Secure Decision Tree Inference Using Bloom Filters

by

Sean Richard Dimalouw Lalla

Bachelor of Computer Science, UNB, 2022
Bachelor of Science, UNB, 2022

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Rongxing Lu, Ph.D., Faculty of Computer Science
Examining Board:	Hung Cao, Ph.D., Faculty of Computer Science, Chair
	Sajjad Dadkhah, Ph.D., Faculty of Computer Science
	Zhen Lei, Ph.D., Department of Civil Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September 2023

Abstract

Cloud computing allows model providers to distribute machine learning models at scale without purchasing dedicated hardware for model hosting. However, when hosting their models in the cloud, model providers may be forced to disclose private model details. Due to the time and monetary investments associated with model training, model providers may be reluctant to host their models due to these privacy concerns. To combat these issues, several privacy preserving decision tree schemes have been proposed which ensure the privacy of the decision tree models, the client query, and the final classification of the model. However, most existing schemes require significant communication or computational overhead. In this work, we propose a privacy preserving scheme for decision tree inference, which uses Bloom filters to hide the original decision tree structure while maintaining reliable classification results. Our scheme's security and performance are verified through rigorous testing and analysis.

Dedication

I dedicate this thesis to the people who got me where I am today. This is for all the teachers, professors, co-workers, friends, and family members who have supported me throughout my life. Without your support I would not have been able to achieve all that I have.

Acknowledgements

To my supervisor Dr. Rongxing Lu, I express my deepest gratitude. His guidance, expertise, and encouragement have gotten me to where I am today. Dr. Lu was always willing to make time to discuss ideas with me, inform me of new opportunities, and just chat about life. I am thankful for both his mentorship, and his friendship throughout my university career.

To my family, thank you for your never ending support. Thank you for cheering me on during tough weeks and listening to me talk about papers, code, and conferences.

Table of Contents

Abstract **ii**

Dedication **iii**

Acknowledgments **iv**

Table of Contents **v**

List of Tables **ix**

List of Figures **x**

Abbreviations **xiii**

1 Introduction **1**

1.1 Cloud Computing 2

1.2 Outsourcing Decision Tree Models 4

1.3 Contributions 6

1.4 Organization 7

2 Related Work **8**

2.1 Secure Machine Learning 8

2.2 Bloom Filter Use in Machine Learning 10

2.3 Privacy Preserving Decision Trees 12

3 Models and Design Goals **15**

3.1 System Model 15
3.2 Security Model 17
3.3 Design Goals 17
3.3.1 Privacy Preservation 18
3.3.2 Efficiency 18

4 Preliminaries **20**
4.1 Machine Learning 20
4.1.1 Supervised Learning 22
4.1.1.1 Classification Models 22
4.1.1.2 Regression Models 23
4.1.2 Unsupervised Learning 24
4.1.3 Reinforcement Learning 25
4.2 Cybersecurity 26
4.2.1 Encryption 26
4.2.1.1 Symmetric Key Encryption 27
4.2.1.2 Public Key Encryption 27
4.2.1.3 Homomorphic Encryption 28
4.2.2 Hash Functions 30
4.3 Data Structures 31
4.3.1 Decision Tree 31
4.3.1.1 Full Binary Tree 32
4.3.2 Bloom Filters 33
4.3.3 Heap 34

5 Proposed Scheme **36**
5.1 Model Preparation 36
5.2 Query Request and Response 45

5.3 Response Recovery 48

6 Security Analysis **50**

6.1 Privacy-Preserving Decision Tree Model 50

6.2 Privacy-Preserving Client Query 51

6.3 Privacy-Preserving Client Query Result 51

6.4 Privacy Preserving Threshold Values 52

6.5 Prevention of Collusion Attacks 52

7 Experimental Analysis **54**

7.1 Experimental Settings 54

7.2 Model Preparation 55

7.2.1 Decision Tree Conversion 55

7.2.2 Key Generation 59

7.2.3 Shuffling Bloom Filters 59

7.3 Query Request and Response 61

7.3.1 Query Encryption 61

7.3.2 $Q_{X,i}$ Creation 61

7.3.3 Classification by Cloud Servers 62

7.4 Response Recovery 63

7.5 Performance Summaries 64

8 Conclusion and Future Work **67**

8.1 Conclusion 67

8.2 Future Work 68

8.2.1 Machine Learning Model Attacks 68

8.2.2 Allowing Float Values 69

8.2.3 Key Distribution 69

8.2.4 Model Explainability 70

8.2.5 Extensions to Non-Full Trees 70

8.2.6 Extensions to Other Tree Based Models 70

Bibliography **77**

A Code **78**

A.1 Bloom Tree Module 78

Vita

List of Tables

2.1 Comparison of Privacy Preserving Decision Tree Schemes 14

4.1 Differences Between Homomorphic Encryption Schemes 29

7.1 Slate's Letter Recognition Dataset Features 55

7.2 Time (s) to perform scheme steps for various sized decision trees . . . 64

7.3 Classification Times (s) 66

List of Figures

1.1 Model Outsourcing Scheme 2

1.2 Azure Services Diagram [22] 3

2.1 The Left Image Shows the Original Learned Bloom Filter Framework Proposed in [15]. The Right Side Shows the Sandwiched Framework [24] 11

3.1 System model under consideration, which splits the original DT model into two Bloom filter-based DT models distributed to two cloud servers. The client sends an encrypted query request EQ to two servers, after receiving two bit strings S_1 and S_2, the client can recover the query result $S = S'_1 \oplus S'_2$ and get to know the query result. 16

4.1 The Three Main Branches of Machine Learning with Descriptions [28] 21

4.2 An example of a classification model which predicts whether a student will be accepted into UNB or not 23

4.3 An example of a regression model which predicts a UNB student's term GPA 24

4.4 An example of an unsupervised machine learning model which clusters anomalous web traffic 25

4.5 The Reinforcement Learning Pipeline [20] 26

4.6 Symmetric Key Encryption 27

4.7 Public Key Encryption 28

4.8 Hash Function 30
4.9 An Example Decision Tree 32
4.10 An Example Binary Tree and a Full Binary Tree 33
4.11 An example of Bloom filter with $k = 3$, where $x, y \in \mathcal{U}$, $w \notin \mathcal{U}$, but the Bloom filter falsely reports $w \in \mathcal{U}$. 34
4.12 An example of heap-based path decision, where the path is 0-1-4. . . 35

5.1 An illustrative example of our proposed scheme 37
5.2 Vector Extraction from Decision Tree Models 38
5.3 Node Domain Splitting 39
5.4 An Example Demonstrating fv_j in A_j and $checkMembership(BF_{1,j}, fv_j)$ and $checkMembership(BF_{2,j})$ Evaluate to True 41
5.5 An Example Demonstrating fv_j in A_j and $checkMembership(BF_{1,j}, fv_j)$ and $checkMembership(BF_{2,j})$ Evaluate to False 41
5.6 An Example Demonstrating fv_j in B_j and $checkMembership(BF_{1,j}, fv_j)$ Evaluates to True and $checkMembership(BF_{2,j})$ Evaluates to False . 42
5.7 An Example Demonstrating fv_j in B_j and $checkMembership(BF_{1,j}, fv_j)$ Evaluates to False and $checkMembership(BF_{2,j})$ Evaluates to True . 42

7.1 Number of Decision Nodes Versus Time to Convert Tree to Bloom Filters (s) 56
7.2 Decision Tree Depth Versus Time to Convert Tree to Bloom Filters (s) 57
7.3 Number of Leaf Nodes Versus Time to Create Leaf Node Bloom Filters (s) 58
7.4 Decision Tree Depth Versus Time to Create Leaf Node Bloom Filters (s) 58
7.5 Number of Decision Nodes Versus Time to Generate Cryptographic Keys 60

7.6 Number of Decision Nodes Versus Time to Shuffle Bloom Filters . . . 60
7.7 Number of Decision Nodes Versus Time to Create $Q_{X,i}$ (s) 62
7.8 Number of Decision Nodes Versus Time to Create S_i (s) 63
7.9 Number of Decision Nodes Versus Time to Perform Response Recovery (s) 65

List of Symbols, Nomenclature or Abbreviations

DT	Decision Tree
ML	Machine Learning
AES	Advanced Encryption Standard
DES	Data Encryption Standard
$3DES$	Triple Data Encryption Standard
$ECDSA$	Elliptic Curve Digital Signature Algorithm
RSA	RivestShamirAdleman
PHE	Partially Homomorphic Encryption
SHE	Somewhat Homomorphic Encryption
FHE	Fully Homomorphic Encryption
$SMPC$	Secure Multi-Party Communication
CNN	Convolutional Neural Network
GPU	Graphical Processing Unit
CPU	Central Processing Unit
URL	Uniform Resource Locator

Chapter 1

Introduction

Data is being produced at an unprecedented rate. With over 10 exabytes of data being produced every hour [31], the amount of data that is readily available is constantly increasing. As the amount of available data increases in the world, so too does the available insights that can be extracted from that data. As such, many reasearchers, companies, and governments have turned to using machine learning models to discover patterns and insights about their data which may allow them to make predictions on previously unseen data. Amongst these predictive models, one of the most popular machine learning models is decision tree based models [39].

Though predictive models are beneficial, producing them can be time consuming and expensive, especially since models may have hundreds of parameters to tune. Furthermore, data storage can be expensive as it may require databases to be stored on premises. Cloud computing has alleviated some of this stress since its model of providing computing resources on demand allows users to request computing resources only as needed, and scale up and down as requirements change. Due to these benefits, many model providers have adopted cloud computing for training machine learning models and model outsourcing.

In a model outsourcing scenario, the model provider submits their machine learning

model to a cloud server. The cloud server can then host the model and provide inference services to potential clients. Clients can then submit queries to the cloud server and receive their classification results from the cloud server. Thus, by hosting their machine learning models in the cloud, model providers are able to provide their model to a larger audience. An example of a model outsourcing scheme can be seen in Figure 1.1.

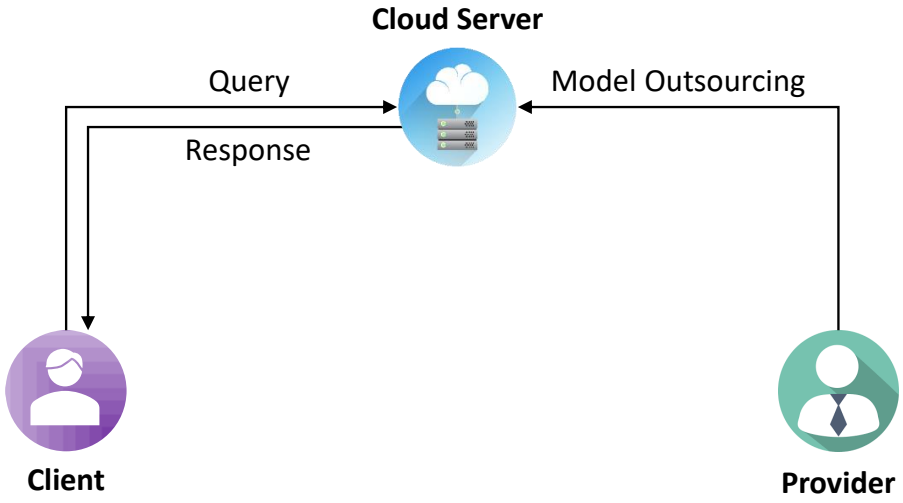

Figure 1.1: Model Outsourcing Scheme

However, the use of cloud computing for model outsourcing also opens up machine learning models to numerous threats. In this chapter, we discuss an overview of cloud computing, especially as it relates to machine learning model hosting, and outsourcing decision tree models, before we discuss the contributions of this thesis and present the organization of this paper.

1.1 Cloud Computing

Cloud computing is a paradigm which seeks to allow clients to be able to request computing resources on demand. Cloud computing allows users to store data, host applications, and perform resource intensive computing without the need for purchasing specialized hardware. Furthermore, as requirements change, clients can eas-

ily request more computing power or scale back resources as needed. Thus, for particularly busy times of the year clients can request more computing resources while during slow times of the year they can reduce them.

