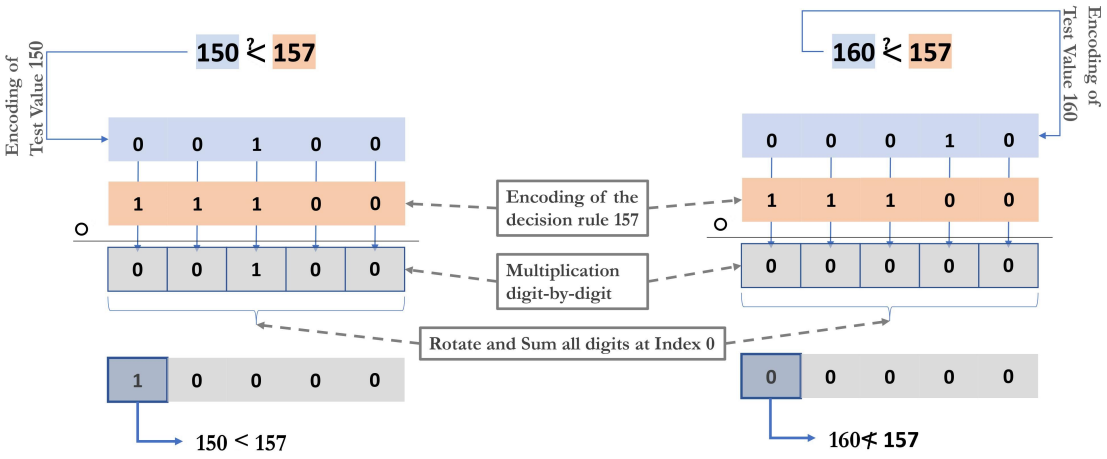


Figure 3.3 Demonstration of comparison operation using encoded values

Example 3.2.3. Continuing our running example (see Examples 3.2.1 and 3.2.2), Figure 3.3 illustrates the comparison operation of encoded values over two instances of data points. Two different query values, 150 and 160, are compared with the split value 157. In the encoding, the buckets defined in the Example 3.2.1 are used. First, the vectors to be compared are multiplied bit-by-bit. Then, the elements of the resulting vector are summed and stored in the position with the index 0 (i.e., the leftmost bit in the encoding) using rotation and addition operations. The result of the first example is <1, 0, 0, 0, 0>, which means the query value is smaller than the split point since the leftmost bit is set. However in the second example on the right hand side, the leftmost bit of the resulting vector is 0 since 160 is not smaller than 157.

This simple, yet powerful method ensures the ability to make comparisons on encrypted data with the cost of a single multiplication. However, in order to apply it to the whole model, the method needs adjustment to accommodate all split points of all features in the data set. First of all, different features may have different numbers of split points. Another point is that the trees of the model are not always complete. Therefore, having a regular structure for the XGBoost trees and the same number

of split points for all features are essential to formulate the inference calculations in homomorphic setting. All the modifications applied are explained in detail in the following subsections.

3.3 Tree Score Calculation

Although the comparison operation can be performed successfully, the resulting value is still encrypted; therefore, it is impossible to tell the correct path to follow on a tree. Nevertheless, it is still possible to obtain the correct leaf score in the homomorphic evaluation of an XGBoost tree as explained in the following.

Recall that the result is 1 if less than condition is true; 0, otherwise. Similarly, if the condition holds, the left branch is taken; otherwise, the right branch. To better illustrate the homomorphic evaluation of an XGboost tree, assume that we are given a tree of depth 3 as showed in Figure 3.4. In the figure, let the values of z_i represent the comparison result performed at tree nodes. For example, $z_0 \in \{0,1\}$ represents the result of the comparison operation at the root node in the tree[1]. If, for instance, the condition at the root node holds, then $z_0 = 1$; otherwise, $1 - z_0 = 1$. This way, the information, whether the branches are taken or not, can be expressed in terms of the node values, z_i. As shown in Figure 3.4, for any intermediate node, the value of the left branch should be identical to the node value z_i, and the value of the right branch should be the complement of the node value, $(1 - z_i)$.

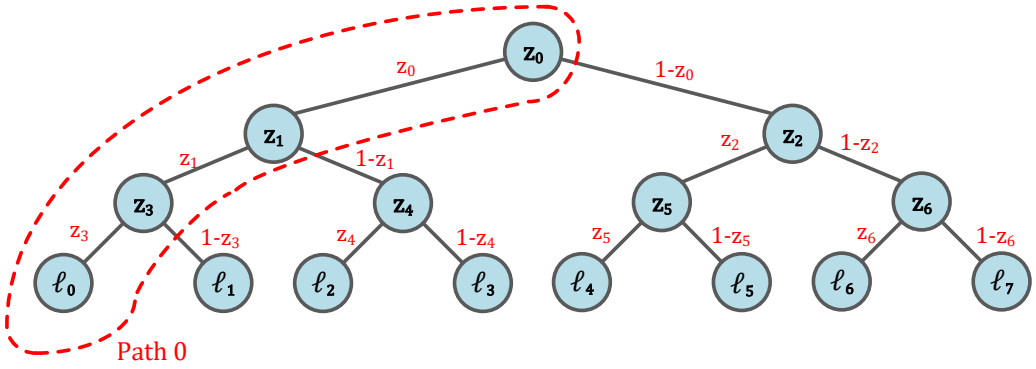


Figure 3.4 A sample binary XGBoost tree of depth of 3

[1]Recall that z_i values are in fact homomorphically encrypted; but we illustrate our algorithms and examples as if they were plaintexts for sake of simplicity.

Once the values of the branches are found, the value of all paths (p_i values) can be calculated easily by multiplying branch values on the path. Since the correct path is not revealed on encrypted data, we homomorphically calculate the values of all possible paths. Naturally, only the values of branches on the correct path are 1 while the others consist of one or more branches with the value of 0, which resets the values of incorrect paths. The leaf node on a path contains a score value that is multiplied by the value of the path. Consequently, when all path values are summed, the result in fact turns out to be the score of the leave on the correct path.

An example tree with a depth of 3 and its path calculations are shown in Figure 3.4, in which only one path out of 8 produces non-zero score, i.e. l_i. In the figure, there are 8 paths as a complete binary tree is needed for homomorphic evaluation. The path values, which will be computed homomorphically, can be given as follows (here we use plaintext values for the sake of simplicity, but in actual computations they are all encrypted):

$$
\begin{aligned}
p_0 &= z_0 \cdot z_1 \cdot z_3 \cdot l_0, \\
p_1 &= z_0 \cdot z_1 \cdot (1 - z_3) \cdot l_1, \\
p_2 &= z_0 \cdot (1 - z_1) \cdot z_4 \cdot l_2, \\
p_3 &= z_0 \cdot (1 - z_1) \cdot (1 - z_3) \cdot l_3, \\
p_4 &= (1 - z_0) \cdot z_2 \cdot z_5 \cdot l_4, \\
p_5 &= (1 - z_0) \cdot z_2 \cdot (1 - z_5) \cdot l_5, \\
p_6 &= (1 - z_0) \cdot (1 - z_2) \cdot z_6 \cdot l_6, \\
p_7 &= (1 - z_0) \cdot (1 - z_2) \cdot (1 - z_6) \cdot l_7.
\end{aligned}
$$

Algorithm 3 presents all the steps taken in the calculation of path scores on a complete binary tree. First, for every path, we need to find out the values of all internal nodes (i.e., the binary result of a comparison result, z_i). In the algorithm they are given as inputs. In Figure 3.4 the nodes z_0, z_1 and z_3 are on Path 0 (p_0). Since the node indexing is fixed to the breadth-first, the nodes' branches on each node can be kept in a look-up table. Alternatively, we can calculate the parent nodes dynamically. It is easy to find the parent nodes given the path index since parent node's index equals to $\lfloor \frac{index-1}{2} \rfloor$, where *index* is the current node's index. Although leaf nodes are indexed differently in Figure 3.4, they can also be indexed in the same manner. For example l_0 in the example 3.4 can be considered as 7^{th} node. Being left or right child is also important in path calculation. As explained above, the value of the right branch is defined as the complement of the node value.

This information can also be found in the index. If *index* is an odd integer, the node is the left child.

Algorithm 3 Tree Score Calculation

procedure CALCTREESCORE(Z, L)
  Z: set of the internal nodes which contains comparison results
  $L = \{ l_0, l_1, ...l_{2^{d_{max}}-1} \}$ is the set of leaves ▷ $2^{d_{max}}$ is the number of paths
  for each path p_i **do**
    $path \leftarrow \emptyset$
    $index \leftarrow i + 2^{d_{max}} - 1$
    for k **from** 0 to d_{max-1} **do**
      $t \leftarrow index - 1$
      $index \leftarrow \lfloor t/2 \rfloor$
      if t is even **then**
        $path \leftarrow path \cup z_{index}$
      else
        $path \leftarrow path \cup (1 - z_{index})$
      end if
    end for
    $pathValue \leftarrow$ BinaryTreeMultiplication($path$)
    $p_i \leftarrow pathValue \cdot l_i$
  end for
  $TreeScore \leftarrow$ sum of all p_i
  return $TreeScore$ ▷ Tree score
end procedure

During the calculation of path scores, the multiplication statements contain (d_{max}+1) elements, where d_{max} is the depth of the XGBoot trees. Therefore, the multiplication cost of the procedure increases substantially if the multiplication operation is performed sequentially. In order to minimize the multiplication cost of the procedure, the binary tree multiplication method (see BinaryTreeMultiplication() method in Algorithm 3) is used. In this method, pairs of elements are multiplied concurrently until only one element is left. Thus, the tree score calculation requires only $\lceil \log_2 (d_{max}) \rceil$ multiplications.

3.4 Inference Operation on a Query Data

Although tree-based machine learning models are of different architectures, their inference part requires the same process: traversing the tree based on the decision rules to reach the correct leaf node. The proposed encoding method enables these

operations to be performed on encrypted data. Thus, it is feasible to implement the inference algorithm of a tree-based ML model. This work focuses on solely the XGBoost algorithm since it is one of the most popular ensemble tree algorithms providing highly accurate results for inference in many applications.

In our study, a client-server model is adopted to demonstrate the feasibility of the proposed method. In this scenario, as depicted in Figure 3.2, an XGBoost model is deployed in a trusted server[2]. The server stores the node values of trees, both internal and leaf nodes, and the split points of each feature. It is assumed that only the split points are accessible by clients that are data owners, who query the server with their data points. In our applications, inference for a data point results in a class, to which the data point is likely to belong; thus we use the terms inference, prediction and classification interchangeably. When a client wants to use the model to classify its data, it first gets the encoding rules including split points from the server. After encoding the value of each feature by the received rules, it encrypts the resulted encodings with its own public key. Thus, the client ensures the security of its data and sends the resulting ciphertext values to the server. Then, the server calculates the model's prediction result using these ciphertexts. Finally, the encrypted prediction result is sent back to the client so that it can be decrypted.

The model owner may need to adapt the ML model for homomorphic evaluation before deploying it to the server due to some possible restrictions of the homomorphic operations. They are explained in detail in Section 3.4.1

3.4.1 Preprocessing of the Model

As mentioned earlier, an XGBoost model is an ensemble of trees. During the inference process, the input data is evaluated on all of the trees of the ensemble. Since the same operations are performed on each tree, the proposed homomorphic XGBoost inference model harnesses the total usage of the SIMD operations. However, the data structure should be uniform and consistent to use the SIMD operations appropriately. For instance, some trees may not grow to the defined depth (d_{max}) in the model. Additionally, the number of split points for features may vary, resulting in different lengths for the encodings of these features. Such differences lead to problems in the implementation of SIMD operations. Therefore, the deployed model

[2]The sever is fully trusted for the model owner as the model is uploaded in plaintext. For the data owner, it is semi-trusted as it follows the protocol step, but curious for the data leaked naturally through the protocol.

to the server is altered to prevent these potential issues.

The features may have a different number of split points. The number of a feature's split points specifies its encoding's length. To use SIMD operations for the whole model, avoiding a collection of various encoding lengths is necessary. Thus, in order to equalize the encoding lengths, dummy points are added to the split sets with fewer elements. The targeted encoding length n_E, or number of digits in the encoding, is expressed as

$$n_E = 2^{\lceil \log_2(n_F) \rceil},$$

where $n_F = \max(n_{f_i})$ is maximum of the number of split points of all features in the XGBoost model.

Theoretically speaking, equalizing the encoding lengths to the maximum of the number of split points should suffice for a basic SIMD operation. However, batch summation (EvalSum) operation is used in the inference operation as shown in Section 2.2.3, which works better with the batches when the number of operands is a power of 2. Thus, for ease of general use, the number of bits is rounded to the next larger power of 2. Consequently, n_E is a power of 2 in our implementation.

Another point is the number of nodes in the trees. The depth of the trees in the model is given as a parameter while creating an XGBoost object. The XGBoost algorithm takes this parameter as the "maximum" depth of the trees (i.e., d_{max}). However, not all trees reach the maximum depth since during the training, the algorithm prunes the redundant branches that do not significantly affect the model's accuracy. The pruning process leads to a decrease in the execution time of inference and the memory requirement of the model. However, the missing branches leads to trees with unequal number of nodes that prevents the use of SIMD-style operations of homomorphic schemes. Also, due to hommmorphic execution, all paths from the root node to leaves should be evaluated as otherwise the computation would leak information about the model. Therefore, the missing branches are completed with dummy nodes that are ineffective on the result score.

After these modifications, the model owner should encode the node values of the model according to the model encoding method explained in Section 3.2. Since all variations in the model are now eliminated, the encoded values can be grouped to get the full benefit from SIMD operations. The nodes that will be processed under the same operation should be kept together, which makes it meaningful to store all nodes with the same index in an array. For instance, the split points of all j^{th} nodes of all trees are calculated together with a single instruction. The resulting vector

for the j^{th} nodes can be shown as

$$M_j = [E_{s_j^0}, E_{s_j^1}, .., E_{s_j^{n_{TT}-1}}], \tag{3.2}$$

where j is the node index for $j = 0, 1, \ldots, 2^{d_{max}} - 2$ and n_{TT} is the number of total trees in the model.

3.4.2 XGBoost Inference

In each of XGBoost trees of depth d_{max}, there are $2^{d_{max}} - 1$ internal nodes, indexed from 0 to $2^{d_{max}} - 2$ following the breadth-first order. In the XGBoost model, the values of j^{th} nodes in the trees are stored in vectors, elements of which contain the encodings of split values at the corresponding nodes of all trees. Therefore, each vector, M_j, is a bit string of length $n_E \cdot n_{TT}$, where n_E is the number of bits used to encode each split point and n_{TT} is the number of total trees (see Equation 3.2). For instance, M_0 contains a bit string corresponding to encodings of the split values of all root nodes. And finally, we have the vector $\overrightarrow{M} = \{M_0, \ldots, M_{2^{d_{max}}-2}\}$, which holds the split values of all nodes of all trees. Therefore, $\overrightarrow{M}$ is aptly referred as *the model*. Using this batching technique, the homomorphic evaluation can benefit from the advantages of SIMD operations of homomorphic encryption schemes.

The inference procedure is initiated by the client and in return, the server sends the encoding rules, i.e., the number of trees (n_{TT}), features, and the buckets of features (also, the number bits used to encode split points (n_E)). Using Algorithm 2, the client can encode each feature value of the query data. Note that both n_{TT} and n_E can be larger than necessary to exploit SIMD operations and to enhance privacy of the model. For instance, the CKKS scheme supports batch encryption of vectors with length of $N/2$, where N is the ring dimension; therefore the server can share the maximum length to prevent to hide the number of trees in the model. For the sake of simplicity, the real values of n_{TT} and n_E are used.

Since the inference algorithm is implemented based on SIMD operations, the query data format has to comply with the structure of the model; for instance, its size and the order features used in the XGBoost trees. However, since the model is hidden (no access to the model is provided to the data owner/client), the client is not informed with the order, the features are arranged into the batches as it contains sensitive information about the model and must be kept secret to ensure the privacy of the model. For example, at the root nodes of the XGBoost trees,

the following features can be used in this order: $f_{O_0}, f_{O_1}, \ldots, f_{O_{n_{TT}-1}}$, which is not revealed to the client. In fact, even the server does not have to be informed with the feature orders at the nodes, as OrderAndMask operation can be performed homomorphically as explained below. Thus, the client is unable to sort the feature encodings in the batches as required by the model. The client, instead, encodes and then encrypts the features of a query separately, and the server performs the ordering homomorphically.

For this, client first encodes the feature value, f_i and obtains E_{f_i}, where $|E_{f_i}| = n_E$. The client creates n_{TT} copies of E_{f_i} with concatenation using the following formula

$$V_{f_i}(j) = E_{f_i}(j \bmod n_E),$$

where $V_{f_i}(j)$ and $E_{f_i}(j)$ represent individual bits of the binary vectors, V_{f_i} and E_{f_i}, respectively. To exploit SIMD operations of homomorphic encryption schemes, $V_{f_i}(j)$ is flattened into a one dimensional array.

The client, then encrypts the encoded binary string V_{f_i} by placing each bit in one of the homomorphic ciphertext slots and obtains the ciphertexts for each feature

$$C_{f_i} = Enc(V_{f_i}), \tag{3.3}$$

for $i = 0, 1, \ldots, m-1$, where m stands for the number of features. It finally sends the resulting ciphertexts C_{f_i} to the server.

The server receives one ciphertext per features from the data owner[3], which needs to be organized into the order adopted in the model M_i for $i = 0, \ldots, 2^{d_{max}} - 2$ (see Equation 3.2). But, this must be performed homomorphically by the server, which may or may not know the order. For this, the server needs so called masks for each feature at each node as explained in the following.

Suppose the order of features for the j^{th} nodes of the XGBoost trees is $O_0, O_1, \ldots, O_{n_{TT}-1}$. Note that one feature can occur more than one tree at the nodes with the same index. Then, the masks for the j^{th} nodes for the features $f_0, f_1, \ldots, f_{n_f-1}$ are generated using Algorithm 4.

The mask generation algorithm simply sets certain n_E bits of the mask for the feature f_i, if the feature is in the j^{th} node of the corresponding three. The bit range

[3]If $n_E \cdot n_{TT} > N/2$, more than one ciphertext is needed for each feature in the CKKS scheme. To keep the discussion simple, we assume $n_E \cdot n_T \leq N/2$.

Algorithm 4 Mask Generation for the j^{th} nodes of the XGBoost Trees

```
1: procedure MaskGeneration({O_0,...,O_{n_TT-1}}, {f_0,...,f_{m-1}})
2:     for i from 0 to m-1 do
3:         Mask_j^i ←< 0,...,0 >          ▷ |Mask_j^i| = n_E · n_TT; mask for f_i
4:     end for
5:     for i from 0 to m-1 do
6:         for t from 0 to n_TT-1 do
7:             if f_{O_t} = f_i then
8:                 for k from 0 to n_E-1 do
9:                     Mask_j^i[tn_E + k] ← 1
10:                end for
11:            end if
12:        end for
13:    end for
14:    return Mask_j                      ▷ The masks of j^th nodes for all features
15: end procedure
```

tn_{TT} to $tn_{TT} + n_E - 1$ corresponds to t^{th} XGBoost tree[4].

After the masks of the j^{th} nodes are generated, the mask and order function MaskAndOrder is applied using Equation 3.4.

$$F_j = \sum_{i=0}^{m-1} Mask_j^i \circ C_{f_i}. \tag{3.4}$$

Here, the symbol $\circ$ stands for homomorphic bitwise multiplication of two vectors. Also, F_j represents the ciphertext of feature values of the query, which are organized with the same order as the model M_j for the j^{th} nodes (see Equation 3.2). Then, we apply the comparison operation at the j^{th} nodes by multiplying homomorphically F_j and M_j bitwise; i.e. $F_j \circ M_j$.

Example 3.4.1. Figure 3.5 illustrates a simplified example of the mask and order operation followed by the comparison operation. In the figure, the features, each of which is encoded into 4-bit binary numbers, are homomorphically bitwise multiplied with their masks for the j^{th} nodes. Before the multiplication, C_{f_i} values are the encryption of the encoding of the value f_i, which is repeated n_{TT}. In the example, we just show the plaintext bits for the sake of simplicity, but the bit strings in Figure 3.5 are homomorphically encrypted.

We see from the figure the feature f_0 is used in the j^{th} nodes of the first and the last trees. After the bitwise multiplication of the features and their masks, the resulting ciphertexts are summed to a single ciphertext, which contains the feature valules in

[4]For instance, the first n_E-bit corresponds to the first tree, and so on.

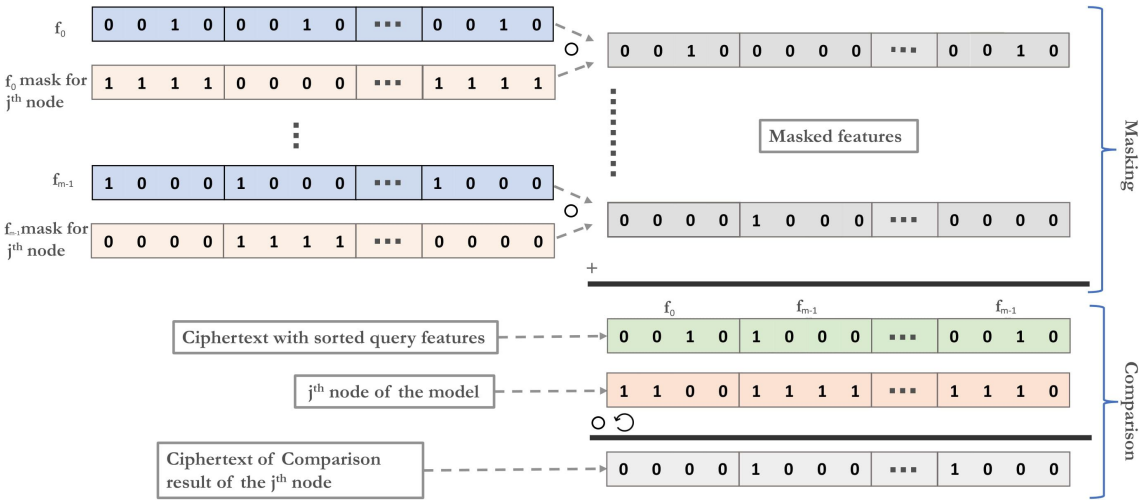


Figure 3.5 Masking, order and comparison operations: Query ciphertexts of the features are masked to create query ciphertext that has same feature ordering of the j^{th} nodes of the trees. The ordered ciphertext is multiplied with the ciphertext of the j^{th} nodes. The resulting ciphertext contains the comparison results.

the same order as in the model M_j for the j^{th} nodes of the tree. Here the summation operation is performed using EvalAdd homomorphic operation. Thus, ciphertext of features for the j^{th} nodes, F_j, can now be homomorphically compared with the model M_j. The result, in the figure, shows that the comparison operation in the j^{th} node of the first tree returns 0, while in the second it returns 1.

Algorithm 4 and Equation 3.4 are repeated for all nodes of the XGBoost trees and we obtain the the comparison results at all the nodes of the XGBoost trees. The results are kept in a vector of ciphertexts, $\overrightarrow{Z} = Z_0, Z_1, \ldots, Z_{2^{dmax}-1}$, where the ciphertext Z_j contains the comparison results at the j^{th} nodes of XGBoost trees. After homomorphically computing node comparions, there are several steps that need to be performed. The following steps are given in Algorithm 5.

After the Steps 2 and 3 of Algorithm 5 are completed, the comparison results are homomorphically computed and stored in the vector $\overrightarrow{Z}$. In the final phase of Step 3, the result of the comparison can be in any of the bits in the n_E-bit interval of the corresponding tree. For instance, suppose Z_j is the comparison results of the j^{th} nodes. Then, the bit range for the t^{th} tree will be $Z_j[tn_E]$ to $Z_j[tn_E + n_E - 1]$. In order to put the comparison result in $Z_j[tn_E]$, we use EvalSum function that sums every n_E bits of Z_j and put the result in $Z_j[tn_E]$.

Then, Step 4 computes the score of each tree using Algorithm 3 using the batching technique of homomorphic encryption schemes. Recall that Algorithm 3 computes the score of a single XGBoost tree. However, since the scores of the nodes of all trees with the same index values are now stored as a batch in a single ciphertext,

Algorithm 5 XGBoost Inference

Input:

$\overrightarrow{C} = \{C_{f_0}, \ldots, C_{f_{2^{d_{max}}-1}}\}$ ▷ Vector of encrypted feature values

$\overrightarrow{L} = \{L_0, \ldots, L_{2^{d_{max}}-1}\}$ ▷ vector of leaf scores

$\overrightarrow{Mask} = \{\{Mask_0^i\}, \ldots, \{M_{2^{d_{max}}-1}^i\}\}$ ▷ Vector of masks; $i = 0, \ldots, n_F - 1$

$\overrightarrow{M} = \{M_0, \ldots, M_{2^{d_{max}}-1}\}$ ▷ Model

Output:

R ▷ Inference result

1: **procedure** XGBoostInference ▷ XGBoost Inference Algorithm with HE
2: $\overrightarrow{F} \leftarrow$ MaskingAndOrder$(\overrightarrow{C}, \overrightarrow{Mask})$
3: $\overrightarrow{Z} \leftarrow$ CompareNodes$(\overrightarrow{F}, \overrightarrow{M})$
4: $TreeScores \leftarrow$ CalcTreeScores$(\overrightarrow{Z}, \overrightarrow{L})$
5: $Score \leftarrow$ EvalSum$(TreeScores, n_T)$
6: $R \leftarrow$ FinalMasking$(Score, Mask_{last})$
7: **return** R ▷ Return the result to the client
8: **end procedure**

when we apply Algorithm 3 to the vector of ciphertexts $\overrightarrow{Z}$, it calculates the scores of trees in a SIMD fashion.

In Step 5, the EvalSum function is used to homomorphically sum the scores of all trees in to a single score and put the result in the leftmost slots of the ciphertext $Score$.

In multiclass XGBoost classification, the score of each class is computed separately by summing up scores of the corresponding trees. If the model is encoded with n_E bits and the number of trees for each class is n_T then the EvalSum function uses the batch size $n_E n_T$ for each class to perform homomorphic batch summation operation. Hence, the score of the C^{th} class is stored at the $Cn_t{n_E}^{th}$ index of the $Score$. Note that while the first bits of the intervals hold now the tree scores, the other bits also contains some information about the comparison result and they must be removed. We will come back to this discussion below and explain the employed method to prevent the data leakage about the comparison results.

Finally, one more action must be performed before sending the results to prevent information leakage from the model. In the result vector obtained after Step 5 EvalSum operation, the ciphertext slots other than the one containing the $Score$ may contain information about the tree scores. The slots other than the slots containing the class score should not contain any information to avoid this potential leak. Therefore, a final masking is performed before sending the results. In the final masking, the result ciphertext is homomorphically multiplied with a plaintext of bit vector of length $N/2$ (see Step 6, $Mask_{last}$), which contains all 0 values except for

those corresponding to the slots that contain the *Score* values. They are set to 1 to keep the resulting *Score* value, and remove all other information. In this way, the result ciphertext contains only the class scores and is sent to the client to be decrypted and evaluated.

4. EXPERIMENTAL RESULTS

In this section, we give experimental results of the proposed method which are implemented using the CKKS homomorphic encryption scheme. The performance of the XGBoost inference is analysed on different data sets for both binary and multi-class classification. Execution times and accuracy metrics are evaluated on different setups. Effects of the depth of the trees and the number of trees on execution times measured. Also, the classification results of the proposed method are evaluated during the analysis. Section 4.1 provides the details of the data sets used in the XGBoost model training and our experimentation. Then, Section 4.2 details the model generation. Homomorphic encryption setup and the chosen CKKS parameters are detailed in Section 4.3. Finally, we give the performance results in Section 4.4

4.1 Data Sets Used in Experiments

For the experiments, two datasets are retrieved from the UCI Machine Learning Repository (Dua & Graff (2017)) and one dataset is provided by the IDASH'20 Secure Genome Analysis [1]. The characteristics of the datasets are explained below. Also, Table 4.1 summarizes the important characteristics of the data sets.