Cloud computing has become especially important today in the world of machine learning as it is much easier to train models in the cloud using data stored in the cloud than training models locally using data stored in a server. This is primarily because machine learning requires large amounts of computing and data storage resources but does not necessarily need them all the time. Furthermore, cloud computing allows for providers to host their models in the cloud without the need for dedicated hardware for model hosting. A diagram of some of the available cloud services available in Microsoft Azure's cloud services is shown in Figure 1.2. Similar offerings are also made available by both Google Cloud, and Amazon Web Services.

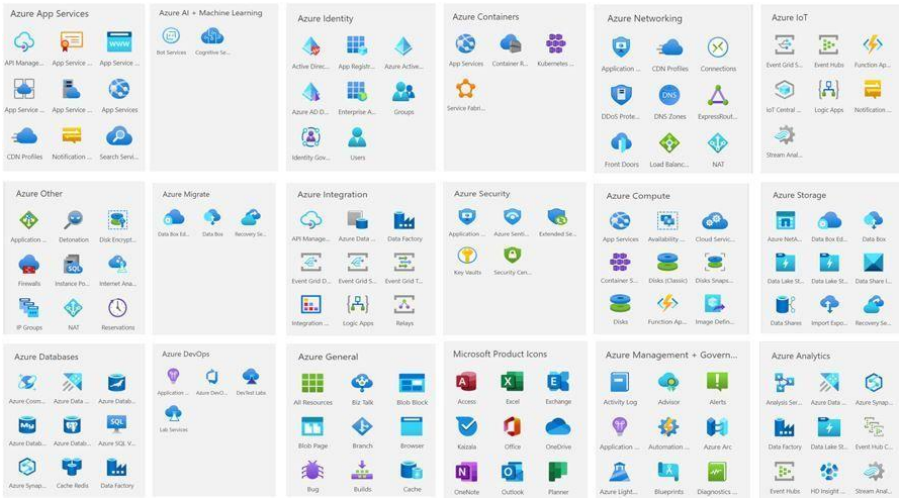

Figure 1.2: Azure Services Diagram [22]

Microsoft lists several key benefits of the cloud paradigm over traditional computing practices [23]. Namely they tout cost benefits, speed increases, global scalability, increased productivity, increased performance, increased reliability, and increased security due to the use of cloud computing [23]. While Microsoft Azure, Amazon Web Services, and Google Cloud all proclaim that security is one of their number

one concerns when it comes to cloud computing, we must consider that they may also be looking to gather data about how their platforms are being used. As such, many papers [39, 40], consider cloud servers to be semi-honest.

However, since machine learning models are often trained on sensitive data including healthcare information, private network information, and even financial history, it is of the utmost concern that algorithms are available which can protect the privacy of machine learning models and their user queries.

1.2 Outsourcing Decision Tree Models

Decision tree based machine learning models are versatile and can be applied to many domains. They can predict cybersecurity attacks [37], provide health-care diagnoses [17, 38], and have even been used for predicting which demographic information is important for online learning efficacy [29]. Decision trees are often chosen for machine learning tasks as they can provide quick and accurate classifications, they are easily understood, and decision trees serve as the basis for many other machine learning models such as AdaBoost and Random Forest models. Furthermore, decision tree models can be trained much quicker than larger and more powerful models such as Artifical Neural Networks. Due to the quick training time of decision tree models, model providers can spend less time producing more accurate models as hypertuning would take a shorter amount of time than hypertuning a more complex model.

Once decision tree models have been trained, model providers may choose to share their model with a wider audience. However, many model providers may not be able to host models on their own as they may not have the hardware required to host models easily. As such, model providers may deploy their models to cloud servers as they allow for multiple copies of the model to be available as needed without the need for providers to purchase dedicated hosting hardware. Furthermore,

cloud providers can easily allow model providers to increase the performance of their machine learning models by giving servers additional computing power at the click of a button.

While cloud computing does provide many benefits for model providers and clients wishing to use of proprietary machine learning models, it also poses several security risks. Uploading a machine learning model to the cloud normally means that sensitive model details such as the features used in the decision tree and the order in which they are used, would be disclosed to the cloud server. As such, a malicious cloud service would be able to easily copy a machine learning model and host it elsewhere. Due to the expensive nature of training machine learning models which includes costs such as data preparation, staff training, and compute costs, model providers may be hesistant to offer their proprietary models in a way that they can easily be copied and hosted elsewhere. Furthermore, clients may be hesitant to use a model if either the cloud server, or the model provider learns anything about their query or classification result. As such, model and query privacy are of grave concern for both providers and clients hoping to make use of such hosted models.

Today, several schemes have been proposed for preserving decision tree model privacy [1, 18, 19, 32, 39, 40]. Namely researchers have focused on preserving the privacy of the decision tree model, as well as the client's query and classification result. While these schemes are able to preserve the privacy of secret information in the decision tree classification process they face compromises in communication and computational efficiency. Today, these schemes largely consider heavy cryptographic protocols such as fully homomorphic encryption which has a high computation overhead [1, 19], or additive secret sharing [39, 40] which has a high communication overhead, which increase the amount of time and resources needed for hosting machine learning models.

1.3 Contributions

To tackle the limitations of the above schemes, we design a privacy-preserving decision tree inference scheme that can protect the privacy of the provider's decision tree, the client's query, and the client's final classification result, while only using simple symmetric encryption and Bloom filters to reduce the communication and processing overhead. Our scheme elminates the need for cloud servers to interact with each other, and allows the client and model provider to disconnect from the cloud servers after submitting their required information. Specifically, this thesis has the following contributions:

• First, we devise a scheme which transforms each node in the original decision tree into two Bloom filters that mimic the function of checking whether a particular feature from a user query is greater than a threshold value associated with that node. Once each of the nodes in the tree is transformed into Bloom filters, they are distributed to two cloud servers. By splitting the decision tree up in this way we prevent the cloud servers from being able to provide classification on their own. As such, neither cloud provider would be able to redistribute the properietary machine learning model. We additionally shuffle the way that each cloud server processes nodes in their modified decision tree. Thus, the output from each cloud server must be individually unshuffled before it can be useful to a client. With this addition, even if a single cloud server's output is leaked by the cloud server, an adversary is unable to retrieve any useful insights without also colluding with the client. Furthermore, this splitting up of the original tree ensures that the client's final classification can only be recovered when the client has received an output from both of the cloud servers.

• Second, we analyze the security of our scheme and show that it is infeasible for the cloud servers to reconstruct the decision tree model or know the classification result of a client query even if the servers collude with one another. Furthermore, we show

that the client learns nothing about the decision tree structure other than the depth of the tree.

• Finally, we analyze our scheme experimentally by calulating the amount of time needed to complete each of the steps of our scheme for various sized decision trees. We further verify our scheme's efficiency by computing the asymptotic complexity needed for different steps in our scheme. We compare our scheme's classification time with the original decision tree model and show that our model still provides fast classification while ensuring model privacy.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 discusses related works in the area of secure machine learning, machine learning augmentation with Bloom filters, and privacy preserving decision tree schemes. In Chapter 3, we discuss our system model and design goals. We discuss preliminaries such as the data structures and techniques used in this thesis in Chapter 4. In Chapter 5 we detail our privacy preserving decision tree scheme. In Chapters 6 and 7, we analyze our scheme's security properites, and demonstrate its efficiency experimentally. Finally we wrap up with our conclusion and discuss future work that could be explored in Chapter 8.

Chapter 2

Related Work

In this chapter we discuss related work pertaining to secure machine learning training and inference, augmentation of machine learning with Bloom filters, and privacy preserving decision tree schemes.

2.1 Secure Machine Learning

Ensuring model privacy is a key area of research today as model providers may wish to keep their models private while offering them to be used by potential clients. Similarly, clients do not want to expose their private information to the provider or cloud servers. As such, it is no surprise that many researchers have developed schemes for ensuring model privacy. Researchers have focused on three main areas of privacy preserving machine learning: preserving dataset privacy, preserving model privacy, and preserving client query privacy [7, 10, 25, 33, 35]. In this section we review works which propose schemes for secure machine learning.

In SecureML [25], Mohassel et al. devise a scheme which enables privacy preserving properties for logistic regression, linear regression, and neural network training. The authors consider a two server model in this work which is consitent with many other privacy-preserving machine learning schemes. In their work, the authors note

that one of the main performance limiting aspects of privacy preserving machine learning algorithms is shared arithmetic oprations on fixed point numbers as these operations must be carried out with garbled circuits or something similar to preserve privacy [25]. They note that while secure addition is easy to perform effeciently, secure multiplication is significantly more complex. Thus, their privacy preserving machine learning algorithm hinges mostly on providing a quicker way to provide secure multiplication. In their scheme they represent shared decimal numbers as shared integers in a finite field and perform a multiplication on shared integers using offline-generated multiplication triplets [25]. Each party then truncates its share of the product so that a fixed number of bits represent the fractional part. Using this strategy they are able to show that they can reconstruct the product from the truncated shares with at most 1 bit off in the least significant position while increasing the performance of secure multiplication [25].

Several other works propose using homomorphic encryption for model privacy. Fang et al. propose a federated machine learning scheme which uses homomorphic encryption to ensure model privacy [10]. Their testing showed that their algorithm could achieve similar levels of accuracy (within 1% of the original model) as normal federated learning but required each of the clients in the federated system to perform encryption and decryption using Paillier homomorphic encryption. Yao et al. instead [35], use partial-homomorphic encryption to preserve the privacy of neural networks in an e-health system. They note that by using partial-homomorphic encryption they can reduce the overhead required to perform crytographic operations when compared to a system using fully-homomorphic encryption.

Another area of interest in secure machine learning algorithms is Secure Multi-Party Communication (SMPC). In [33] researchers make use of the GPU to provide secure training and inference using CNN's (Convolutional Neural Networks). Their work makes use of SMPC to ensure privacy, and GPU acceleration allows the scheme to

perform crytographic functions quicker than running purely on CPU. GenoPPML, a framework for genomic privacy-preserving machine learning on the other hand combines both SMPC and homomorphic encryption to create logistic regression models trained on genomic data from several private data sources [7]. The authors note that SMPC and homomorphic encryption allow for an increased security model, however they require significant computational and communication overhead.

While the above schemes are successful in preserving the privacy of various machine learning models, many of them are used for encrypting more complex machine learning models such as neural networks which require significant time resources for training, require large amounts of data, and aren't as easily understood. Furthermore, many of the schemes above make use of some sort of homomorphic encryption which requires significant computing overhead when compared to simple symmetric encryption, or SMPC which requires significant communication overhead between various parties.

2.2 Bloom Filter Use in Machine Learning

Bloom filters are probabilistic space saving data structures which can perform tests of set membership. Due to their efficient data storing properties and their ability to perform set membership tests many researchers have proposed using Bloom filters in conjunction with machine learning. By adding Bloom filters into their machine learning frameworks and algorithms, researchers are able to gain the benefits of using Bloom filters.

In [15] authors Kraska et al. make a case for producing machine learning models to augment or replicate the function of existing index structures such as B-Trees and Bloom filters. In their work, they consider that Bloom filters are a binary classifier which can sort objects into one of two classes [15]. As such, they draw parallels

between Bloom filters and machine learning classification models. In their work, they propose creating a "Learned Bloom filter" using a CNN or RNN (Recurrent Neural Network) which can predict whether a query X is in a set U or not [15]. In their results they show that their Learned Bloom filter can have smaller space requirements while keeping the false positive rate low. Unlike a traditional Bloom filter their "Learned Bloom filter" model allows for false negatives. However, to mitigate this, they create a overflow Bloom filter which is only checked if the original model's prediction accuracy falls below a specific threshold for a given query [15].

In [24], researchers improve upon the Learned Bloom filter framework by sandwiching the Learned Bloom filter between a prefilter that can remove some initial queries X which are not in U. If the query isn't filtered out by the prefilter, then the scheme continues through the original scheme proposed by Kraska et al. [15]. Sandwiching the Learned Bloom filter in this way can remove false positives up front, then perform classification and then reduce false negatives at the end with the use of the backup filter [24]. Their scheme can be seen in Figure 2.1.