- **IDASH Data Set**: Tumor classification dataset provided by IDASH'20 Competition. The dataset consists from 2713 patient samples with 46327 features among 11 different cancer categories. The features of the dataset represents the copy number states for each tumor category which indicates the deletion and amplification on genes. The gene mutations are given in 5 different levels where -2 and -1 represent a deletion while 1 and 2 represent an amplification. In other words, the features take values from the set $\{-2, -1, 0, 1, 2\}$.

[1] `http://www.humangenomeprivacy.org/2020/`

Table 4.1 Summary of the datasets: Classification type, number of the features and the size of the datasets (number of instances/samples)

Dataset	Classification	#Features	Size
Heart	Binary	75	303
Parkinsons	Binary	23	197
IDASH	Multiclass (11)	46327	2713

- **Cleveland Heart Data Set**: The Cleveland heart data set by Dua & Graff (2017) is used to detect the presence of heart disease in patients; hence, it is a binary classification data set. It contains 303 samples and has 75 features, in total. However, the all published works that use this data set utilized at most 14 of them.
- **Parkinsons Data Set**: A dataset for the detection of Parkinson's disease (Little, McSharry, Roberts, Costello & Moroz (2007)). It consists of 197 data points that are composed from 31 patients' (23 with Parkinson's disease) biomedical voice measurements and there are 23 features, in total. This data set is also used for binary classification.

4.2 Model Training and Classification Results

In machine learning applications, certain preprocessing operations on the dataset are usually needed before training. In the experiments in this thesis, some rudimentary preprocessing methods are also applied to all data sets. For instance, feature selection is applied to select the most relevant features and the samples with too many missing values are discarded. After preprocessing, 80% of each dataset is seleceted for model training and the rest for the testing (or prediction).

Hyperparameter tuning is a crucial step to leverage the performance of ML models. Various sets of parameters should be considered to obtain the optimum hyperparameters for the learning algorithm. For XGBoost models, we considered three main hyperparameters that have significant impact on the general performance of the trained model:

- **Maximum Depth (d_{max})**: Usually, the accuracy of a model tends to increase as the depth of the trees increases. Yet, Hastie, Tibshirani & Friedman (2001) showed that the accuracy of the models rarely improves when the depth is 6 or

more. In light of this observation, we trained models with depths from 2 to 6.

- **Number of Trees (n_T)**: The number of trees (a.k.a. estimators) is one of the fundamental parameters that affects the complexity, accuracy and size of the model. We used following number of trees to train models: $\{24, 32, 48, 64, 72, 128\}$.
- **Learning Rate (t_ℓ)**: This parameter affects the complexity of the model. We used this parameter to generate models with different characteristics. We tuned the models with these set of learning rate values: $\{0.06, 0.07, 0.08, 0.09, 0.1, 0.12, 0.3\}$

The best hyperparameter sets are selected using the grid search method. This method exhaustively search the best parameters over the specified set of hyperparameters. Also, it applies the *k-fold cross-validation* technique to reduce the *overfitting* problem. The cross-validation method splits the dataset into k folds then, trains and tests several models using the different combinations of the folds. In this way, it ensures that the selected hyperparameters are aligned with the whole dataset. This work uses $3-$fold cross validation in hyperparameter evaluation. The accuracies and the behaviours of the all models generated by the cross validation were similar. This similarity shows that the selected parameter set suits the whole dataset rather than representing only a small subset of it. In this way, the validity of the parameters is proved.

The full set of the training results is given in Appendix A. Due to high number of models trained, here we present the performance results using a carefully selected subset of the trained models to improve the readability of the thesis. Interested readers are referred to Appendix A for the implementation results of all models.

The tuning and cross-validation processes yield optimal hyperparameters for high-accuracy models. We obtained a different set of hyperparameters for all our datasets by tuning. For the heart dataset, we tuned 6 different sets for hyperparameters; each with a different combination of parameters (i.e., d_{max}: Maximum depth of the trees and n_T: Number of trees/estimators) to measure the method performance over different models. All selected parameter sets are given in Table 4.2. d_{max}, n_T, t_ℓ and m' columns correspond to the maximum depth of the trees, number of trees, learning rate and the number of features used in the algorithm, respectively. Every setup is given an id (1^{st} column of Table 4.2), which is a combination of the dataset name and the setup number. The prefix letters 'H', 'I' and 'P' in the model IDs stand for Cleveland Heart data set, IDASH data set and Parkinsons data set, respectively.

After the hyperparameter selection, the XGBoost models are trained for each setup

Table 4.2 Selected XGBoost Hyperparameters for Training

Model ID	Dataset	d_{max}	n_T	t_ℓ	m'
H53	Heart	2	128	0.08	13
H48	Heart	2	24	0.08	13
H2	Heart	2	48	0.06	13
H25	Heart	2	32	0.07	13
H78	Heart	3	24	0.08	13
H132	Heart	4	24	0.09	13
I13	IDASH	2	128	0.1	2064
P3	Parkinsons	2	48	0.12	17
P10	Parkinsons	3	48	0.12	17
P23	Parkinsons	5	24	0.08	18
P31	Parkinsons	6	32	0.12	19

in Table 4.2 using the 80% of the datasets. The trained models are evaluated using 3 different metrics: accuracy, microAUC and F1 score. The accuracy metric is the most fundamental and easily understandable metric. It uses the class labels instead of the class scores. On the other hand, the microAUC uses the class scores. This metric is especially useful when the rankings of the class predictions are important. Finally, the F1 score is also calculated from predicted classes, not from predicted scores. Different from the accuracy, F1 score is more suitable for the imbalanced data. These three metrics are chosen so that our datasets that has different characteristics can be evaluated better. Table 4.3 shows the accuracy, micro-accuracy and the F1 score of the models. The experiments and the evaluations are performed using these XGBoost models.

Among the models we trained, the ones with the highest accuracy have relatively smaller depth. To test and analyze the performance of the proposed method on high-depth models two Parkinsons models are selected despite their low accuracy values. These models P23 and P31 have depth of 5 and 6 respectively.

For the IDASH dataset, a similar preprocessing phase is applied and the best parameter set is chosen. The selected IDASH model is initially trained for the IDASH'20 competition however due to the homomorphic comparison problem explained in Section 2.4, it was not submitted to the competition. Since the proposed method, now, solves this problem, a homomorphic version of the IDASH model is included in the thesis. A comparison of the new IDASH model with the SVM model, which was submitted to the IDASH'20 competition is given in Section 4.4.3.

Table 4.3 Accuracy, microAUC and F1 scores of the selected models

Dataset	Model ID	Accuracy	microAUC	F1 score
Heart	H53	0.8852	0.939	0.832
	H48	0.8852	0.956	0.88
	H2	0.901	0.963	0.899
	H25	0.8852	0.957	0.882
	H78	0.8524	0.923	0.848
	H132	0.836	0.92	0.832
IDASH	I13	0.828	0.985	0.817
Parkinson	P3	0.927	0.975	0.811
	P10	0.927	0.96	0.88
	P23	0.89	0.92	0.873
	P31	0.916	0.981	0.887

4.3 CKKS parameters

The ciphertext parameters used in the experiments are determined as below:

- **Multiplication depth of the circuit for homomorphic inference operation**: This parameter depends on the depth of the model d_{max}. Tree score computations needs $\lceil \log_2 d_{max} + 1 \rceil$ multiplications. Masking and ordering, comparison of nodes and final masking operations require one multiplication each. Hence, the multiplication depth of the circuit is formulated as $\lceil \log_2 d_{max} + 4 \rceil$.

- **Scaling factor (Δ)**: Scaling factor the CKKS homomorphic encryption scheme depends on the multiplication depth of the circuit used in the homomorphic evaluations and the total number of operations. Although there is no precise formulation for scaling factor, the PALISADE library suggests a rough estimation method to pick a scaling factor. Accordingly, the CKKS scheme itself causes 25-bit approximation loss and each homomorphic operation, such as addition, multiplication and rotation, causes 1-bit loss. There are $\lceil \log_2 d_{max} + 4 \rceil$ multiplication operations and as many other homomorphic operations (addition and rotation) in the proposed XGBoost inference algorithm. Therefore the scaling factor is computed as $((\lceil (\log_2 d_{max}) + 4 \rceil) \cdot 2 + 25)$.

- **Security level (λ)**: 128 bit security, which is generally accepted as high security level, is selected. We use the parameters available or given by the CKKS scheme to guarantee the intended security level.

Table 4.4 CKKS parameter specifications of each model

Model ID	Multiplication depth	Δ	λ	N	$\log_2 q$
H53	5	35	128	16384	235
H48	5	35	128	16384	235
H2	5	35	128	16384	235
H25	5	35	128	16384	235
H78	6	37	128	16384	270
H132	6	37	128	16384	270
I13	5	35	128	16384	235
P3	5	35	128	16384	235
P10	6	37	128	16384	270
P23	7	39	128	32768	334
P31	7	39	128	32768	334

- **Ring Dimension (N)**: The ring dimension is decided by the PALISADE library based on the multiplication depth of the homomorphic circuit and the security level.
- **Ciphertext Modulus** q: The ciphertext modulus is another parameter determined by the PALISADE library. It depends on other CKKS parameters such as scaling factor and security level.

A summary of the selected CKKS parameters of each XGBoost model is given in Table 4.4.

4.4 Results

The details of the computing platform used in our implementation and experiments are included in Table 4.5 along with details about its specifications.

The main focus of the experiments is the computation time and accuracy of the inference or classification. The accuracy of the homomorphic XGBoost inference and the original XGBoost inference are same besides from the minor errors caused by the CKKS noise. The computation time of the proposed homomorphic XGBoost inference method is affected by the following:

- Depth of the XGBoost trees (d_{max})
- Total number of trees (n_{TT})

Table 4.5 Hardware specifications of the computing platform used to obtain implementation results

Hardware	Specs
CPU	i9-11900K
GPU	NVIDIA GeForce RTX 3070 Ti
CPU's RAM	16384 MB DDR4 2.6 GHz
GPU's RAM	8192 MB GDDR6X
CPU Clock Frequency	5.100GHz
Operating System	Ubuntu 20.04.4 LTS
Programming language	C++14

- Encoding length (n_E)
- Number of features used in the model(m')

The values of these elements on the selected models are summarized in Table 4.6. The encoding length depends on the tree depth, number of trees and the learning rate of the XGBoost training. Consequently, the model owner has no direct control over the encoding length. Hence, the computation time of the proposed method is primarily evaluated for the tree depth and the number of trees.

Table 4.6 Some model properties important in homomorphic inference operations

Model No	m'	d_{max}	n_{TT}	n_E	Min $batchSize$	Rotations
H53	13	2	128	16	2048	15
H48	13	2	24	4	96	9
H2	13	2	48	8	384	12
H25	13	2	32	4	128	9
H78	13	3	24	16	384	13
H132	13	4	24	32	768	15
I13	2064	2	1408	2	2816	9
P3	17	5	48	8	384	12
P10	17	3	48	8	384	12
P23	18	5	24	8	192	11
P31	19	6	32	16	512	13

In Section 4.4.1, the effect of the tree depths on the computation time is analyzed. Section 4.4.2 presents an execution time analysis of number of the trees in the model. Finally, the classification results of the proposed inference method is evaluated and compared with the original classification results in Section 4.4.3

4.4.1 The Effect of the Tree Depth on Computation Time

The depth of the trees is the most significant parameter that affects the multiplication depth of the circuit, which is homomorphically evaluated in our method. As the depth of a XGBoost tree increases, the number of multiplications needed in the calculation of a path's score also increases. This overhead is reduced with the help of binary tree multiplication method, which increases logarithmic with the depth. More importantly, the number of paths and number of nodes in the tree doubles as depth increments by one. Since the algorithm requires the computation of all of the node values and path scores, the depth of the model has an important influence in the execution time.

Additionally, the number of split points also tend to increase as the depth grows since an increase in the depth leads to more complex models if the other hyperparameters remain the same. Hence, the tree depth has impact on the computation time both directly and indirectly.

The computation times of a single query for different model depths for Heart dataset are given in Figure 4.1. The timings are for the sequential execution, where only one thread of CPU cores is used to run the homomorphic evaluation. For this experiments we selected the models H48, H78, H132 from Heart dataset (see Table 4.6), which have the same number of trees but different depth values; i.e., for all models, the number of trees is 24. The ciphertext modulus q is selected as 270-bit for the H132 and H78 and 235-bit ciphertext modulus is used for the H48. Ring size is the same for all three models, which is 16384.

In Figure 4.1, we give the breakdown of the total execution times based on the various operations performed during homomorphic evaluations of the circuit for classification given a homomorphically encrypted data sample. These operations, which are performed at the server side are i) homomorphic **ordering**, ii) homomorphic execution of **node comparisons**, iii) **Tree score** calculations, and iv) **final masking**. As can be observed from the figure, final masking brings negligible overhead to the total execution times. All other three operations (ordering in particular) have considerable contribution to the execution time.