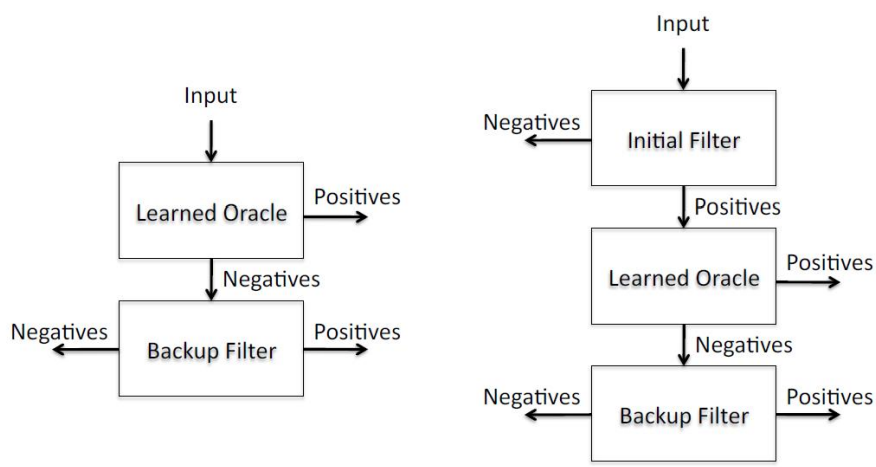

Figure 2.1: The Left Image Shows the Original Learned Bloom Filter Framework Proposed in [15]. The Right Side Shows the Sandwiched Framework [24]

.

In [27], authors make use of the Learned Bloom filter framework described in [15] as well as evolutionary deep learning to create machine learning classifiers which can

predict whether a URL is malicious or benign. Their approach was able to classify URLs with 99.97% accuracy.

AdaBF [9] also seeks to improve upon the Learned Bloom filter framework by instead breaking the set U into g distinct groups instead of two groups as in [15]. Here each group j uses K_j hash functions to test set membership of each of the subsets of U instead. Authors note that their framework requires less space than both [24] and [15]. Authors confirm their scheme's efficiency and usability by using it in various expierments including determining whether a URL is malicious or benign.

Other researchers have proposed different ways of using Bloom filters to augment machine learning. Google's Superbloom [3] algorithm combines Bloom filters with Transformers to perform natural language processing tasks. In their work the authors use Bloom filters and multiple hashing to help represent a large number of categorical values such as product and video ids [3]. Hashing these ids effectively reduces the vocabulary size needed for a natural language processing model [3]. However, since hashing values may result in collisions, the authors make use of Transformers to disambiguate word meanings when collisions occur [3]. Asadi et al. [4] propose a two-phase document ranking system which uses Bloom filters for initial candidate generation, and machine learning models to refine the results from the Bloom filters. While Bloom filters have been used in various machine learning frameworks, to our knowledge, our work is the first attempt at using Bloom filters to perform privacy preserving decision tree inference.

2.3 Privacy Preserving Decision Trees

As decision tree models are some of the most popular machine learning models, it is natural that offering secure model inference and hosting of decision tree models has been a key area of focus in the secure machine learning space. In regards to privacy

preserving decision tree algorithms, there are many aspects which the model provider and the client may wish to keep secret. The features used by the decision tree, the dataset the tree was trained on, the client's query, and the structure of the decision tree itself are all key aspects that a privacy preserving decision tree algorithm may wish to keep secret.

In [18] authors Liao et al. use partially homomorphic encryption to secure decision tree models. In their improved framework [19], the authors opt to use fully homomorphic encryption(FHE) instead. Their new scheme is not only able to work with Random Forest and XGBoost models but can also work for decision tree models such as CART, ID3, and C4.5.

In [32] authors Tai et al. devise a privacy preserving decision tree scheme. They transform the decision tree into a series of linear functions to represent the path cost to each leaf node in the decision tree. By using linear functions to represent the path cost to each leaf node, the authors are able to eliminate the need to use FHE since they only require addition operations under this scheme. Their scheme further is able to allow for only 4 communication steps between the client and the server [32]. Unlike the schemes proposed in [18, 19, 32], our scheme only relies on simple symmetric encryption. Since operations on homomorphically encrypted data take longer to complete [18, 19], using symmetric encryption allows us to reduce the amount of time needed to perform operations such as multiplication and addition when compared to the schemes in [18, 19, 32].

The works which are most similar to our work also involve schemes for secure decision tree inference. We briefly review two works [39, 40], which are closely related to our proposed scheme. In [39], Zheng et al. propose a lightweight scheme to ensure decision tree privacy. Similar to our scheme, Zheng et al. make use of a two-server model. However, their scheme make use of additive secret sharing for secretly distributing the decision tree, whereby each of the elements in the feature vector, each

threshold value, the mapping of features used by the decision tree, and the value at each leaf node is secret shared using additive secret sharing [39]. Classification is then carried out by the two servers over encrypted feature values and final classification is provided to the client.

In their improved scheme in [40], they are able to increase the performance of their previous scheme by changing the binary matrix used for determining the mapping of feature vectors to the order in which features are used in the tree from a $O(X * N)$ sized matrix to an indexing vector of size $O(X)$, where X represents the number of decision nodes in the tree, and N represents the number of features used by the decision tree.

Our scheme primarily differs from these works in how we preserve model privacy using Bloom filters instead of additive secret sharing. By doing so, our scheme is able to still preserve the privacy of a decision tree while having the increased performance benefits of Bloom filter set membership testing over additive secret sharing and multiplicative secret sharing.

Table 2.1: Comparison of Privacy Preserving Decision Tree Schemes

Work	**Method for Preserving DT Privacy**
[18]	Partial Homomorphic Encryption
[19]	Fully Homomorphic Encryption
[32]	Linear Functions and Partial Homomorphic Encryption
[39]	Additive Secret Sharing
[40]	Additive Secret Sharing
Our Scheme	Bloom Filters and Symmetric Encryption

Chapter 3

Models and Design Goals

In this chapter, we describe our system model, security model, and identify our design goals.

3.1 System Model

Similar to recent works [39, 40], our system model, as shown in Figure 3.1, considers a two-server model consisting of the following three types of entities:

• **Model Provider**: The model provider first trains the initial decision tree (DT) model as normal. Using our scheme (described in detail in later sections) the provider then converts their model into a Bloom filter-based DT model which is outsourced to the cloud servers C_1 and C_2. The provider is also responsible for generating cryptographic keys which are distributed to the cloud servers and the client which enable encryption of the client's query, and the client's classification result. Once the provider has uploaded its modified DT model to the cloud servers and distributed cryptographic keys to the correct parties, it no longer needs to interact with the cloud servers or the client.

• **Cloud Servers**: Two cloud servers C_1 and C_2 are employed in our system model. Each server hosts a Bloom filter-based DT model that is distributed to them by the

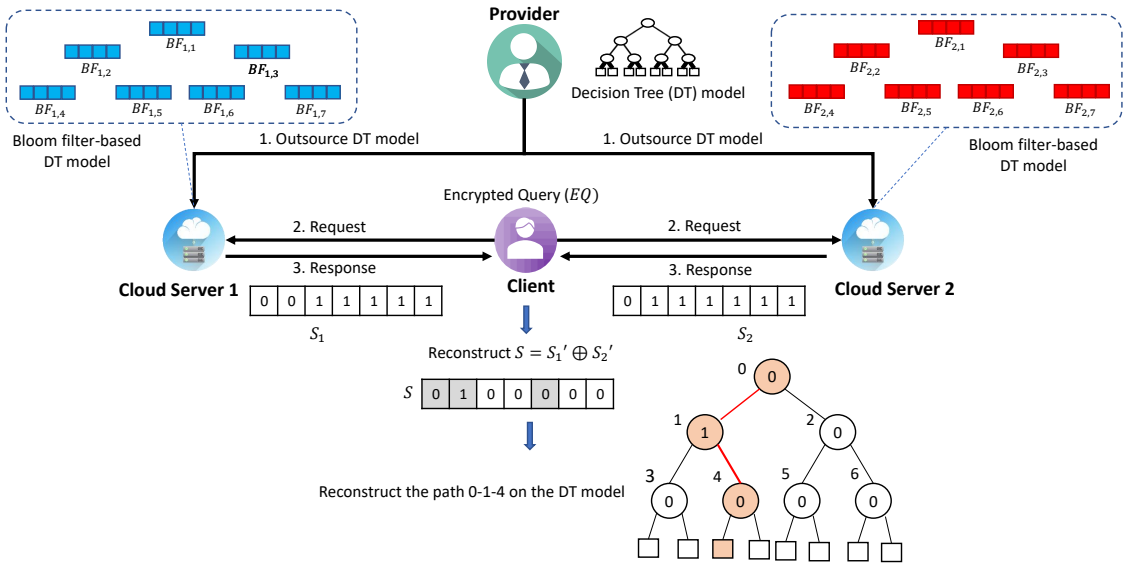

Figure 3.1: System model under consideration, which splits the original DT model into two Bloom filter-based DT models distributed to two cloud servers. The client sends an encrypted query request EQ to two servers, after receiving two bit strings S_1 and S_2, the client can recover the query result $S = S_1' \oplus S_2'$ and get to know the query result.

provider. Individually each of the Bloom filter-based DT models hosted by the cloud servers are useless to the cloud servers, however, when they are combined properly (as discussed in later sections), they can mimic the original DT model's ability to provide classifications. After receiving the query from the client, each server uses the Bloom filter-based DT model to return a bit string as its classification result to the client.

• **Client**: The client wishes to make use of the machine learning model produced by the provider. The client will encrypt a feature vector using a key provided by the model provider and then submits the encrypted feature vector to the two cloud servers. After submitting their encrypted feature vector, the client is able to disconnect from the cloud servers and wait for a result to be returned from them. Once the models have completed classification using their Bloom filter-based DT model, the client receives two bit strings. By combining the two bit strings from the two cloud servers, the client can reconstruct the final classification result of their query.

3.2 Security Model

In our security model, we consider both servers C_1 and C_2 are semi-honest, i.e., while the cloud servers behave honestly, they may try to gleam some information about the queries or the machine learning model. As such, we must take precaution to ensure that the cloud servers cannot learn any information about the model they are hosting, the queries clients submit to them, and the final prediction provided to clients.

In [39, 40] the authors consider that the two cloud servers are hosted by two different cloud providers. In this scenario, they can safely assume that the cloud servers will not collude with each other since doing so would harm the respective cloud provider's reputation. In our scheme however, we do not have this limitation since we have security measures in place such that even if the cloud providers act maliciously and collude with one another, they are unable to gather any useful information.

We assume that the client and the model provider behave honestly in our scheme. As such, they will both carry out our scheme as written and will not try to gleam information not privy to them.

Since we aim to achieve privacy preservation of the provider's model and the client's query in this work, other attacks such as model extraction attacks are beyond the scope of this work and will be explored in our future work.

3.3 Design Goals

Our goal in this work is to propose an efficient privacy-preserving query scheme over decision tree (DT) model, which should satisfy the following two requirements:

3.3.1 Privacy Preservation

In our proposed scheme, the DT machine learning model and the user query should be protected. This means that at no point should the cloud servers or client gain insight into the model's sensitive information, and that the cloud servers and provider should gain no insight into the client's query. By preserving the privacy of the provider's model we ensure that model providers are able to share their proprietary machine learning models without the risk of having it redistributed without their permission, thus making cloud adoption and model sharing more likely for model providers. Similarly protection of the client's query and classification result increase the likelihood that a client will use a cloud hosted machine learning model.

3.3.2 Efficiency

While achieving the requirement of privacy preservation, our proposed scheme should also be efficient. This means that we should still be able to provide model inference in a quick and efficient manner. Furthermore, the client and the provider should not need to remain connected to the cloud servers at all times since this defeats the purpose of using cloud services for offloading machine learning model inferfence.

A decision tree is an efficient machine learning classifier with the ability to provide classification in $O(log(N))$ for a decision tree with N nodes. As such, we do not wish to degrade the performance of the decision tree too much by adding privacy preserving techniques. Other machine learning models have significantly longer classification times so maintaining the model efficiency of a decision tree is key to our scheme.

Furthermore, since we are using cloud computing for the processing of the decision tree, it would also be beneficial to consider that parallization is also a benefit that cloud computing can offer. As such, if there is an opportunity for parallelization in our scheme this would also be seen as beneficial since our scheme makes use of

two cloud servers so being able to carry out tasks in parallel can increase the overall performance of the scheme.

Chapter 4

Preliminaries

4.1 Machine Learning

In 2022, over 97 zettabytes of data were created worldwide and by 2025 analysts expect over 181 zettabytes of data will be created every year [31]. While storage of this data is important, extraction of patterns and creation of models to predict trends in the data is often the intended purpose for storing data.

Machine learning is a subdiscipline within the field of artificial intelligence which focuses on creating models that can recognize or discover patterns in data and make predictions about previously unseen data [6]. Various advantages of using machine learning models include that they can uncover patterns which a human might not see, they can deal with large amounts of data that would be otherwise hard to analyze, models can operate without human intervention, and model results may improve as more data becomes available [2]. Disadvantages of using machine learning include that it is computationally and financially expensive to train machine learning models, machine learning models can be difficult to understand [2], and machine learning models are subject to bias [21] and dual use issues [26].

Machine learning models have been used for many tasks ranging from healthcare

diagnoses [17, 38] to network intrusion detection [37]. Industrial use of machine learning models has been seen in many industries such as banking for predicting credit card risk, HR departments for hiring processes, and quality control of manufacturing. However, public interest in machine learning has been rising due to large machine learning models such as Chat-GPT, a conversational chatbot, and Dall-E, a machine learning model used for generating images from text prompts, which have become fixtures of pop culture. Both of these models have been trained on the large amount of data available to them.

We broadly consider there to be 3 main types of machine learning as shown in Figure 4.1:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

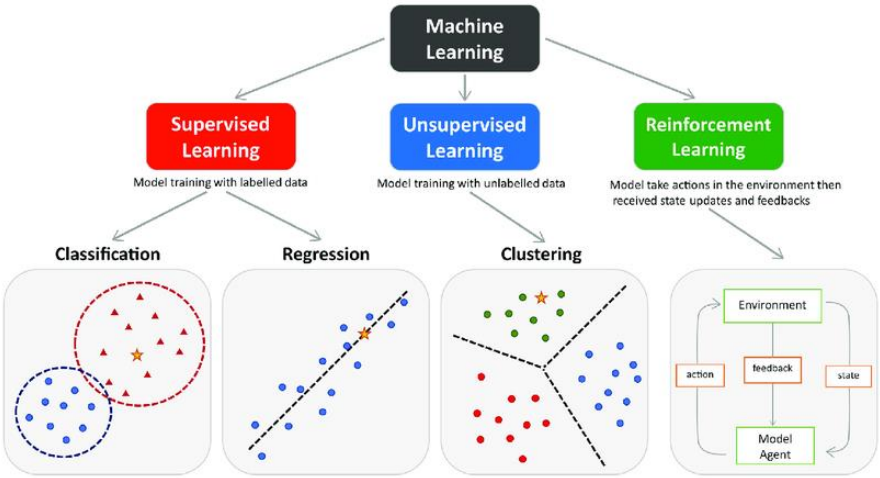

Figure 4.1: The Three Main Branches of Machine Learning with Descriptions [28]

4.1.1 Supervised Learning

Supervised machine learning is the most common form of machine learning [6]. There are two main phases to the supervised learning. In the first step, model training, a machine learning model M is trained on a dataset D with labelled elements. Using some machine learning algorithm, a model is created to predict the labels of elements in this dataset by using the other features of the dataset D. Once a model has been produced, the second phase, model prediction can occur. Here users can submit queries to the model to receive a prediction. A good supervised learning model should be able to provide accurate predictions for both data it has previously seen in the training step as well as previously unseen data.

Supervised machine learning models vary in complexity, with simple models often having shorter training times, and more complex models having higher accuracy [6], although this is not a hard and fast rule. For example, certain supervised machine learning models are better suited for a particular type of data than others. As such, model providers must often weigh the pros and cons of using a particular supervised machine learning model to provide predictions.

Supervised machine learning models can more specifically be broken down into two main types of models: classification models which aim to determine the class of a query, and regression models which aim to predict a numeric value for a query.

4.1.1.1 Classification Models

A classification model is trained on a dataset D where each of the labelled elements belong to a particular class C. The model's goal is to learn how to determine which class C an element belongs to based on the other features provided to the model. Once the model has been trained, given a new query Q, the classification model M can predict the class this query belongs to based on the value of each of its features. A classification model's success is often measured by model accuracy, a metric which

measures the total number of correct classifications the model made over the total number of classifications made by the model.

An example classification model could aim to predict whether a high school student is likely to be accepted as an undergraduate student at UNB based on features such as their high school GPA, the number of clubs they participate in, their number of volunteer hours, and the number of sports they play competitively as seen in Figure 4.2.

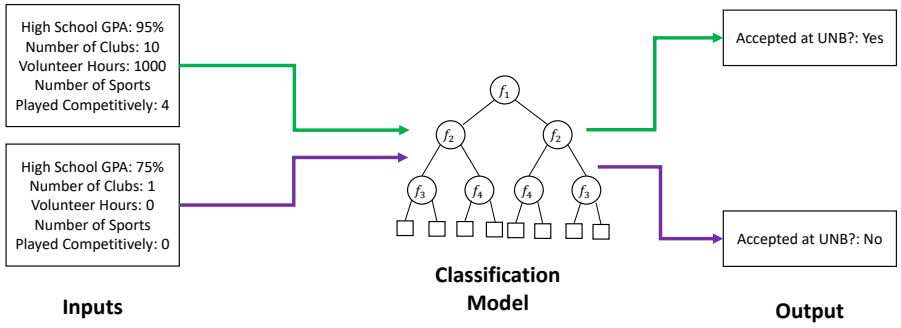

Figure 4.2: An example of a classification model which predicts whether a student will be accepted into UNB or not

Popular classification models include ID3, Random Forest, and AdaBoost.

4.1.1.2 Regression Models

A regression model is trained on a dataset D where the labelled elements of the dataset are numeric values rather than class labels. The model's goal is to predict the numeric quantity that an element would take on based on the other features provided to the model. Once the model has been trained, given a new query Q, the regression model M can predict the numeric value of a feature that the query would have based on the value of each of its other features. A regression model's success is often measured by its mean squared error, a metric which calculates the mean of the squared differences between the model's predicted numeric values and the actual values.

An example regression model could aim to predict the numeric GPA of the current school term that a UNB student will achieve based on features such as the student's age, their previous term's GPA, and the number of courses they are taking, as seen in Figure 4.3.

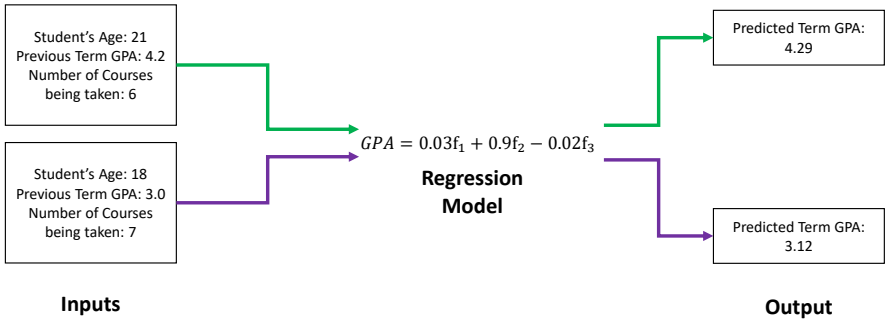

Figure 4.3: An example of a regression model which predicts a UNB student's term GPA

Popular regression models include Linear Regression, Logistic Regression, and Polynomial Regression.

4.1.2 Unsupervised Learning

In unsupervised machine learning, a model is trained on an unlabelled dataset D. Here the model does not aim to label elements in the dataset but rather discover patterns within the dataset [6]. The model may attempt to group similar elements in a task called clustering, may analyze the features in a dataset to determine which ones are linked in a task called association rules, or be used to perform dimensionality reduction, a technique used to reduce the number of features in a dataset [12].

An example of an unsupervised machine learning task would be identifying an anomalous traffic event to a web server. A clustering model would be able to identify which network traffic is normal and group them together while abnormal behaviour may fall into a separate cluster. This example can be seen in Figure 4.4.

Popular unsupervised models include Isolation Forests, K-Means Clustering, and

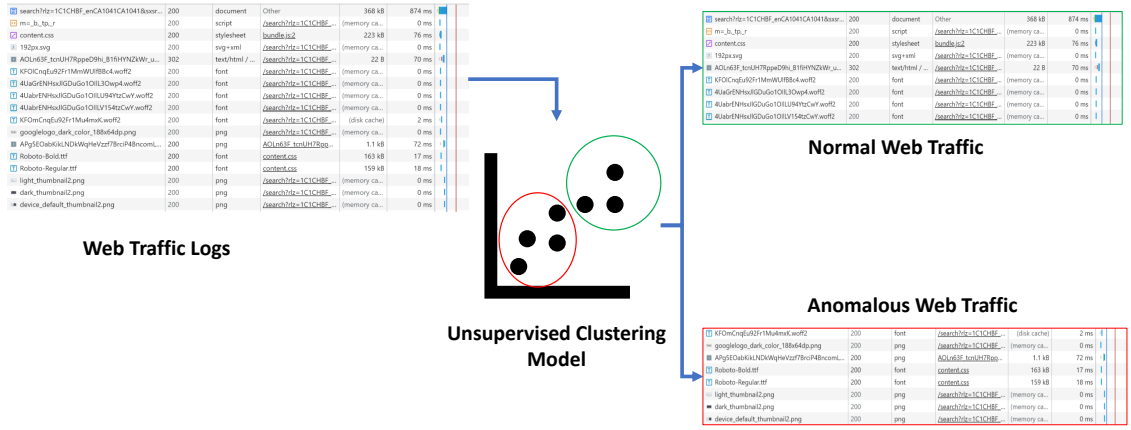

Figure 4.4: An example of an unsupervised machine learning model which clusters anomalous web traffic

Principal Component Analysis.

4.1.3 Reinforcement Learning

In reinforcement learning, models are trained in a natural way that mimics the that way humans learn [11]. Models are often called agents and perform actions in their environment in order to achieve a task during a particular round of training. The action that the agent performs is assigned a reward based on how well it performs the action at the end of each round. In the next round the agent will perform another action which aims to balance the risk between exploration of the possible actions it can take and exploitation of the knowledge it has gained about its task from previous rounds. Once the algorithm has determined that the agent has reached a threshold for peforming its task, the reinforcement model has been trained.

Reinforcement learning has been applied in many domains including regression, classification, and clustering tasks as described above. However, reinforcement learning can also be used in other domains such as creating deep fakes, and training agents to become better at playing video games.

Popular reinforcement learning algorithms include Q-learning and Monte Carlo.

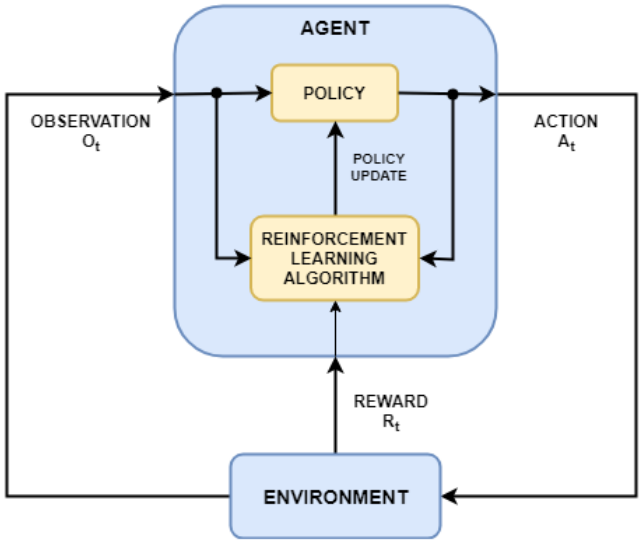

Figure 4.5: The Reinforcement Learning Pipeline [20]

4.2 Cybersecurity

In this section we discuss some preliminaries on cybersecurity fundamentals.

4.2.1 Encryption

Sharing unencrypted data—also known as plaintext—is not secure. If attackers are able to compromise a system and gain access to the unencrypted data, there is nothing standing in the way of the attackers using that information for their own personal gain. To ensure the privacy of data, data must be encrypted.