The total computation time, including the preprocessing (masking, ordering) of the ciphertext, is 180 milliseconds when the depth is 2 (model H48). Since the number of nodes and paths doubles with the increment of the depth by one, it is expected that the computation time also doubles with the depth increase. Although there are some deviations due to the differences in in ciphertext size and the encoding, the

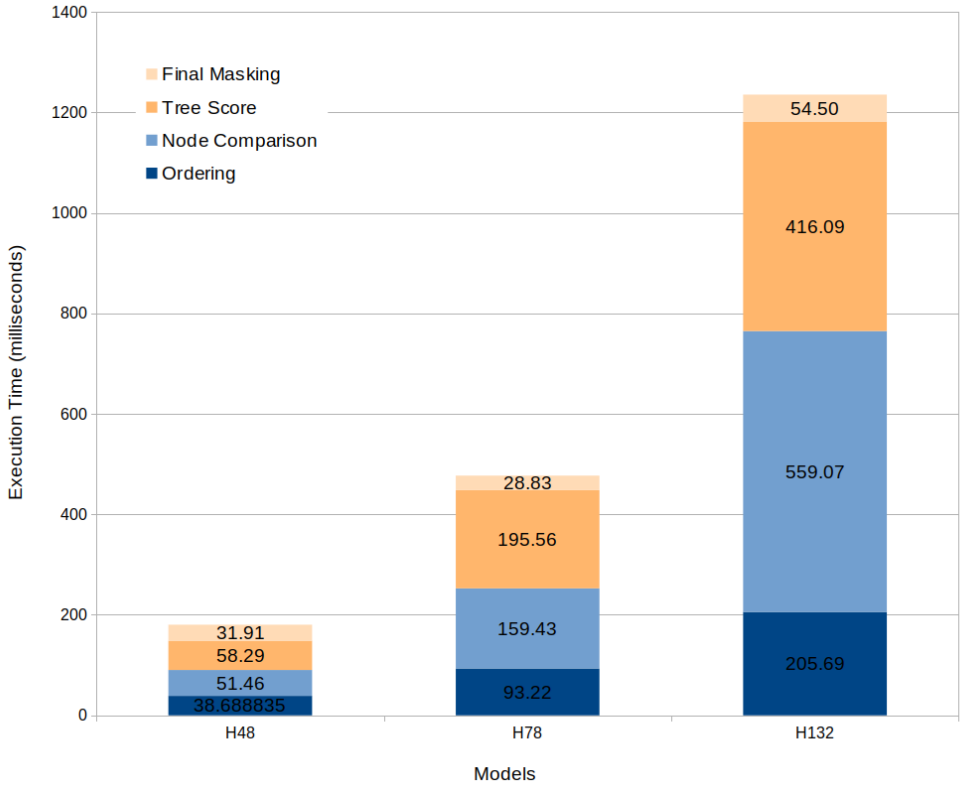


Figure 4.1 Sequential prediction time of a single query in homomorphic XGBoost Heart model

expected increase in the computation time is observed, albeit approximately.

The effect of the tree depth can also be observed in the selected Parkinsons models P3, P10, P23 and P31, which have depths of 2, 3, 5 and 6, respectively. P3 predicts the label of a single query in 216 milliseconds while the computation time of P10 is 534 milliseconds. The detailed timings are shown in Figure 4.2. The computation times of the models P23 and P31 are shown in Figure 4.3. Their prediction times for a single query are 6.9 and 16.16 seconds respectively.

As seen in the Figures 4.1, 4.2 and 4.3, when the depth increases by 1 it is expected that the execution time approximately double if the other parameters are similar. Note that this is true only the other parameters are the same. For instance, increasing the depth may also increase the ring size. In this case, the observed increase would be much more than double. P10 and P23 can be an example of this situation. Since the depth difference between these models is 2, it would be expected that the time would increase 4 times. However, the execution time of P23 is 12.9 times of P10 since the ring size of P23 is 32768 while that of P10 is 16384.

As a result, tree depth has a strong and non-negligible impact on computation times when homomorphic computation is applied.

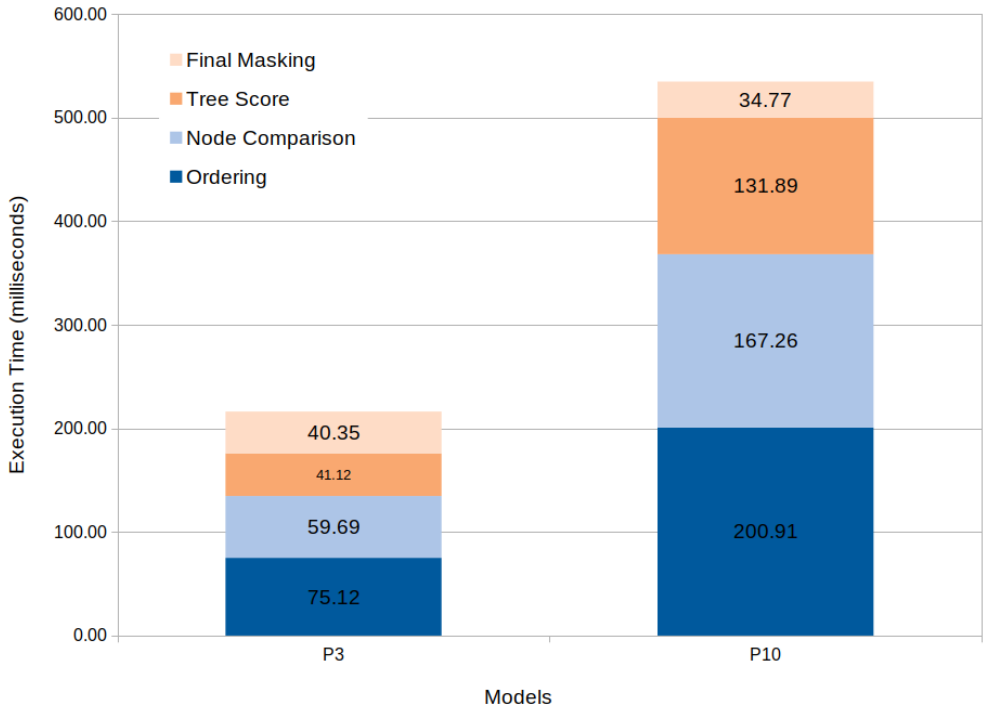


Figure 4.2 Effect of XGBoost trees depth on execution time of homomorphic classification of Parkinsons Data Set

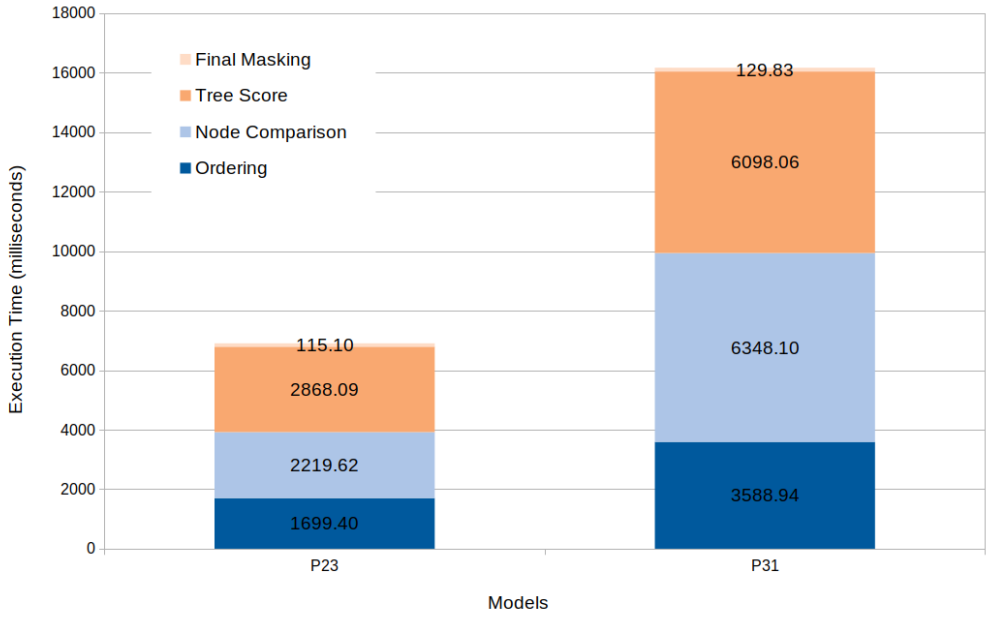


Figure 4.3 Effect of XGBoost trees depth on execution time of homomorphic classification of Parkinsons Data Set

4.4.2 The Effect of the Number of Trees on Computation Time

The number of trees in the XGBoost model is another factor with a high effect on both the model complexity and the CKKS computations. From the models introduced in Section 4.2 and summarized in Table 4.2, the models with IDs H25, H48, H53 and H2 are selected to evaluate the effect of the number of trees (n_T). These models are chosen because they were of the same depth (i.e., 2) so that the impact of n_T is more clearly observed.

The execution times of the proposed homomorphic inference's various phases performed at the server side (namely, homomorphic **ordering**, homomorphic execution of **node comparisons**, **tree score** calculations, and **final masking**) are shown in Figure 4.4. As seen in the figure, the computation times of the models are highly close with a standard deviation of 30 milliseconds. These close results are a natural and expected consequence of the batching features. Since the trees are packed into plaintexts (or ciphertext) and processed together, the number of trees has a relatively minimal and indirect impact on the results.

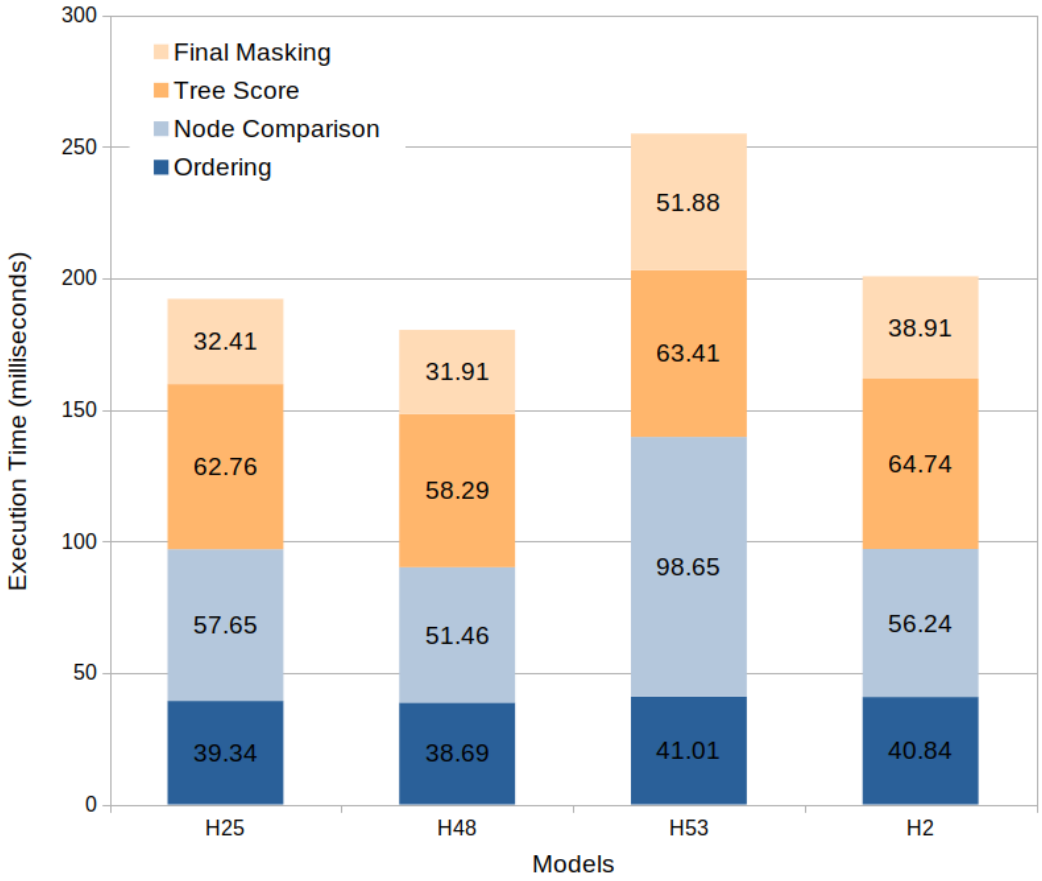


Figure 4.4 The execution time of the models with same depth: The effect of the tree number

The variations in the results are due to the indirect effects of the number of trees. The number of trees in the model affects the number of split values hence encoding length and the number of rotations. As said earlier, the XGBoost models tend to be more complex as the number of trees increases and this lead to more split points and encoding length. In our test environment, H53 and H2 models supports this proposition by having more trees and more encoding length compared to others. That being said, split numbers also depends on other XGBoost hyperparameters therefore the effect of tree numbers should be evaluated together with other parameters. Still, it is subject to further investigations for better understanding.

The other element affected by the number of trees is the number of homomorphic rotation operations. As mentioned earlier, the EvalSum function often used in the proposed method requires homomorphic rotation operations. Table 4.6 shows the number of rotations performed during the inference calculations. H53 requires more rotations than the other models discussed in this section as a result of its large tree number and encoding length. Consequently its execution time is longer than the others. Similarly, H2 model of the second largest rotation number is the slowest one after H53.

In conclusion, the number of trees indirectly changes the computation time by changing factors such as the number of rotations and encoding length. Nevertheless, these changes can be regarded as minor variations considering the total time since it changes the execution time approximately by $10-15\%$.

4.4.3 Classification Accuracy of the Proposed Inference Method

Since the CKKS scheme is an approximate HE scheme, it inherently introduces some approximation error and each operation adds up extra noise to the ciphertext. Consequently, some amount of precision loss is expected in classification scores. The loss or noise in the resulting scores should be small enough to preserve the accuracy of the model. In this section, the proposed method's prediction results are evaluated by comparing it with the original classification scores.

As mentioned before 80% of all data sets are used in training. The rest is used to test the accuracy of the models. The test data is evaluated first using the XGBoost's original prediction function without any encryption. The prediction performance of the models are already shown in Table 4.3. Later, the proposed homomorphic inference method is tested using the same test dataset. Table 4.7 shows the accuracy

of both predictions: Accuracy of the original XGBoost models are given in column named *Original Accuracy*, the accuracy of the proposed inference method is given in *Homomorphic Accuracy* column and finally, the relative error between them is given at the last column named *Relative error*.

Table 4.7 Accuracy comparison of original models and proposed inference method

Model ID	Original Accuracy	Homomorphic Accuracy	**Relative Error**
H53	0.8852	0.8852	**0**
H48	0.8852	0.901	**1.8**
H2	0.901	0.918	**1.8**
H25	0.8852	0.901	**1.8**
H78	0.8524	0.868	**1.9**
H132	0.836	0.803	**3.9**
I13	0.828	0.828	**0**
P3	0.927	0.897	**3.2**
P10	0.927	0.948	**2.3**
P23	0.89	0.794	**10.7**
P31	0.916	0.821	**10.4**

Since there are 3 different datasets used, it would be more appropriate to evaluate models produced from the same dataset together.The IDASH model, I13, has the same predictions as the original model, hence its relative error is 0. For the Cleveland Heart data set, the relative error is at most 1.9% for the models with depth 2 and 3 (i.e., H53, H48, H2, H25, H78), as shown in Table 4.7. The error rises to 3.9% when the depth increases to 4 (Model H132). A similar increase is also observed in Parkinsons models. The model P3 has 3.2 relative error, while that of P10 is 2.3. When the depth rises to 5 and 6 in P23 and P31 respectively, the relative errors also go up to 10. It appears from these results that there is a correlation between the relative error of the proposed method and the depth. Models with a depth of two or three differentiated at most one sample from the original model, while the model with a depth of 4 classified 2 samples differently. When the depth reaches 5 or 6, the relative error worsens.

In the models with deeper trees, it is expected that more error accumulates in the ciphertexts due to the higher number of multiplication operations. Therefore, it is somewhat expected that there will be more errors in models P23 and P31 than in the others and the results given in 4.7 proved it.

When we further investigate the relative errors, it is observed the samples that are classified differently from the original model have prediction scores close to 0. In the binary classification the class label is decided based on the positiveness of the prediction score. So, when some error is added in the prediction, the samples with

scores close to zero are the first ones to be affected. There are not any relative error in the multiclass model (I13), however, the equivalent of this situation in a multiclass model would be that the score of the two classes being very close to each other. As a summary, the error first affects the samples from where the classes converges each other. When the depth increases, the total error also increases, so the span affected by the error will also expand.

In conclusion, the proposed method shows minimal accuracy degradation and has decent performance especially for the models with shallow trees. Moreover, the multiplication depth is the most important factor for the successful realisation and application of the proposed model. If possible, shallow models should be preferred to get better performance. In our experience, the high accuracy models with shallow trees can be generated easily. Therefore, the proposed method can be easily applied to other datasets.

Performance of the proposed XGBoost method compared to the other homomorphic ML inference algorithms can also be investigated. For this purpose, the models submitted to the IDASH competition and their results were used. As mentioned earlier, the XGBoost model initially trained for the competition could not be adapted to HE due to the homomorphic comparison problem, which is the subject of this thesis. Instead, an SVM model, which does not need a comparison process, was trained for the competition. The Homomorphic SVM inference algorithm involves a simple homomorphic matrix multiplication and addition. Although it is a fast algorithm because of its computational simplicity, it is not as accurate as the XGBoost model. Among the models trained for IDASH data the most accurate SVM and XGBoost models are selected. The SVM model uses 187 features in prediction and has microAUC score of 0.96. On the other hand the XGBoost model uses more than 900 features and has 0.985 microAUC as given in Table 4.8.

Table 4.8 Performance Table: Execution times (Key generation, encryption, computation, decryption and total timing) and MicroAUCs

	KeyGen	Enc	Comp	Dec	EndtoEnd	microAUC
SVM	8ms	6ms	17ms	13ms	44ms	0.96
XGBoost(I13)	111ms	34ms	298ms	14ms	457ms	0.985

The execution times of a single query on both SVM and XGBoost models are also given in Table 4.8, in which key generation (**KeyGen**), encryption (**Enc**), inference computation (homomorphic evalutaiton) (**Comp**), decryption (**Dec**) and end-to-end (**EndtoEnd**) timings are given. Here, we are interested primarily in homomorphic compuptation ("Comp" in the table) and the others are given for the sake of completeness. Since the IDASH'20 competition scenario assumes the model is

not hidden, the inference computation time results in Table 4.8 contain contain no Masking and Ordering and Final Masking steps which are introduced in this thesis to ensure the security of the model, as well. This results shows that the XGBoost model is superior to SVM model in terms of accuracy. Although the XGBoost model is slower than the SVM model in terms of computation time, the discrepancy is not prohibitively large to prevent it from being used in practice. Considering its accuracy advantage, its high execution times can be tolerated.

The competition organizers ranked the submissions using a different dataset and the end-to-end timing results are given in Table 4.9. Our SVM submission predicted their test dataset with 0.93 microAUC in about 16 seconds (see the fifth row in the table with **gencSU**). Since a different test dataset is used in the competition results, it is not possible to compare the homomorphic XGBoost model directly with them. Nevertheless, when compared with the SVM model, the proposed method seems to have a noteworthy achievement as its accuracy compares favorably with those given in Table 4.9.

Table 4.9 IDASH'20 End-End time sorted rankings

Team Name	Institution	Country	End-End Time (Sec.)	AUC
Chimera	Inpher	Switzerland	0.74	0.9704
SamsungSDS	SamsungSDS	South Korea	3.39	0.9745
Alibaba Gemini Lab	Alibaba Group	China	3.82	0.9668
Rosetta	PlatON and Matrix Elements	China	13.58	0.9598
gencSU	**Sabanci University**	**Turkey**	**16.01**	**0.9380**
Desilo	Desilo	South Korea	43.70	0.9774
Team Genigma	Sandia National Laboratories	USA	51.00	0.9218
Team Inspire	Georgia State University	United States	61.00	0.8846
A*FHE	A*STAR	Singapore	186.01	0.9774
CodeHopper	Temple University	United States	239.79	0.9542
SNU	Seoul National University	South Korea	286.13	0.9857
NYUAD-Yale	NYU Abu Dhabi / Yale University	UAE / USA	308.32	0.9286
LangWolf	Wuhan University Of Technology	China	48mins	0.8607
ANT_SCI	Ant Group	China	N/A	N/A
BitSecure Group	Tsinghua University	China	N/A	N/A

5. SECURITY DISCUSSION

In an ideal PPML algorithm, all information about the queries, model, and results has to be confidential. Since the security of the presented work is based mainly on the security of the CKKS homomorphic encryption scheme, and the security analyses of the homomorphic encryption and the CKKS scheme are given in Sections 5.1 and 5.2, respectively. The possible privacy concerns of the client are addressed in Section 5.3. Finally, the confidentiality of the XGBoost model is discussed based on different adversarial models in Section 5.4.

5.1 Security of Homomorphic Encryption

As mentioned in Section 2.2, most of the practical HE schemes are based on the RLWE problem, which is a lattice-based, hard mathematical problem. The security statement of these HE schemes indicates that if the scheme is broken, an efficient solution to RLWE hard problems exists (*Decision* and/or *Search* problems). Currently, numerous reliable works have proven the RLWE problem's hardness (Balbás (2021)), which confirms that the RLWE-based HE schemes are at least as secure as the other standardized encryption schemes. Additionally, due to the nature of hard lattice-based problems, the RLWE-based schemes are considered secure against quantum computers (Lyubashevsky et al. (2013)).

That being said, there are also many lattice and RLWE attacks that focus on finding insecure RLWE instances or parameters. An open Industry/Government/Academic Consortium, HomomorphicEncryption.org, was formed to generate a community-driven secure homomorphic encryption standard. The community constructed a standardization for HE parameters by reviewing best-known lattice attacks and algorithm proposals for solving RLWE problems (Albrecht, Chase, Chen, Ding, Goldwasser, Gorbunov, Halevi, Hoffstein, Laine, Lauter, Lokam, Micciancio, Moody,

Morrison, Sahai & Vaikuntanathan (2018)).

The current standardization suggests ring dimension and ciphertext modulus pairs for 3 different security levels: 128-bit, 192-bit, and 256-bit security. Several open-source HE libraries comply with the HomomorphicEncryption.org security standards, such as PALISADE, Seal, and HELib. Consequently, when up-to-date libraries and recommended parameters are used, HE is a highly secure method.

5.2 Security of CKKS Scheme in Practice

So far, most of the homomorphic encryption schemes ensure security against the concept of INDistinguishability under the Chosen Plaintext Attack (IND-CPA). In IND-CPA secure systems, adversary, who has access to encryption oracle, cannot determine which one of given plaintext messages encrypts to a given ciphertext. This feature makes it impossible to extract information about plaintext messages from the ciphertext. So, when an adversary gets many encrypted messages, he or she cannot get an idea of the distribution or variety of the plaintext messages.

However, Li & Micciancio (2020) observe that the decryption results may leak information about the RLWE noise in approximate homomorphic encryption schemes. This leakage leads to the exposure of the secret key if the ciphertext and the decrypted value provided together. In addition to pointing out the security flaw in the approximate HE schemes, Li & Micciancio (2020) also proposed a new security notion called IND-CPA+ for the approximate homomorphic schemes. This notion allows adversary to decrypt ciphertexts. The details are covered in the article, but simply put, they have redefined the security notion to cover the the flaw they mentioned. Shortly after that, both the CKKS decryption algorithm and the libraries are updated in order to eliminate this problem (Cheon, Hong & Kim (2020)). The current CKKS implementations are secure under IND-CPA+. For example, CKKS implementation of PALISADE Library satisfies the IND-CPA+ security starting from the PALISADE v1.10.6 version.

Other than the above issue, which is currently fixed, there is no other known security flaw in the CKKS scheme. Therefore, the users choosing up-to-date versions of the libraries are safe to IND-CPA and IND-CPA+.

5.3 Privacy of The Client's Data

The client (data owner) naturally wants her data to remain confidential during hommorphic classification operation. It is important for the client to fully trust the encryption system used in the application service. In the proposed model, the CKKS homomorphic encryption scheme is used to encrypt the client data. As already explained in the Sections 5.1 and 5.2, homomorphic encryption and CKKS scheme in particular have a well standardised and fully accepted security levels. Briefly, the privacy and confidentiality of the data owner is ensured in the proposed method. Also it is assumed that the data owner and the server communicate through a secure channel, which provides confidentiality, integrity and authentication.

5.4 Privacy of The Model

The confidentiality of the model is as important as the client data since the trained model may contain information about private data, which are used in the training phase. Also, the model is propriety information, which model owner has no incentive to share without financial benefits. Ideally, all information about the model should be hidden to preserve privacy of the model. Nonetheless, there is usually a trade-off between the security and the practicability of such PPML applications, leading to algorithms, which claim to provide ideal security that are not sufficiently practical. Therefore, the real-life PPML applications choose to leak some non-crucial information about the model to ensure that the application is efficient for practical use (Curtmola, Garay, Kamara & Ostrovsky (2006), Chase & Kamara (2010)). Here, the critical point is establishing a balance between the model's privacy and the application's efficiency in practice. In this section, we investigate the advantage the data owner gains in learning the model given the information returned by the server in response to the query of data owners.

One of the goals in adversarial machine learning is *model extraction* to obtain a second model that approximates the accuracy of the original protected model. In black box attacks, adversary has no knowledge on the model, but can query the model using synthetic or real data to get the predicted label for those data points. Then, they use the query data and the predicted labels to train a local model that

approximates the target model's decision behaviour.

The proposed method in this thesis, however, shares the encoding rules with the client, which includes the split points of the model. This may provide the adversarial data owner an advantage in extracting a better model than the one in the black box attack. Thus, it is crucial to examine the possible utilization of this information in adversarial attacks for the privacy analysis of the model. Therefore, we established model extraction attack scenarios to measure the effect of encoding rules on the attack success rate. First, black box model extraction attacks are applied using real and synthetic data. In this scenario, the encoding rules are not available. In the second scenario, encoding rules are exploited to generate specially prepared query data so that the higher accuracy models can be constructed. This scenario is also repeated with real and synthetic data. In the third scenario, we examine a brute force attacks using encoding rules. These three attack scenarios aim to determine whether the encoding rules enhance the performance of the known attacks or not. The implementation details and results are given in the subsequent sections.

5.4.1 Selected Adversarial ML Tool and Model Extraction Attacks

The black-box attacks are implemented using Adversarial Robustness Toolbox (ART) library v1.11 (Nicolae, Sinn, Tran, Buesser, Rawat, Wistuba, Zantedeschi, Baracaldo, Chen, Ludwig, Molloy & Edwards (2018)). ART is an extensive machine learning security framework containing many attack, defence and estimation modules. Two of the model extraction methods implemented in this library are CopyCat CNN and KnockoffNets. The CopyCat CNN method (Correia-Silva, Berriel, Badue, de Souza & Oliveira-Santos (2018)) is originally designed to train a *substitute* CNN models by querying *target* CNN models with non-labeled samples. Similarly, KnockoffNets (Orekondy, Schiele & Fritz, 2018) method targets CNN models. This attack has two modes: *Argmax* and *Probabilistic*. The *Argmax* method uses hard-labels in the training, in other words it selects the class with highest score as the label and uses this label in the training. The *Probabilistic* approach uses the class scores of the query data directly in the training. In addition to these modes, the KnockoffNets method has two additional modes for sampling. The *Adaptive* mode uses data labels to sample best query sets. The *Random* mode selects the query data randomly from the data pool. Since the synthetic data have no labels, the adaptive mode is applied only in the attack using real data.

In order to make these attacks as compatible as possible with target XGBoost mod-

els, CNN hyperparameters are selected carefully. In addition, a naive model extraction attack is implemented where the substitute model is XGBoost. All of these model extraction attacks roughly follows the following steps:

2.1 Creating a query dataset with real or synthetic data

2.2 Querying the target model

2.3 Receiving the labels predicted by the model

2.4 Training a substitute model using query data and predicted labels

2.5 Improving the substitute model by repeating step 2, 3, 4 if necessary

This fundamental scenario can be enhanced with additional techniques such as data augmentation and advanced sampling methods. For instance, Papernot, McDaniel & Goodfellow (2016) introduced a new data augmentation and sampling method. They attacked 5 different types of models using synthetic data and generated substitute models that has approximately 85% similarity with the target models using 3200 queries. The details of the similarity amount are given in Table 5.1.

Table 5.1 Similarity between the target and substitude model: Accuracies matched between the substitutes and their target classifiers. DNN: Deep Neural Network, LR: Logistic Regression, SVM:Support Vector Machines, kNN:k-Nearest Neighbour.