Encryption schemes largely fall into one of two categories: symmetric key encryption whereby the same key is used for encryption and decryption, and public key encryption whereby different keys are used for encryption and decryption. Homomorphic encryption, a subset of public key encryption, allows operations to be carried out on ciphertexts without the need to decrypt them first, and is commonly used for privacy preserving machine learning algorithms such as the schemes in [1, 18, 19, 32]. By encrypting a set of data, we aim to prevent attackers from being able to extract useful information from a ciphertext it has obtained.

Encryption schemes contain two main functions. The first function $Encrypt(K, M)$ takes a message M and uses the key K to produce the ciphertext (encrypted message), C. The second function $Decrypt(K, C)$ takes a ciphertext C and uses the key K to decrypt it to produce the original message M. While the process by which the key is used to encrypt M and decrypt C varies between encryption algorithms, these are the two fundamental pieces required for an encryption scheme.

4.2.1.1 Symmetric Key Encryption

In symmetric key encryption pictured in Figure 4.6, the same key K is used for both encryption and decryption. As such, to encrypt a message M, the function $Encrypt(M, K)$ is called which produces the ciphertext C. Once the ciphertext has been distributed to the correct party, the original message M can be recovered by running the function $Decrypt(C, K)$ using the same key that was used to encrypt the original message.

Some popular symmetric encryption algorithms include AES, DES, 3DES, and Blowfish.

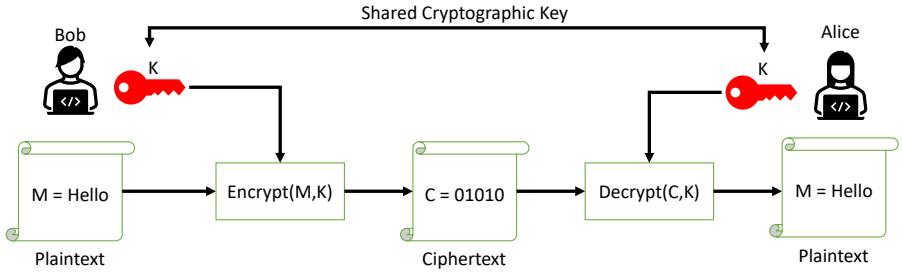

Figure 4.6: Symmetric Key Encryption

4.2.1.2 Public Key Encryption

In public key encryption, also known as asymmetric encryption pictured in Figure 4.7, two different keys are used for encryption and decryption. The public key PK is used for encrypting a message M, using the function $Encrypt(M, PK)$. Here the

public key is known to all parties and is published publicly. However, to recover the original message, the function $Decrypt(C, SK)$ is used which uses the client's secret key, SK. SK is known only to the receiver of a message. In order to be a proper public key encryption scheme, it must be infeasible to recover the secret key SK if an attacker knows the public key PK. Due to these properties, public key encryption is often mathematically more complex than symmetric key encryption.

Public key encryption algorithms include Diffie Helman, ECDSA, RSA, and ElGamal.

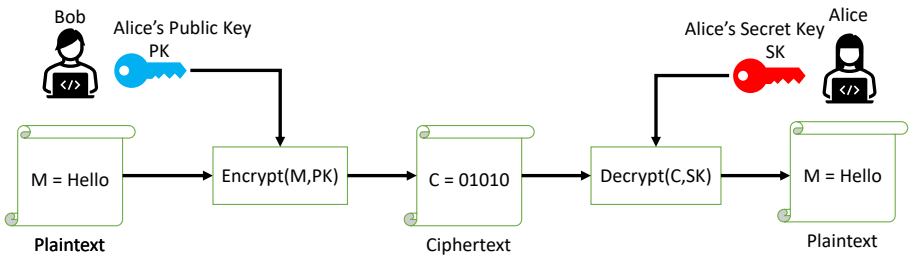

Figure 4.7: Public Key Encryption

4.2.1.3 Homomorphic Encryption

In homomorphic encryption, a subset of private key encryption, data does not need to be decrypted before operations can be performed on it. Transformations that need to be carried out on the data can be performed on encrypted data, unlike in traditonal encryption schemes. This property of homomorphic encryption makes it a good candidate for many scenarios such as secure machine learning algorithms. However, as homomorphic encryption schemes are a subset of private key encryption schemes, they can often be time consuming to perform [39].

A homomorphic operation is a mathematical operation, $\cdot$, which satisfies the following property: Given two plaintext messages M_1 and M_2, and the homomorphic encryption scheme's encryption function, $Encrypt(M_1) \cdot Encrypt(M_2) = Encrypt(m1 \cdot m2)$.

In general we consider that there are 3 types of homomorphic encryption: partially homomorphic encryption (PHE), somewhat homomorphic encryption (SHE), and fully homomorphic encryption (FHE). The differences between these schemes depends on the set of operations that can be carried out on encrypted data, and the number of times that this operation can be carried out. The differences between these schemes is summarized in Table 4.1.

Table 4.1: Differences Between Homomorphic Encryption Schemes

Homomorphic Encryption Scheme	**Homomorphic Operation**	**Number of Homomorphic Operations Permitted**	**Example Scheme**
Partially Homomorphic Encryption (PHE)	Addition *OR* Multiplication	Unlimited	RSA
Somewhat Homomorphic Encryption (SHE)	Addition *AND* Multiplication	Limited	YASHE
Fully Homomorphic Encryption (FHE)	Addition *AND* Multiplication	Unimited	Gentry's FHE Scheme

In a partially homomorphic encryption scheme only a single mathematical operation is supported. The supported operation can be either addition or multiplication. Under partially homomorphic encryption, the supported operation can be carried out an unlimited number of times on the encrypted data without compromising its homomorphic nature.

In somewhat homomorphic encryption, both addition and multiplication operations can be carried out on encrypted data. Unlike partially homomorphic encryption, operations can only be carried out a finite number of times while still maintaining the homomorphic properties of the operation.

In fully homomorphic encryption, both addition and multiplication operations can be carried out on encrypted data. Fully homomorphic encryption is less restrictive than somewhat homomorphic encryption as it allows the homomorphic operation to be carried out an unlimited number of times without compromising its homomorphic nature.

While homomorphic encryption is attractive due to its ability to allow operations to be carried out on both plaintext and encrypted data, homomorphic encryption schemes are often more computationally expensive than simple symmetric encryption [13].

4.2.2 Hash Functions

A hash function is a function $H : \{0,1\} * - > \{0,1\}^l$, which takes an input of arbitrary length and maps it to a fixed length output of size l. The hash value of a message M, called the message digest, is computed as $h = H(M)$. Figure 4.8 shows this process.

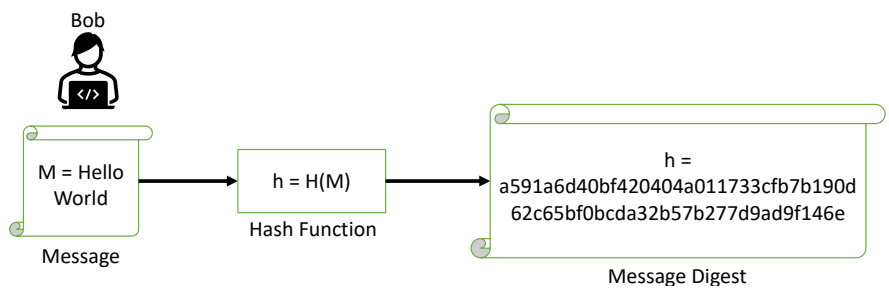

Figure 4.8: Hash Function

Hash functions have several properties which they must satisfy to be considered a good hash function. These are:

- Preimage Resistance: Given a message digest y it is computationally infeasible to find a message x such that $H(x) = y$. I.e. it is computationally infeasible for an attacker to determine a message used to produce a hash value if they have recovered the hash value.

- Second Preimage Resistance: Given a message x and its hash value $y = H(x)$ it is computationally infeasible to find another message x' such that $H(x') = H(x)$. I.e. it is computationally infeasible to come up with another message that produces the same hash value as a known message.

- Collision Resistance: It is computationally infeasible to find any two distinct values x, y such that $H(x) = H(y)$. I.e. it is computationally infeasible to come up with two different messages which produce the same hash value.

Hash functions can be used for many purposes including digital signatures to verify who has sent a message, password verification without storage of plaintext passwords,

and can also be seen in proof-of-work blockchain based systems as well. Various cryptographic hash functions exist with varying degrees of complexity. Some common hash functions include DES MAC, MD5, and SHA.

4.3 Data Structures

In this section we discuss some preliminaries on the data structures which are used in our scheme.

4.3.1 Decision Tree

Decision trees are popular machine learning models, which can be used for classification tasks. Though there are many algorithms for producing decsion tree models, including ID3, C45, and CART, a decision tree's structure can be described generally. A decision tree is a tree data structure of depth d that is organized with decision nodes D_x which are each associated with a threshold value t_x. When a client wishes to predict the class of their query, they submit a feature vector f. Based on a mapping q, the decision node D_x chooses a feature from the feature vector supplied to the tree, and compares it to the threshold value t_x for that node. If the value is less than or equal to t_x, the classification process continues down the left branch of the decision tree, otherwise it continues down the right branch of the tree. This process continues until a leaf node is reached where a classification l is output by the tree. Decision tree models are highly regarded for their wide range of applicability in many differerent domains. A few classification tasks which decision trees have been shown to perform well at include tasks such as healthcare diagnoses [17, 38], identifying network intrusions [37], and determining what demographic information is important for successful online learning [29]. Furthermore decision trees can provide quick classification times. Decision trees can classify an input in $O(log_2(X + Y))$ for a

decision tree with X decision nodes, and Y leaf nodes.

An example decision tree can be seen in Figure 4.9.

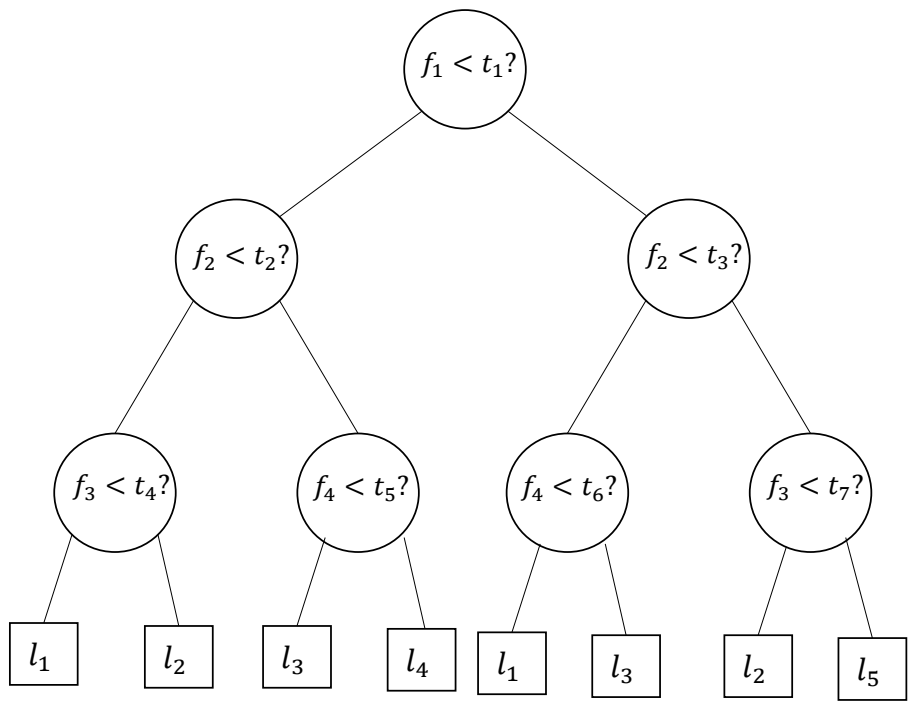

Figure 4.9: An Example Decision Tree

4.3.1.1 Full Binary Tree

A full binary tree is a tree in which all internal nodes have exactly 2 children, one left child, and one right child, and all leaf nodes are at the same level. As such, a full binary tree with d levels has 2^d leaf nodes and $2^d + 1$ internal nodes. For simplicity, we consider that all decision trees discussed in this thesis are full binary trees. While not all decision trees are full binary trees, it has been shown that binary trees can be extended to full trees through the use of dummy nodes [34]. Throughout this thesis we consider that all decision tree models are full binary trees, and in the event that they are not we first extend them to be complete binary trees using dummy nodes as described in [34].