	Target Model					
Substitute	DNN	LR	SVM	DT	kNN	Queries
DNN 1	78.01	82.17	79.68	62.75	81.83	51200
DNN 2	89.28	89.16	83.79	61.10	85.67	51200
DNN 3	82.90	83.33	77.22	48.62	82.46	3200
LR 1	64.93	72.00	71.56	38.44	70.74	51200
LR 2	69.20	84.01	82.19	34.14	71.02	51200
LR 3	67.85	78.94	79.20	41.93	70.92	3200

Inspired by this work, we developed a gray-box attack which uses the encoding rules as data augmentation method. It aims to enhance the diversity of the query data by eliminating the samples with same or similar encodings. The details of this attack are given in Section 5.4.4.

5.4.2 Synthetic Data Generation

The attacks are performed using both real and synthetic query data. Due to the costly and time consuming process of collecting real data, it is hard to obtain suffi-

cient amount of real data. Additionally, the limited amount of real data available is often not as diverse enough as the data used in the target model. Hence, the lack of real data is the major impediment to the success of black box model extraction attacks. Synthetic data generation is an alternative to real data that is becoming more and more prevalent.

In this thesis, DataSynthesizer (Ping, Stoyanovich & Howe (2017)) library is used to generate synthetic data. The tool has 3 modes based on the relation between the attributes: correlated attribute mode, independent attribute mode and random mode. The correlated attribute mode creates a differentially private Bayesian network that apprehends the correlation between the attributes then generates new samples using this network. This mode works better when the attributes are correlated. The library suggests the independent mode if the attributes are not sufficient to create a network or the correlation mode is computationally expensive. The independent mode generates samples based on the attribute histograms. Finally, the random mode, which simply produces type-consistent random values, is suggested for extremely sensitive data.

For the IDASH dataset the correlated attribute mode cannot be used due to the extreme computational overload. The synthetic data is generated using independent attribute and random modes.

5.4.3 Scenario 1: Basic Black Box Attack

As previously mentioned, the black-box attacks are common in adversarial machine learning. In most of the cases the adversary has no information about the model including the algorithm type, which is a typical scenario in real life applications. Therefore, the black box attacks are well studied (Bhambri, Muku, Tulasi & Buduru (2019), Shi, Sagduyu, Davaslioglu & Li (Shi et al.)) and many attack scenarios are developed that requires small amount of queries. CopyCat CNN and KnockoffNets are two examples of these attack proposals.

The performance of the extraction attack is tested and evaluated using various number of queries. First, the IDASH model is attacked using real data. For this attack 80% of the test dataset, 434 samples, are used for attack queries and the rest is used to test the accuracy of the substitute model. The extraction attack is repeated for the query sets of different sizes: $\gamma \in [50, 100, 200, 300, 400, 434]$. The amount of real data is limited to make the attack more realistic as we assume that

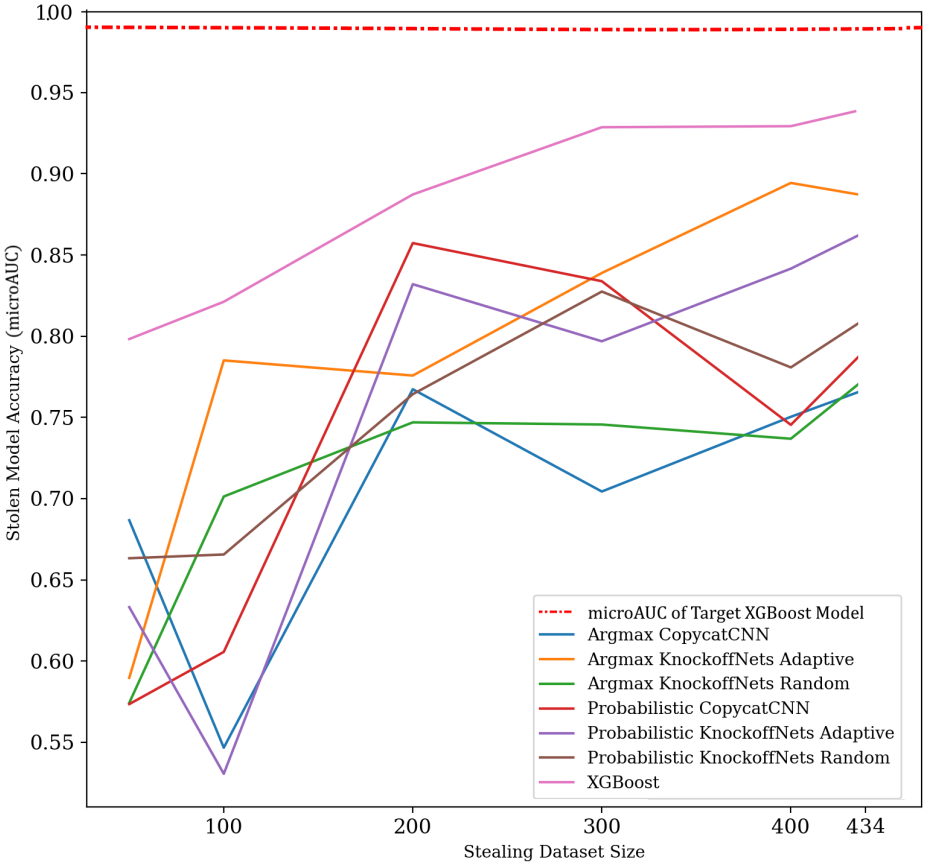


Figure 5.1 Prediction scores of the substitute models in the black-box attack with real data

a typical data owner is unlikely to posses as large and diverse data set as the one used to train the target data set.

For the attack using synthetic data two different data sets are generated using the *independent attribute* and *random* modes of the Datasynthesizer tool. Each of the sets has 2200 samples. The extraction attack is repeated for the query sets of different sizes: $\gamma \in [100, 200, 300, ..., 1000, 1200, ..., 2200]$.

The results of the extraction attack with real data is shown in Figure 5.1. The x-axis shows the size of the query dataset while y-axis is the microAUCs of the extracted (substitute or stolen) models. The score of the target model, which is shown with red dashed line in the figure, is 0.985 microAUC. The classification performance of the substitute models tends to increase with more queries.

The best results are obtained in the attack that uses an XGBoost algorithm to train the substitute model. This attack reaches 0.94 microAUC score, which is relatively close the target score.

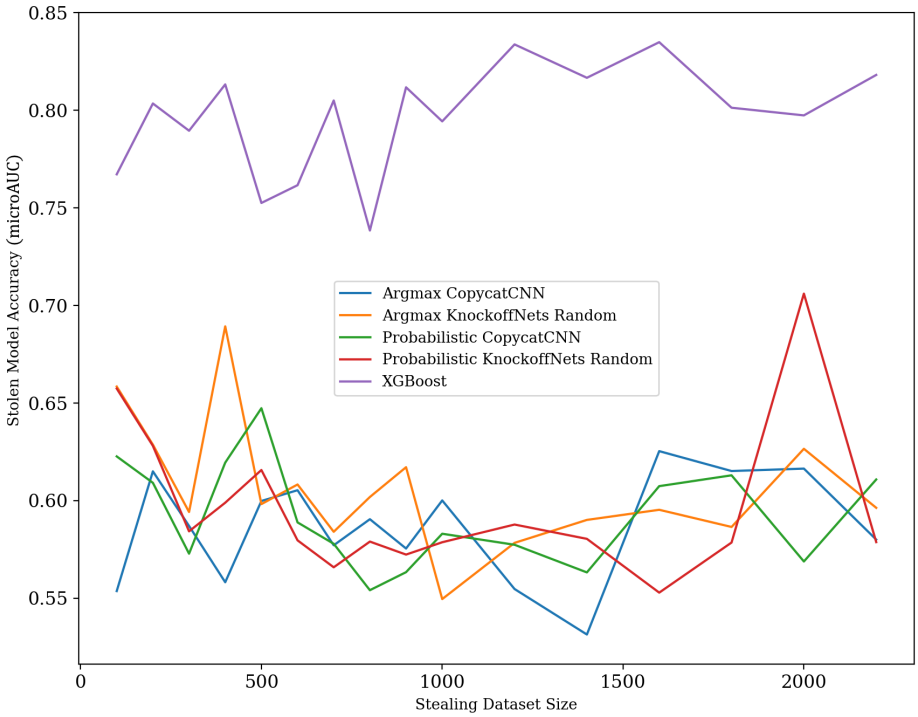


Figure 5.2 Prediction scores of the substitute models in the black-box attack with synthetic data (independent attribute mode)

Figure 5.2 includes the extraction attack results when synthetic data is generated using the independent attribute mode. The correlation between the size of the query data set and the accuracy rate is not evident in this attack. Nevertheless, the microAUC of the XGBoost substitute model shows a slight upward trend. Compared to the real data attack, this attack has much less success rate. The best of the substitute model reaches at most 0.84 microAUC. The CopyCat CNN and KnockoffNets atacks do not produce remarkable results.

The synthetic data attack is also repeated with the data produced by the random mode. The result of this attack is given in Figure 5.3. Unlike the independent mode data, there is a slight increase in the prediction scores as the number of queries increases. However, this increase is not strong and performance of the models eventually reaches a plateau after some amount of queries. The best model in this attack reaches 0.85 microAUC at its highest.

The score of 85% was obtained in the substitute model using only synthetic data however, this rate does not change significantly when the number of queries increases. Additionally, this score is much less than the prediction score of the attack that uses a limited number of real data. In conclusion, synthetic data can be an alternative

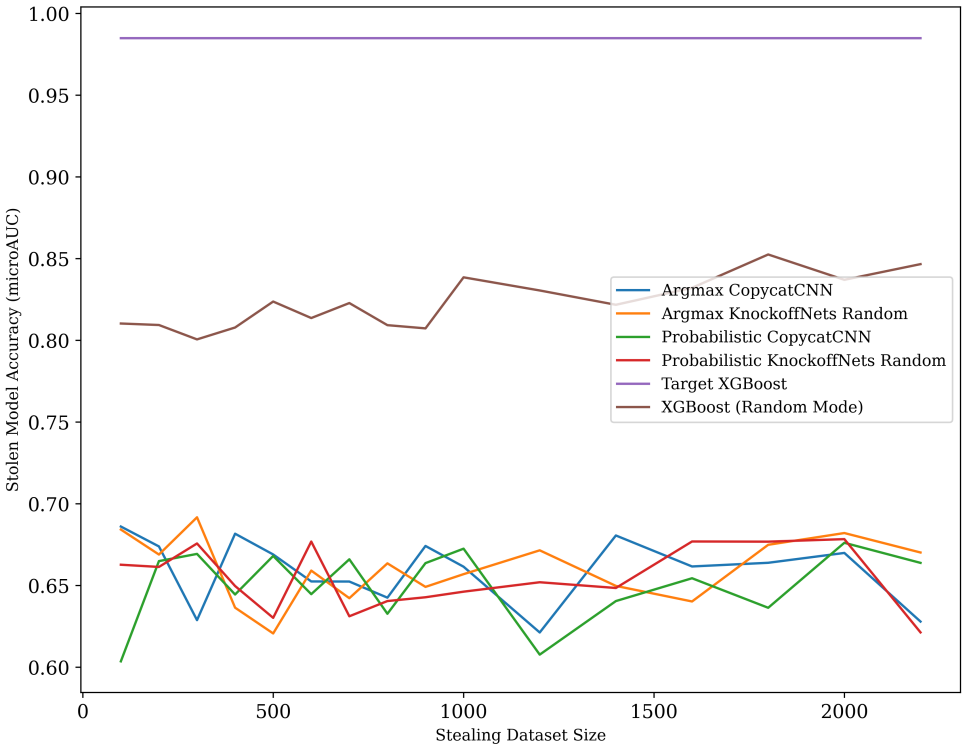


Figure 5.3 Prediction scores(microAUC) of the substitute models in the black-box attack with synthetic data (random mode)

only if the real data is not available since it performs much poorly compared to real data.

5.4.4 Scenario 2: Gray-Box Attack:Data Selection with Encoding Rules

This section describes the attack method that utilizes the encoding rules to improve the success of black box attacks. The black-box attacks explained in Section 5.4.3 assumes that the ML model is hidden and there is no (or negligible amount of) information available about the model. Unlike the conventional ML services, the proposed method implicitly shares the split points of the model. In the production of encoding rules, some dummy points are also added to blind the split point information. Even though adding dummy points makes it harder to compromise the real split points, it does not hide it completely. An attack scenario, named gray-box attack, is designed to test whether sharing encoding rules leads to an information

leakage or not.

There are two main factors behind the lack of effectiveness of the attacks: data scarcity and the limited variety of the data. Usually, the adversary produces synthetic data to overcome the problem of data scarcity. Synthetic data is probably not as rich as the model data because it is assumed to be generated using a small subset of real data, which cannot represent the model data completely. Therefore, increasing the number of queries does not improve the accuracy of the substitute model after a point.

Split points can be interpreted as a way of grouping data points. For instance, if two query data have same encodings it means that they will be labeled with the same class. By this logic, the gray-box attack exclude the query data with same or similar encodings to obtain a more balanced and divergent set. Considering the number of features and the encoding length, the range of possible query encodings can be extremely wide. Due to the wide range of encoding combinations, it is highly unlikely that two queries have the exact encodings. For instance, the IDASH model has more than 2000 features and each feature value can be encoded into 3 different numbers which leads to 3^{2000} possible encodings for a query. Therefore, the queries with high similarity are eliminated rather than only eliminating the ones with the exactly same encodings.

The Hamming distance between two query encodings is used as a metric to detect the similarity rate of the query data. A random query is selected as a reference point. The Hamming distances of the other queries to the reference point are calculated. If the Hamming distance is less than the 30% of the encoded query length the corresponding queries are discarded from the data set.

The hamming distances of all samples in the synthetic data set generated by independent attribute mode was below the 30% limit. Hence, the gray box attack cannot be applied for the synthetic data of the independent mode. The elimination process, which uses the encoding rules, was not performed on the real data as the it was already limited in number. Hence only the synthetic data set of the random mode is eliminated by the gray-box attack.