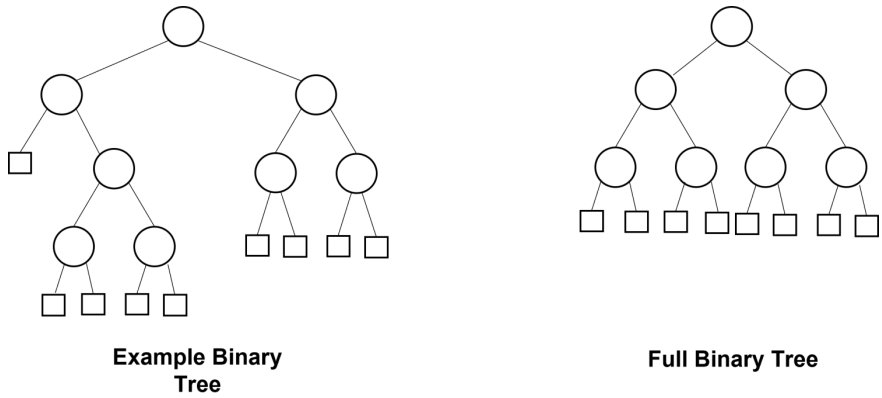

Figure 4.10: An Example Binary Tree and a Full Binary Tree

4.3.2 Bloom Filters

Bloom filters are data structures which are used for efficient tests of set membership [5]. A Bloom filter (BF) is composed two main components: a bit array $A[0...n-1]$ of length n, in which each element is initialized to 0, and a set of k independent hash functions $\mathcal{H} = \{h_1, h_2, \cdots, h_k\}$. Each h_i is defined as $h_i : \{0,1\}^* \rightarrow \{0, 1, 2, \cdots, n-1\}$. To create a Bloom filter that can check set membership of the set $\mathcal{U} = \{u_1, u_2, \cdots, u_m\}$ of size $|\mathcal{U}| = m$, we perform the following steps: for each element $u_j \in \mathcal{U}$, set $A[h_i(u_j)] = 1$. This is repeated for $i = 1, 2, \cdots, k$. Once all elements of $\mathcal{U}$ have been inserted in this way, our Bloom filter can perform tests of set membership for the set $\mathcal{U}$.

To check whether a new element w is in the set $\mathcal{U}$, we simply call the function $checkMembership(BF, w)$ which verifies that $A[h_i(w)] = 1$, for $i = 1, 2, \cdots, k$. If all of these elements of the Bloom filter are set to 1, then we say that the element w is probably in the set $\mathcal{U}$ with a false positive rate of fp and $checkMembership(BF, w)$ returns true. Otherwise, if even a single $A[h_i(w)] = 0$ then we can say with certainty that the element w is not in the set $\mathcal{U}$ and $checkMembership(BF, w)$ returns false. While Bloom filters allow for quick tests of set membership they do so by allowing for some percentage of false positives to be possible. As shown in Figure 4.11, the Bloom filter correctly identifies that $x, y \in \mathcal{U}$. However for $w \notin \mathcal{U}$, the BF

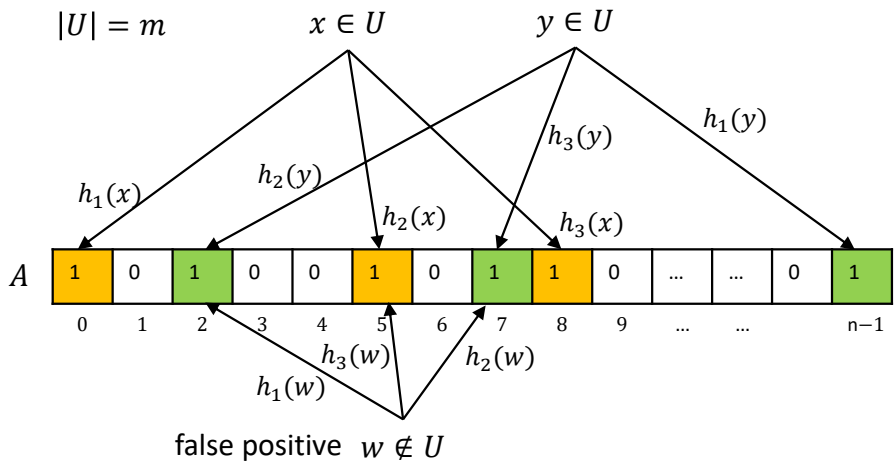

Figure 4.11: An example of Bloom filter with $k = 3$, where $x, y \in \mathcal{U}$, $w \notin \mathcal{U}$, but the Bloom filter falsely reports $w \in \mathcal{U}$.

incorrectly says that $w \in \mathcal{U}$, since $A[h_i(w)] = 1$ for $i = 1, 2, 3$. Luckily, by modifying the parameters of our Bloom filter, we can minimize our false positive rate. For an n-bit Bloom filter in which m elements have been mapped to the n positions with equal probabilities, the probability of the Bloom filter reporting a false positive is $fp = (1 - (1 - 1/n)^{km})^k \approx (1 - e^{-km/n})^k$. fp can be minimized when $k = \ln 2 \cdot \frac{n}{m}$. In this paper, we use Bloom filters to perform privacy-preserving outsourcing of decision tree models to cloud servers.

4.3.3 Heap

A heap is a tree based data structure whereby nodes of a tree are stored in an array structure $H[0...n]$. As shown in Figure 4.12, the first element in the heap $H[0]$ represents the root of the tree, while all other elements of the heap $H[i]$ represent other nodes in the tree. Children of a particular node i can be found by looking at $H[2i + 1]$ for its left child, and $H[2i + 2]$ for its right child.

In this paper, we consider using a heap to help represent the path that a decision tree followed to make its final classification on an input. Specifically, we consider a heap to be an array where the value of a particular $H[i]$ represents whether we take

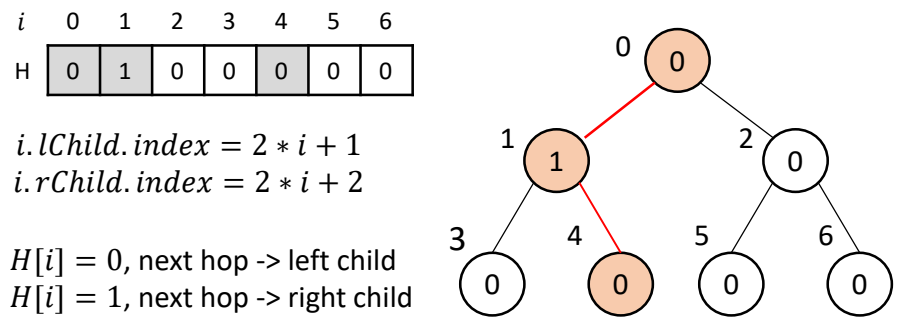

Figure 4.12: An example of heap-based path decision, where the path is 0-1-4.

the left branch ($H[i] = 0$), or the right branch ($H[i] = 1$) at a particular node i in the tree. By storing all of the decisions at each node in the tree in this way, we can easily store the path a decision tree took to provide classification of a query.

For example, in Figure 4.12, as the value of the root node 0 is $H[0] = 0$, we go to the left child node 1; as the value of $H[1] = 1$, we then go to the right child node 4. As node 4 is a leaf node, the path $0 - 1 - 4$ is decided by the heap. In our proposed scheme, a heap will be used to determine the path that the decision tree takes to classify a feature vector.

Chapter 5

Proposed Scheme

In this section we present our proposed scheme, which is composed of three main phases: model preparation, query request and response, and response recovery.
The first phase, model preparation, describes how the model provider prepares the decision tree model to be uploaded to the cloud servers, and distribution of cryptographic keys. The second phase, query upload and processing, is carried out by the client and the servers. In this phase, the client submits their query to the server and then disconnects from the servers while the query is processed. The final phase, response reconstruction, is then carried out by the client once they have received strings from the two servers. From this information the client will be able to retrieve a classification for their query. An overview of our scheme can be seen in Figure 5.1. A more detailed description of these phases follows.

5.1 Model Preparation

Given a set of public features $\mathcal{F} = \{f_1, f_2, \cdots, f_N\}$ of size N, the model provider first trains a decision tree (DT) model as normal, which is formed by using a subset of features $\mathcal{F}_S = \{f'_1, f'_2, \cdots, f'_S\} \subseteq \mathcal{F}$ of size S. This subset of features, determined by the decision tree algorithm, are the features that the decision tree algorithm

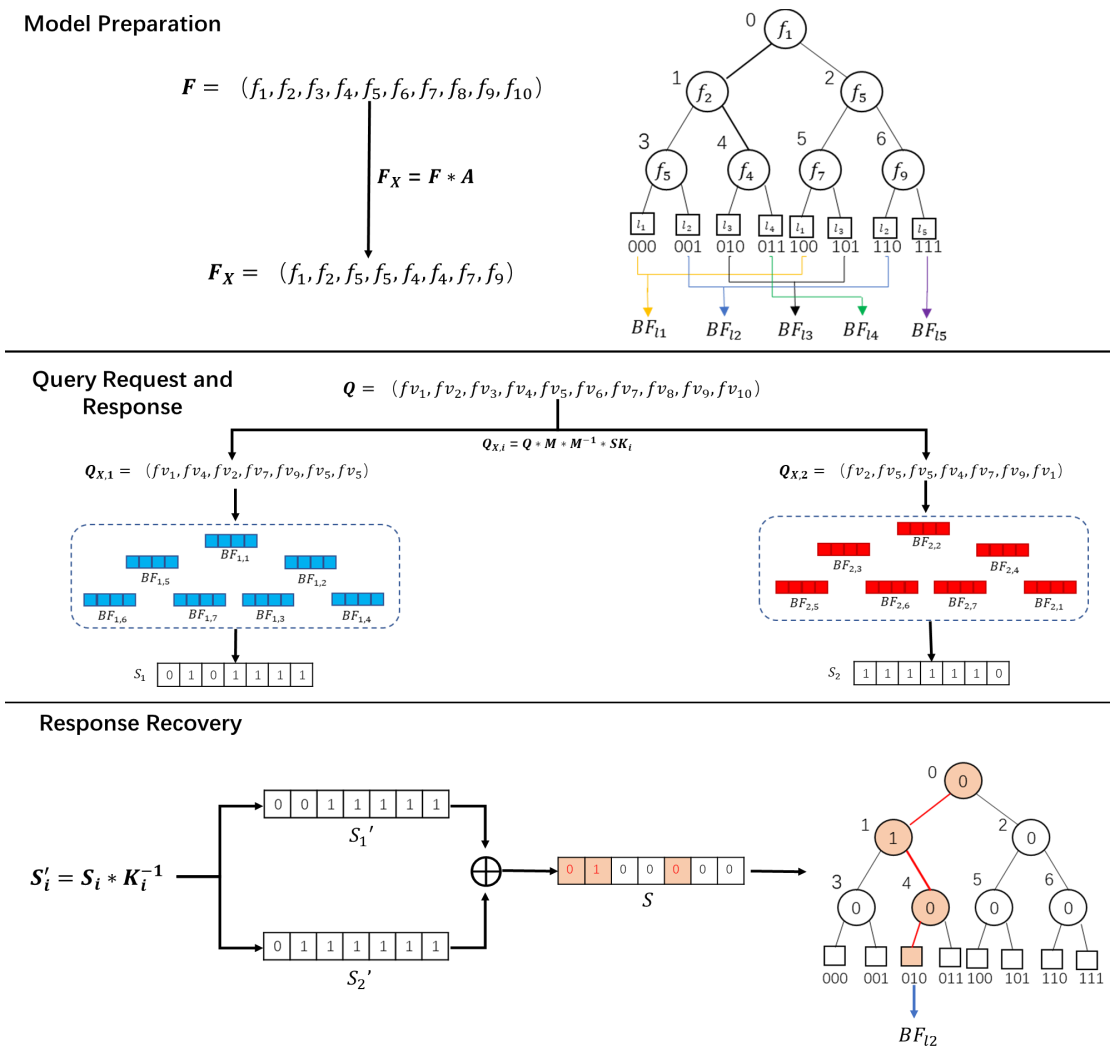

Figure 5.1: An illustrative example of our proposed scheme

uses to perform classification. It is possible that some $f_i \in F$ are deemed to be unimportant to the decision tree algorithm, and will not be used in the decision tree model. For simplicity, we consider that the tree produced by the model provider is a full binary tree. In the event that the model produced is not a full binary tree, we can choose to extend it by simply including additional dummy nodes as discussed in [34]. These dummy nodes can simply test any feature value with any threshold value such that the dummy nodes' decisions do not alter the final classification provided by the original DT.