The attacks applied in Section 5.4.3 are repeated after clearing up the Random mode synthetic data set. The accuracies of the substitute models are shown in Figure 5.4. In this attack, the XGBoost substitute model outperforms the other models as well.

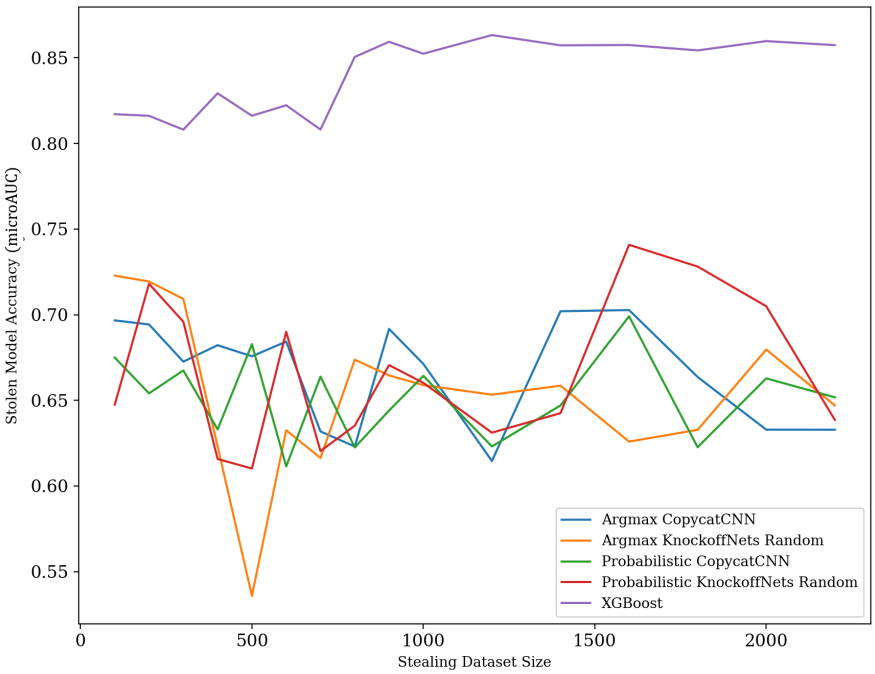


Figure 5.4 Prediction scores of the substitute models in the gray-box attack with synthetic data (random mode)

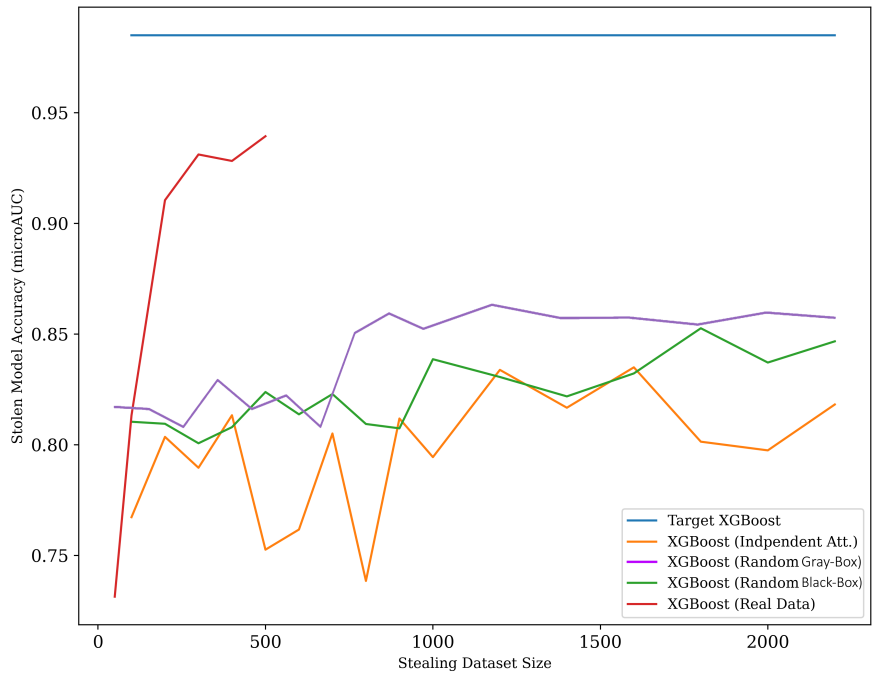


Figure 5.5 The microAUC scores of the best attack models

For a better insight, all the best attack results are gathered in Figure 5.5. In this figure, the microAUC scores of the target model, all tree basic black-box XGBoost models and the gray-box XGBoost model are visualized. As seen in the figure, the gray-box attack has higher microAUC score than those of the black-box attacks. In the case where the improvement is at maximum, it increases the microAUC by 3 points. This shows that the encoding rules can be used to enhance the performance of the existed attacks, albeit marginally.

In summary, eliminating the query data set using the encoding rules improved the existing attacks slightly in our experiments. It should be also considered that there may be other ways to for the attacker exploit the split points, and a further investigation in this direction is needed, which is left for the future work.

5.4.5 Scenario 3: Brute Force Attack using Encodings

If an adversary attacks the model by using only split point information, they need to query all possible feature and encoding combinations in the worst case. The number queries needed can be formulated as

$$QueryNum = {n_E}^{m'}$$

where n_E is the number of digits in the encoding and m' is the total number of features.

The original complexity of the model is usually smaller than this number, however dummy split points added by the proposed method increases the number of queries needed to extract the model. Hence, the brute force attack depends on only the number of features used in the model (m') and the length of the encoding (n_E). Table 5.2 shows the number of queries needed for the selected models to extract the model.

Most of these query numbers are computationally feasible however, such brute force attacks are well-known attacks. Therefore ML services are protected to such attacks on system level. One of the common solution to brute force attacks is limiting the query numbers. Another method is charging the queries for deterrence.

Table 5.2 Number of queries needed in the brute force attack for the selected models

Models	m'	n_E	$QueryNum$
H48, H25	13	4	2^{26}
H2	13	8	2^{39}
H53, H78	13	16	2^{52}
H132	13	32	2^{65}
I13	2064	2	2^{2064}
P3, P10	17	8	2^{51}
P23	18	8	2^{54}
P31	19	16	2^{76}

5.4.6 Comparison of the Scenarios and Discussion

Considering the results of the black-box and gray-box attacks, the most successful attack is the basic black-box attack with real data. We tried to improve the model extraction attacks using encoding rules to find out whether the sharing encoding rules lead to extra information leakage. Only one of the black-box attacks is improved and the improvement rate is 3 microAUC at its maximum. The gray-box attack shows that the encoding rules may improve the model extraction attack in small scale however the overall performance of the attack didn't improve significantly. That being said, the privacy of the model is still an open issue and open to further investigations.

In conclusion, the presented encoding method gives minimal advantage to the adversary. For the black-box and brute force attack scenarios mentioned above which are a natural and inevitable result of machine learning services, service-level protection methods can be applied such as limiting the number of queries, using suspicious behaviour detection systems, and adjusting the query fees accordingly.

6. CONCLUSION AND FUTURE WORK

This thesis presents a privacy-preserving XGBoost inference algorithm based on homomorphic encryption. A new data encoding method for the XGBoost algorithm is proposed that provides a low-cost solution to comparison operations on encrypted data. A tree score calculation method that uses the proposed encoding method is provided to implement the inference algorithm. Then, a privacy-preserving XGBoost inference algorithm is implemented using the proposed encoding and the tree score calculation methods. The algorithm is implemented using the PALISADE HE library. The CKKS homomorphic encryption scheme is preferred because of its ability to work with fixed point numbers.

The performance of the presented work is evaluated on different datasets and with various metrics, including computation time and the precision of the algorithm. The applicability and efficiency of the algorithm are demonstrated.

Privacy of the model is examined and tested under different scenarios. Privacy concerns of both client and model owner are assessed. It is shown that the privacy of the client is preserved with the help of HE. The effect of encoding rules on the privacy of the model is evaluated and it is observed that it some black-box attacks may be improved up to 3 microAUC point. The privacy of the model is open to further investigations since the encoding rules might be used for different type of attacks.

As a future work, the encoding method can be used in the implementation of other tree based machine learning algorithms or inference algorithms that requires comparison operation in general. The proposed method is suitable for CARTs therefore it can be used in ML models consist of CARTs. Additionally, the encoding method can be extended to support other comparisons such as “greater than”, “smaller or equal”.

BIBLIOGRAPHY

Acar, A., Aksu, H., Uluagac, A. S., & Conti, M. (2018). A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, *51*(4).

Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., & Vaikuntanathan, V. (2018). Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada.

Babenko, M., Tchernykh, A., Golimblevskaia, E., Pulido-Gaytan, L. B., & Avetisyan, A. (2020). Homomorphic comparison methods: Technologies, challenges, and opportunities. In *2020 International Conference Engineering and Telecommunication (En&T)*, (pp. 1–5).

Balbás, D. (2021). The hardness of lwe and ring-lwe: A survey. Cryptology ePrint Archive, Paper 2021/1358. `https://eprint.iacr.org/2021/1358`.

Bhambri, S., Muku, S., Tulasi, A., & Buduru, A. B. (2019). A study of black box adversarial attacks in computer vision. *CoRR*, *abs/1912.01667*.

Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2014). (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, *6*(3), 13:1–13:36.

Callahan, A. & Shah, N. H. (2017). Chapter 19 - machine learning in healthcare. In A. Sheikh, K. M. Cresswell, A. Wright, & D. W. Bates (Eds.), *Key Advances in Clinical Informatics* (pp. 279–291). Academic Press.

Chase, M. & Kamara, S. (2010). Structured encryption and controlled disclosure. In Abe, M. (Ed.), *Advances in Cryptology - ASIACRYPT 2010*, (pp. 577–594)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Chen, T. & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, (pp. 785–794).

Cheon, J. H., Hong, S., & Kim, D. (2020). Remark on the security of ckks scheme in practice. Cryptology ePrint Archive, Paper 2020/1581. `https://eprint.iacr.org/2020/1581`.

Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017a). Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, (pp. 409–437). Springer.

Cheon, J. H., Kim, A., Kim, M., & Song, Y. S. (2017b). Homomorphic encryption for arithmetic of approximate numbers. In Takagi, T. & Peyrin, T. (Eds.), *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, (pp. 409–437). Springer.

Cheon, J. H., Kim, D., & Kim, D. (2019). Efficient homomorphic comparison methods with optimal complexity. Cryptology ePrint Archive, Paper 2019/1234. `https://eprint.iacr.org/2019/1234`.

Cheon, J. H., Kim, D., Kim, D., Lee, H. H., & Lee, K. (2019). Numerical method

for comparison on homomorphically encrypted numbers. Cryptology ePrint Archive, Paper 2019/417. https://eprint.iacr.org/2019/417.

Correia-Silva, J. R., Berriel, R. F., Badue, C., de Souza, A. F., & Oliveira-Santos, T. (2018). Copycat CNN: Stealing knowledge by persuading confession with random non-labeled data. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE.

Curtmola, R., Garay, J., Kamara, S., & Ostrovsky, R. (2006). Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, (pp. 79–88)., New York, NY, USA. Association for Computing Machinery.

Dasgupta, D., Akhtar, Z., & Sen, S. (2022). Machine learning in cybersecurity: a comprehensive survey. *The Journal of Defense Modeling and Simulation*, *19*(1), 57–106.

De Cristofaro, E. (2021). A critical overview of privacy in machine learning. *IEEE Security & Privacy*, *19*(4), 19–27.

Dua, D. & Graff, C. (2017). UCI machine learning repository.

Dua, S. & Du, X. (2016). *Data mining and machine learning in cybersecurity*. CRC press.

Gentry, C. (2009). Computing on encrypted data. In Garay, J. A., Miyaji, A., & Otsuka, A. (Eds.), *Cryptology and Network Security, 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings*, volume 5888 of *Lecture Notes in Computer Science*, (pp. 477). Springer.

Goodell, J. W., Kumar, S., Lim, W. M., & Pattnaik, D. (2021). Artificial intelligence and machine learning in finance: Identifying foundations, themes, and research clusters from bibliometric analysis. *Journal of Behavioral and Experimental Finance*, *32*, 100577.

Halevi, S. & Shoup, V. (2014). Algorithms in helib. Cryptology ePrint Archive, Paper 2014/106. https://eprint.iacr.org/2014/106.

Handa, A., Sharma, A., & Shukla, S. K. (2019). Machine learning in cybersecurity: A review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *9*(4), e1306.

Hastie, T., Tibshirani, R., & Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc.

Jayaraman, P. P., Forkan, A. R. M., Morshed, A., Haghighi, P. D., & Kang, Y.-B. (2020). Healthcare 4.0: A review of frontiers in digital health. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *10*(2), e1350.

Kasyap, H. & Tripathy, S. (2021). Privacy-preserving decentralized learning framework for healthcare system. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, *17*(2s), 1–24.

Ku, H., Susilo, W., Zhang, Y., Liu, W., & Zhang, M. (2022). Privacy-preserving federated learning in medical diagnosis with homomorphic re-encryption. *Computer Standards & Interfaces*, *80*, 103583.

Kwabena, O.-A., Qin, Z., Zhuang, T., & Qin, Z. (2019). Mscryptonet: Multi-scheme privacy-preserving deep learning in cloud computing. *IEEE Access*, *7*, 29344–29354.

Lee, E., Lee, J.-W., No, J.-S., & Kim, Y.-S. (2020). Minimax approximation of sign function by composite polynomial for homomorphic comparison. Cryptology

ePrint Archive, Paper 2020/834. https://eprint.iacr.org/2020/834.

Li, B. & Micciancio, D. (2020). On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Paper 2020/1533. https://eprint.iacr.org/2020/1533.

Little, M. A., McSharry, P. E., Roberts, S. J., Costello, D. A., & Moroz, I. M. (2007). Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *BioMedical Engineering OnLine*, *6*(1), 23.

Lyubashevsky, V., Peikert, C., & Regev, O. (2013). On ideal lattices and learning with errors over rings. *J. ACM*, *60*(6).

Ma, J., Naas, S.-A., Sigg, S., & Lyu, X. (2022). Privacy-preserving federated learning based on multi-key homomorphic encryption. *International Journal of Intelligent Systems*, *1*, 1.