Once the decision tree algorithm concludes, a decision tree model with X decision nodes and Y leaf nodes is produced. As shown in Figure 5.2 a vector $F_X =< f_{x1}, f_{x2} ... f_{xX} >$ of size X is then extracted from the decision tree model. F_X is composed of features from F_S whereby the ith element of F_X denotes the feature tested by the decision tree at node i, numbered in heap order. It is possible that some elements in F_S appear multiple times in the tree, since a decision tree may test a single feature multiple times. Thus, $X \geq S$. A second vector $T =< t_1, t_2 ... t_X >$ is also extracted from the decision tree model, in which each element t_i represents the threshold value for the feature tested at node i.

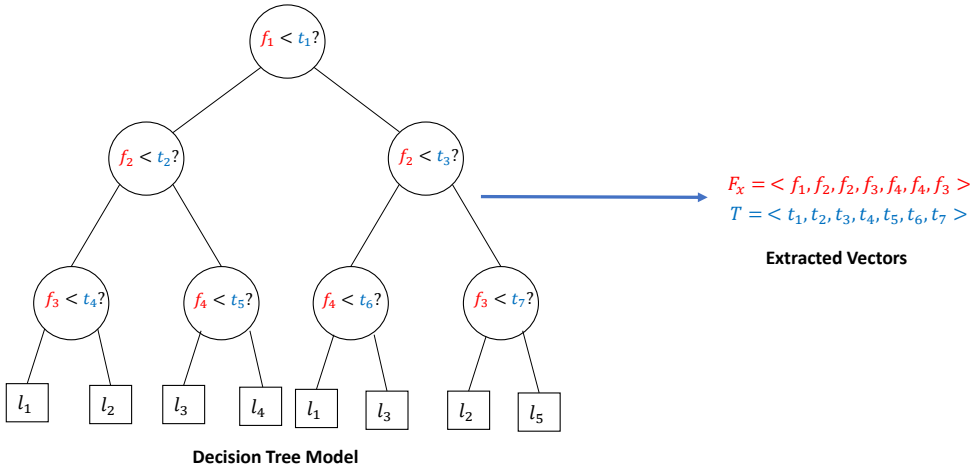

Figure 5.2: Vector Extraction from Decision Tree Models

Once the vectors T and F_X have been extracted from the provider's model, the model

can be prepared for outsourcing to the cloud servers C_1 and C_2.

To hide threshold values used by the DT model from the cloud servers, the model provider uses a novel Bloom filter technique to split the DT model into two vectors of Bloom filters, $BF_1 =< BF_{1,1}, BF_{1,2}...BF_{1,X} >$ and $BF_2 =< BF_{2,1}, BF_{2,2}...BF_{2,X} >$ which can mimic the original DT model. The first vector of Bloom filters, BF_1, is outsourced to C_1, and BF_2 is outsourced to C_2. The vectors of Bloom filters are created by running the following steps:

Step 1: The model provider takes the DT model with X internal nodes (the root node included) and constructs two Bloom filters for each internal node in the tree. We recall that a particular node j in the DT model tests whether feature $f_j \in \mathcal{F}_X \leq t_j \in T$. Each f_j comes from a domain U_j, which denotes all the possible values a feature f_j could have. As such we can consider that the threshold value t_j for a particular node j in the DT, splits the domain U_j into two sets: $A_j = \{u|u \in U_j, u \leq t_j\}$ and $B_j = \{u|u \in U_j, u > t_j\}$.

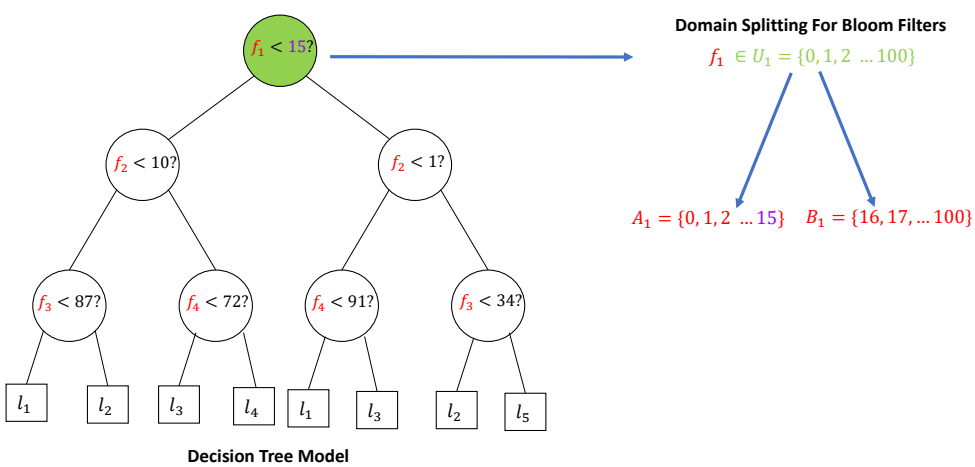

Figure 5.3: Node Domain Splitting

Using this insight, we can create two Bloom filters, $BF_{1,j}$ and $BF_{2,j}$ for each node j in the DT, which can mimic the the test done at a particular decision node. Each $BF_{1,j}$ and $BF_{2,j}$ are constructed by inserting all $u \in U_j$ into the appropriate Bloom filter based on the following process:

- If $u \in A_j$:
  - With a probability of $\frac{1}{2}$, insert u into both $BF_{1,j}$ and $BF_{2,j}$.
  - With a probability of $\frac{1}{2}$, do nothing.
- Otherwise, if $u \in B_j$:
  - With a probability of $\frac{1}{2}$ insert u into $BF_{1,j}$ and not into $BF_{2,j}$.
  - With a probability of $\frac{1}{2}$, insert u into $BF_{2,j}$ and not into $BF_{1,j}$.

This process will be repeated for all internal nodes in the decision tree until a total of $2X$ Bloom filters have been created.

By creating the Bloom filters in this way, we aim to split a decision node j between the two cloud servers. Previously on query time, a single node j could be used to test whether the value fv_j the client's value for feature f_j is less than or equal to t_j. With our scheme however we change the test $f_j \leq t_j$ into a set membership problem instead. If $f_j \leq t_j$ evaluates to true, then the client's feature value $fv_j \in A_j$. Otherwise if $fv_j \leq t_j$ evaluates to false, then the client's feature value $fv_j \in B_j$.

We can see that we have split this test of set membership between the Bloom filters $BF_{1,j}$ and $BF_{2,j}$, based on the above creation scheme. For a particular value fv_j to be tested by node j we can see that if $fv_j \leq t_j$ then $fv_j \in A_j$. In our scheme we could say that $fv_j \in A_j \; iff$ both $checkMembership(BF_{1,j}, fv_j)$ and $checkMembership(BF_{2,j}, fv_j)$ evaluate to true as seen in Figure 5.4, or if both $checkMembership(BF_{1,j}, fv_j)$ and $checkMembership(BF_{2,j}, fv_j)$ evaluate to false as seen in Figure 5.5.

Similarly if $fv_j > t_j$ then $fv_j \in B_j$. In our scheme we could say that $fv_j \in B_j \; iff$ either $checkMembership(BF_{1,j}, fv_j)$ evaluates to true and $checkMembership(BF_{2,j}, fv_j)$ evaluates to false as seen in Figure 5.6 or if $checkMembership(BF_{1,j}, fv_j)$ evaluates to false and $checkMembership(BF_{2,j}, fv_j)$ evaluates to true as seen in Figure 5.7.

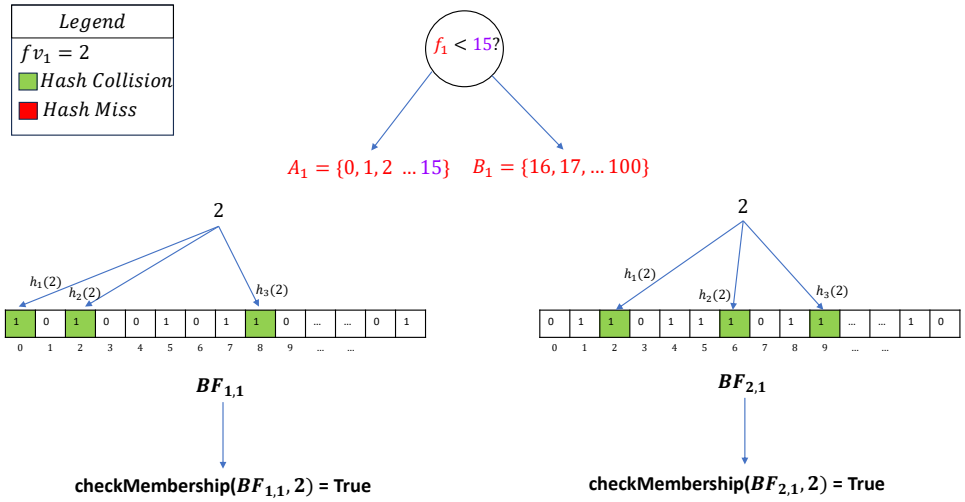

Figure 5.4: An Example Demonstrating fv_j in A_j and $checkMembership(BF_{1,j}, fv_j)$ and $checkMembership(BF_{2,j})$ Evaluate to True

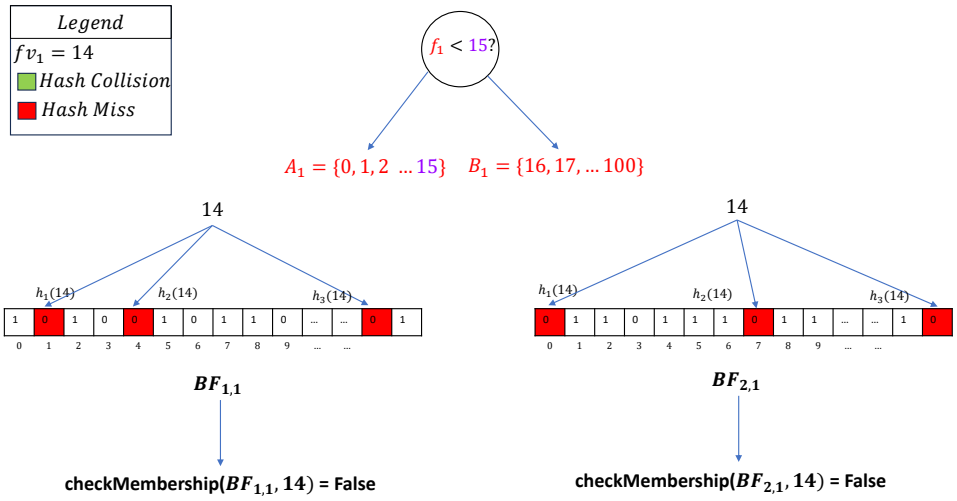

Figure 5.5: An Example Demonstrating fv_j in A_j and $checkMembership(BF_{1,j}, fv_j)$ and $checkMembership(BF_{2,j})$ Evaluate to False