Ma, Z., Ma, J., Miao, Y., Liu, X., Choo, K.-K. R., Yang, R., & Wang, X. (2020). Lightweight privacy-preserving medical diagnosis in edge computing. *IEEE Transactions on Services Computing*, *1*, 1.

Meng, X. & Feigenbaum, J. (2020). Privacy-preserving xgboost inference. *CoRR*, *abs/2011.04789*.

Nicolae, M.-I., Sinn, M., Tran, M. N., Buesser, B., Rawat, A., Wistuba, M., Zantedeschi, V., Baracaldo, N., Chen, B., Ludwig, H., Molloy, I. M., & Edwards, B. (2018). Adversarial robustness toolbox v1.0.0.

Orekondy, T., Schiele, B., & Fritz, M. (2018). Knockoff nets: Stealing functionality of black-box models.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In Stern, J. (Ed.), *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, (pp. 223–238). Springer.

Papernot, N., McDaniel, P., & Goodfellow, I. (2016). Transferability in machine learning: from phenomena to black-box attacks using adversarial samples.

Passerat-Palmbach, J., Farnan, T., McCoy, M., Harris, J. D., Manion, S. T., Flannery, H. L., & Gleim, B. (2020). Blockchain-orchestrated machine learning for privacy preserving federated learning in electronic health data. In *2020 IEEE International Conference on Blockchain (Blockchain)*, (pp. 550–555). IEEE.

Pattnayak, P. & Panda, A. R. (2021). *Innovation on Machine Learning in Healthcare Services—An Introduction*, (pp. 1–30). Singapore: Springer Singapore.

Paul, J., Annamalai, M. S. M. S., Ming, W., Al Badawi, A., Veeravalli, B., & Aung, K. M. M. (2021). Privacy-preserving collective learning with homomorphic encryption. *IEEE Access*, *9*, 132084–132096.

Peikert, C. (2009). Public-key cryptosystems from the worst-case shortest vector problem: Extended abstract. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, (pp. 333–342)., New York, NY, USA. Association for Computing Machinery.

Ping, H., Stoyanovich, J., & Howe, B. (2017). Datasynthesizer: Privacy-preserving synthetic datasets. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA. Association for Computing Machinery.

Polyakov, Y., Rohloff, K., & Ryan, G. W. (2017). Palisade lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute*

ofTechnology (NJIT), Tech. Rep, *15*.

Regev, O. (2005). On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, (pp. 84–93)., New York, NY, USA. Association for Computing Machinery.

Renault, T. (2020). Sentiment analysis and machine learning in finance: a comparison of methods and models on one million messages. *Digital Finance*, *2*(1-2), 1–13.

Rivest, R. L. & Dertouzos, M. L. (1978). On data banks and privacy homomorphisms.

SEAL (2022). Microsoft SEAL (release 4.0). `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA.

Sharma, A. & Verbeke, W. J. (2020). Improving diagnosis of depression with xgboost machine learning model and a large biomarkers dutch dataset (n= 11,081). *Frontiers in big Data*, *1*, 15.

Shi, Y., Sagduyu, Y. E., Davaslioglu, K., & Li, J. H. Active deep learning attacks under strict rate limitations for online api calls.

Singh, S., Rathore, S., Alfarraj, O., Tolba, A., & Yoon, B. (2022). A framework for privacy-preservation of iot healthcare data using federated learning and blockchain technology. *Future Generation Computer Systems*, *129*, 380–388.

Wood, A., Najarian, K., & Kahrobaei, D. (2020). Homomorphic encryption for machine learning in medicine and bioinformatics. *ACM Computing Surveys (CSUR)*, *53*(4), 1–35.

Yang, Y., Niehaus, K. E., Walker, T. M., Iqbal, Z., Walker, A. S., Wilson, D. J., Peto, T. E., Crook, D. W., Smith, E. G., Zhu, T., et al. (2018). Machine learning for classifying tuberculosis drug-resistance from dna sequencing data. *Bioinformatics*, *34*(10), 1666–1671.

Zhang, L., Xu, J., Vijayakumar, P., Sharma, P. K., & Ghosh, U. (2022). Homomorphic encryption-based privacy-preserving federated learning in iot-enabled healthcare system. *IEEE Transactions on Network Science and Engineering*, *1*(1), 1–17.

APPENDIX A

Table A.1 Heart Dataset: XGBoost Hyperparameters. Scores are the average of the accuracies of 3-fold cross validation

Model ID	d_{max}	n_T	t_ℓ	Score (Accuracy)
H0	2	24	0.06	0.77083
H1	2	32	0.06	0.78646
H2	2	48	0.06	0.79167
H3	2	64	0.06	0.77084
H4	2	72	0.06	0.77084
H5	2	128	0.06	0.78125
H6	3	24	0.06	0.78125
H7	3	32	0.06	0.77084
H8	3	48	0.06	0.77084
H9	3	64	0.06	0.77084
H10	3	72	0.06	0.77084
H11	3	128	0.06	0.77604
H12	4	24	0.06	0.76562
H13	4	32	0.06	0.75521
H14	4	48	0.06	0.75521
H15	4	64	0.06	0.74479
H16	4	72	0.06	0.75533
H17	4	128	0.06	0.76042
H18	5	24	0.06	0.75521
H19	5	32	0.06	0.76042
H20	5	48	0.06	0.73958
H21	5	64	0.06	0.77604
H22	5	72	0.06	0.78125
H23	5	128	0.06	0.77083
H24	2	24	0.07	0.78646
H25	2	32	0.07	0.78646
H26	2	48	0.07	0.79167
H27	2	64	0.07	0.76562
H28	2	72	0.07	0.74480
H29	2	128	0.07	0.75521
H30	3	24	0.07	0.75521
H31	3	32	0.07	0.75521
H32	3	48	0.07	0.75521
H33	3	64	0.07	0.75521

Table A.1 continued from previous page

Model ID	d_{max}	n_T	t_ℓ	Score (Accuracy)
H34	3	72	0.07	0.74479
H35	3	128	0.07	0.75543
H36	4	24	0.07	0.77604
H37	4	32	0.07	0.76562
H38	4	48	0.07	0.76562
H39	4	64	0.07	0.79687
H40	4	72	0.07	0.78125
H41	4	128	0.07	0.77604
H42	5	24	0.07	0.79167
H43	5	32	0.07	0.78646
H44	5	48	0.07	0.80208
H45	5	64	0.07	0.76562
H46	5	72	0.07	0.76562
H47	5	128	0.07	0.77604
H48	2	24	0.08	0.77084
H49	2	32	0.08	0.77604
H50	2	48	0.08	0.77083
H51	2	64	0.08	0.76042
H52	2	72	0.08	0.76562
H53	2	128	0.08	0.76562
H54	3	24	0.08	0.76562
H55	3	32	0.08	0.77083
H56	3	48	0.08	0.77604
H57	3	64	0.08	0.75515
H58	3	72	0.08	0.75423
H59	3	128	0.08	0.76042
H60	4	24	0.08	0.77604
H61	4	32	0.08	0.77083
H62	4	48	0.08	0.76562
H63	4	64	0.08	0.78646
H64	4	72	0.08	0.79167
H65	4	128	0.08	0.78125
H66	5	24	0.08	0.78646
H67	5	32	0.08	0.79167
H68	5	48	0.08	0.79167
H69	5	64	0.08	0.79167

Table A.1 continued from previous page

Model ID	d_{max}	n_T	t_ℓ	Score (Accuracy)
H70	5	72	0.08	0.77604
H71	5	128	0.08	0.78646
H72	2	24	0.09	0.76562
H73	2	32	0.09	0.76042
H74	2	48	0.09	0.77604
H75	2	64	0.09	0.76562
H76	2	72	0.09	0.76042
H77	2	128	0.09	0.74479
H78	3	24	0.09	0.75521
H79	3	32	0.09	0.76042
H80	3	48	0.09	0.76562
H81	3	64	0.09	0.75523
H82	3	72	0.09	0.74480
H83	3	128	0.09	0.77084
H84	4	24	0.09	0.76562
H85	4	32	0.09	0.77084
H86	4	48	0.09	0.77084
H87	4	64	0.09	0.78646
H88	4	72	0.09	0.78646
H89	4	128	0.09	0.77604
H90	5	24	0.09	0.78125
H91	5	32	0.09	0.78125
H92	5	48	0.09	0.78646
H93	5	64	0.09	0.77604
H94	5	72	0.09	0.77083
H95	5	128	0.09	0.77083
H96	2	24	0.1	0.76562
H97	2	32	0.1	0.77083
H98	2	48	0.1	0.77604
H99	2	64	0.1	0.76042
H100	2	72	0.1	0.76042
H101	2	128	0.1	0.76562
H102	3	24	0.1	0.76042
H103	3	32	0.1	0.77604
H104	3	48	0.1	0.76562
H105	3	64	0.1	0.75521

Table A.1 continued from previous page

Model ID	d_{max}	n_T	t_ℓ	Score (Accuracy)
H106	3	72	0.1	0.75521
H107	3	128	0.1	0.73437
H108	4	24	0.1	0.75521
H109	4	32	0.1	0.75521
H110	4	48	0.1	0.77083
H111	4	64	0.1	0.78125
H112	4	72	0.1	0.78125
H113	4	128	0.1	0.78646
H114	5	24	0.1	0.77083
H115	5	32	0.1	0.77604
H116	5	48	0.1	0.77083
H117	5	64	0.1	0.77604
H118	5	72	0.1	0.78125
H119	5	128	0.1	0.76562
H120	2	24	0.3	0.77604
H121	2	32	0.3	0.75521
H122	2	48	0.3	0.76562
H123	2	64	0.3	0.79167
H124	2	72	0.3	0.77084
H125	2	128	0.3	0.78125
H126	3	24	0.3	0.78125
H127	3	32	0.3	0.78125
H128	3	48	0.3	0.78125
H129	3	64	0.3	0.75521
H130	3	72	0.3	0.75521
H131	3	128	0.3	0.76562
H132	4	24	0.3	0.76562
H133	4	32	0.3	0.76042
H134	4	48	0.3	0.76042
H135	4	64	0.3	0.78125
H136	4	72	0.3	0.78125
H137	4	128	0.3	0.78125
H138	5	24	0.3	0.75521
H139	5	32	0.3	0.75521
H140	5	48	0.3	0.76562
H141	5	64	0.3	0.76562

Table A.1 continued from previous page

Model ID	d_{max}	n_T	t_ℓ	Score (Accuracy)
H142	5	72	0.3	0.76042
H143	5	128	0.3	0.76042

Table A.2 Parkinsons Dataset: XGBoost Hyperparameters. Scores are the average of the accuracies of 3-fold cross validation

Model ID	d_{max}	n_T	t_ℓ	m'	**Score(Accuracy)**
P1	2	24	0.12	14	0.91667
P2	2	32	0.12	16	0.91667
P3	2	48	0.12	17	0.92708
P4	2	64	0.12	17	0.91667
P5	2	72	0.1	17	0.90625
P6	2	72	0.12	18	0.91667
P7	3	24	0.12	17	0.91667
P8	3	24	0.09	16	0.91667
P9	3	32	0.12	17	0.91667
P10	3	48	0.12	17	0.92708
P11	3	64	0.12	18	0.91667
P12	3	64	0.1	17	0.91667
P13	3	72	0.12	18	0.91667
P14	3	72	0.1	17	0.91667
P15	4	24	0.12	19	0.91667
P16	4	24	0.09	18	0.89583
P17	4	32	0.12	19	0.91667
P18	4	48	0.12	19	0.91667
P19	4	48	0.08	19	0.91667
P20	4	64	0.12	19	0.90625
P21	4	72	0.12	19	0.90625
P22	5	24	0.12	19	0.90625
P23	5	24	0.08	18	0.89583
P24	5	32	0.12	19	0.91667
P25	5	48	0.1	19	0.91667
P26	5	48	0.12	19	0.91667
P27	5	64	0.12	19	0.90625
P28	5	72	0.12	19	0.90625
P29	6	24	0.12	19	0.90625
P30	6	24	0.08	18	0.89583
P31	6	32	0.12	19	0.91667
P32	6	48	0.09	19	0.91667
P33	6	48	0.12	19	0.91667
P34	6	64	0.12	19	0.90625
P35	6	72	0.12	19	0.90625

Table A.3 Selected IDASH Models: XGBoost Hyperparameters. Scores are the microAUC values of the 5-fold cross-validation

Model ID	**Feature Description and Additional Preprocessings**	n_T	d_{max}	**Score (microAUC)**
I1	Min-Max Scaling	100	2	0.980
I2	Min-Max Scaling	40	2	0.973
I3	-	100	2	0.981
I4	-	40	2	0.974
I5	feature values = {0,1} where nonzero feature values mapped to 1	100	2	0.956
I6	feature values = {0,1} where CNV nonzero values mapped to 1	40	2	0.948
I7	feature values = {0,1} where CNV -2,2 values mapped to 1	100	2	0.952
I8	feature values = {0,1} where CNV -2,2 values mapped to 1	40	2	0.943
I9	feature values = {0,1} where CNV nonzero values mapped to 1	100	3	0.951
I10	feature values = {0,1} where CNV nonzero values mapped to 1	40	3	0.945
I11	feature values = {0,1} where CNV -2,2 values mapped to 1	100	3	0.951
I12	feature values = {0,1} where CNV -2,2 values mapped to 1	40	3	0.945
I13	**feature values = {-1, 0,1} where CNV -2, -1 are mapped to -1 and CNV 1, 2 are mapped to 1**	**100**	**2**	**0.983**
I14	feature values = {-1, 0,1} where CNV -2, -1 are mapped to -1 and CNV 1, 2 are mapped to 1	40	2	0.973
I15	feature values = {-1, 0,1} where CNV -2 mapped to -1 and CNV 2 mapped to 1. All others -1,0,1 are mapped to 0.	100	2	0.953
I16	feature values = {-1, 0,1} where CNV -2 mapped to -1 and CNV 2 mapped to 1. All other CNV -1,0,1 are mapped to 0.	40	2	0.944