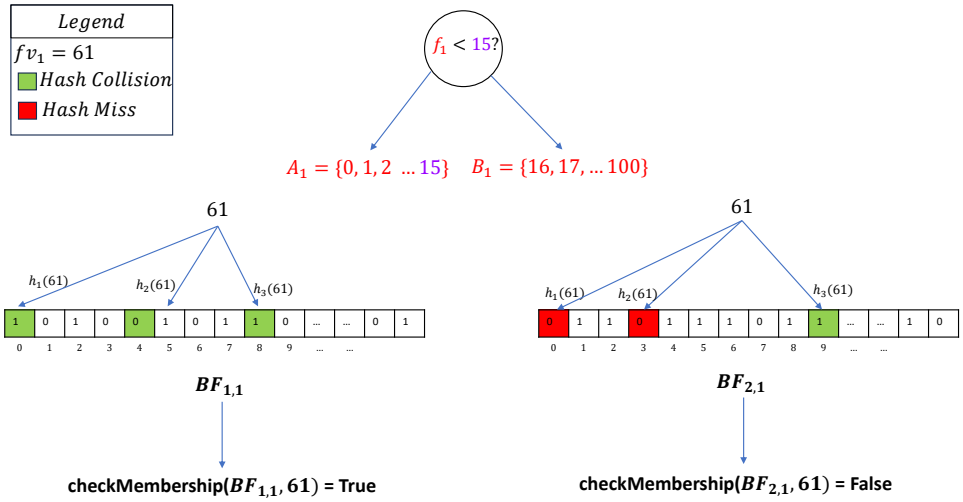

Figure 5.6: An Example Demonstrating fv_j in B_j and $checkMembership(BF_{1,j}, fv_j)$ Evaluates to True and $checkMembership(BF_{2,j})$ Evaluates to False

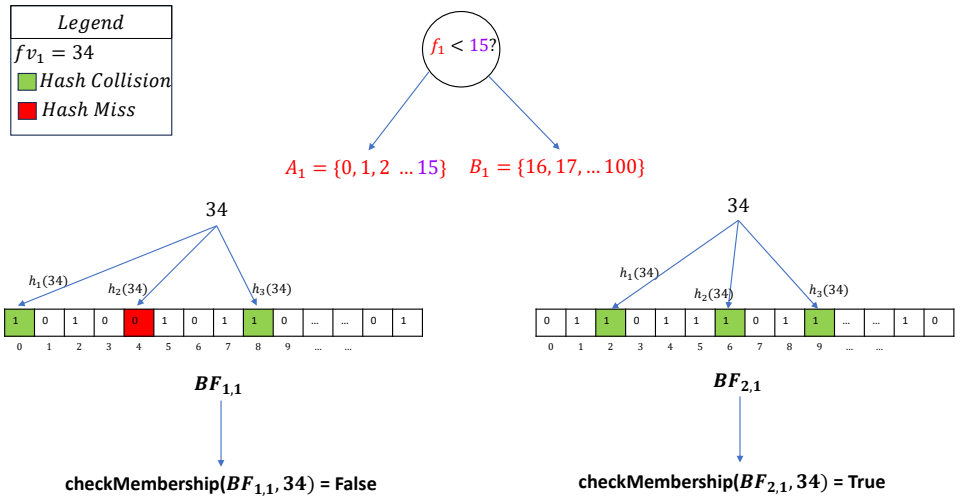

Figure 5.7: An Example Demonstrating fv_j in B_j and $checkMembership(BF_{1,j}, fv_j)$ Evaluates to False and $checkMembership(BF_{2,j})$ Evaluates to True

By splitting the problem of set membership in this way we prevent a single cloud server from being able to provide classification, since with only one of BF_1 or BF_2 a cloud server can only determine that a value $fv_j \in A_j$ or $fv_j \in B_j$ with a 50% chance.

Note that we consider that each of the domains U_j is an integer based domain in this scheme. So if the domain U_j would normally contain floating point values, we must first convert it to integer based values by multiplying by powers of 10 or something similar before the model training occurs. However, if the domain U_j contains boolean values $\{0, 1\}$, we can easily convert U_j into a larger integer based domain $U'_j = \{1 * * \cdots *, 0 * * \cdots *\}$, where each $* \in \{0, 1\}$. This can prevent the cloud servers from guessing the boolean variables in the Bloom filters, and allow our scheme to also account for boolean variables.

Step 2: Next, the model provider considers the set, L, of possible classifications that the DT model can predict. For each element $l \in L$, the model provider creates an additional Bloom filter BF_l for l, which will be used by the client to determine the final classification of their query. To populate these BF_l we first number the leaf nodes (LN) from 0 to $X + 1$ and create sets $Z_l = \{LN | LN.prediction = l\}$. Then for each $l \in L$ we insert all elements of Z_l into BF_l. These Bloom filters will be used by the client for the reconstruction step at the end of the scheme.

As the DT model is a full binary tree, there are a total of $X + 1$ leaf nodes. If each leaf node provides a unique classification label, then there are total $|L| = X + 1$ labeled Bloom filters for classification. However, if there are some dummy nodes in the DT model or if some leaf nodes predict the same class as another leaf node, then $|L| < X + 1$. For example, in Figure 5.1, if f_6 and f_4 are dummy nodes, then the leaf nodes $000, 001, 010, 011$ belong to the same class l, and they will be put into one Bloom filter with the same label. Upon completion these Bloom filters are publicly published.

Step 3: The model provider creates $\mathbf{M}$, an $N \times N$ invertible secret matrix, which will be securely sent to the client for encrypting their feature vector. Next a selection matrix $\mathbf{A}$ of size $N \times X$ is produced which extracts the required features from the client's original query in heap order. To achieve this, all of $A's$ elements are set to 0, except for a single element in each column of the matrix. Each column i has its jth element set to 1 where j represents the jth feature in the feature vector to be extracted and the column i represents the decision node in heap order which uses that tests the feature f_j.

Two matrices, K_1, K_2 of size $X \times X$ are then created by the provider. These matrices are used to shuffle the order of the Bloom filters BF_1 and BF_2 created in the previous step of this scheme. In each K_z column i has its jth element set to 1 where j represents the Bloom filter $BF_{z,j}$ which is moved from its original position to i. K_1 is then used to shuffle BF_1 the set of Bloom Filters to be distributed to C_1, and K_2 is used to shuffle BF_2, the set of Bloom Filters to be distributed to C_2. To produce the new shuffled vectors, we calculate $BF_1' = BF_1^T \cdot K_1$ and $BF_2' = BF_2^T \cdot K_2$. An example of shuffling BF_1 with K_1 to produce BF_1' can be seen in Equation 5.1.

$$BF_1' = \left(_{BF_1}\right)^T \cdot \mathbf{K_1} = \begin{pmatrix} BF_{1,1} \\ BF_{1,2} \\ BF_{1,3} \\ BF_{1,4} \\ BF_{1,5} \\ BF_{1,6} \\ BF_{1,7} \end{pmatrix}^T \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} BF_{1,1} \\ BF_{1,5} \\ BF_{1,2} \\ BF_{1,6} \\ BF_{1,7} \\ BF_{1,3} \\ BF_{1,4} \end{pmatrix}^T \tag{5.1}$$

By shuffling the order in which the Bloom filters are passed to the cloud servers we prevent them from knowing which nodes in the original tree test which features, since the order of the nodes in the tree are not preserved. Furthermore, even if the cloud servers were to collude, their output would be useless without knowing how

the Bloom filters have been shuffled since neither server would know which node in the original decsion tree a particular Bloom filter represents. As such, neither cloud server can gain any useful information from the decision tree.

Finally, the cloud servers secret keys SK_1, SK_2 where $SK_i = \mathbf{M}^{-1} \cdot \mathbf{A} \cdot K_i$ are produced. These keys will be used for privately extracting the required features from the client's original query for the DT classification in the required order needed by their Bloom filters.

The provider then creates the client's keys K_1^{-1} and K_2^{-1}. These keys will be used to unshuffle the outputs from C_1 and C_2 respectively at the end of the scheme.

Step 4: BF_1' and SK_1 is then distributed to C_1 and BF_2' and SK_2 is sent to C_2. Finally, the client's keys K_1^{-1}, K_2^{-1}, and M are securely distributed to the client. At the end of this phase, the model provider has completed their work, and remains inactive for the remainder of the scheme.

5.2 Query Request and Response

When the client wishes to get a prediction from the DT model, the client and two cloud servers C_1, C_2 will run the following steps:

Step 1: Based on the publicly available feature set $\mathcal{F} = \{f_1, f_2, \cdots, f_N\}$, the client fills the feature value fv_i for each feature $f_i \in \mathcal{F}$ to form a query $Q = (fv_1, fv_2, \cdots, fv_N)$.

Step 2: The client uses $\mathbf{M}$ to encrypt the query $Q = (fv_1, fv_2, \cdots, fv_N)$ into $EQ = Q \cdot \mathbf{M}$, and sends EQ to the two cloud servers.

Step 3: After receiving EQ, each cloud server C_i, $i \in \{1, 2\}$, uses its secret key SK_i to calculate $Q_{X,i}$ a vector containing the feature values in the order that they are tested in by BF_i'. This is done by calulating Q_{X_i} as follows:

$$Q_{X,i} = EQ \cdot SK_i = Q \cdot \mathbf{M} \cdot \mathbf{M}^{-1} \cdot \mathbf{A} \cdot \mathbf{K_i} = Q \cdot \mathbf{A} \cdot \mathbf{K_i} \tag{5.2}$$

For $Q_{X,1}$ in the example in Figure 5.1, we have

$$Q_{X,1} = \begin{pmatrix} fv_1 \\ fv_4 \\ fv_2 \\ fv_7 \\ fv_9 \\ fv_5 \\ fv_5 \end{pmatrix}^T = Q \cdot \mathbf{M} \cdot \mathbf{M}^{-1} \cdot \mathbf{A} \cdot \mathbf{K_1} = \begin{pmatrix} fv_1 \\ fv_2 \\ fv_3 \\ fv_4 \\ fv_5 \\ fv_6 \\ fv_7 \\ fv_8 \\ fv_9 \\ fv_{10} \end{pmatrix}^T \cdot \mathbf{A} \cdot \mathbf{K_1} \tag{5.3}$$

It is simple for us to verify that this equation correctly grabs the feature values that we want from a public query $Q =< f_1, f_2 ... f_N >$ to a vector $Q_{X,i}$ in the correct order. Equation 5.2 simplifies to $Q_{X,i} = EQ \cdot \mathbf{SK_i} = Q \cdot \mathbf{A} \cdot \mathbf{K_i}$. Since A acts as a matrix which extracts the features required from Q in the order in which they are tested in the DT represented as a heap, then multiplying this by K_i simply shuffles the order of the feature values to match the order that they would be tested in by BF'_i, the shuffled Bloom filter vector for C_i.

For example, as shown in Figure 5.1, given $\mathcal{F} = \{f_1, f_2, \cdots, f_{10}\}$, a DT model with $\mathcal{F}_X = \{f_1, f_2, f_5, f_5, f_4, f_7, f_9\}$ will be trained. As discussed above, we can describe the DT model as a vector $(f_1, f_2, f_5, f_6, f_4, f_7, f_9)$ converted from the public vector $(f_1, f_2, \cdots, f_{10})$ with the matrix $\mathbf{A}$ as follows:

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \\ f_{10} \end{pmatrix}^T \cdot \mathbf{A} \cdot \mathbf{K_1} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \\ f_{10} \end{pmatrix}^T \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_5 \\ f_5 \\ f_4 \\ f_7 \\ f_9 \end{pmatrix}^T \tag{5.4}$$

Thus multiplying the above by K_1 in the example above will result in

$$\begin{pmatrix} f_1 \\ f_2 \\ f_5 \\ f_5 \\ f_4 \\ f_7 \\ f_9 \end{pmatrix}^T \cdot \mathbf{K_1} = \begin{pmatrix} f_1 \\ f_2 \\ f_5 \\ f_5 \\ f_4 \\ f_7 \\ f_9 \end{pmatrix}^T \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_4 \\ f_2 \\ f_7 \\ f_9 \\ f_5 \\ f_5 \end{pmatrix}^T \tag{5.5}$$

Given that BF_1 requires the features to be processed in this way due to our shuffling in the previous step, the vector produced above shows the correct order to process the features in the DT.

Although the cloud servers can recover $Q_{X,i} = (fv_1, fv_2, fv_5, fv_5, fv_4, fv_7, fv_9)$ in our example, they do not know which features each value $fv_i \in Q_{X,i}$ represents since the cloud does not know what the matrices A or K_i are. Since the cloud server C_i only receives $SK_i = \mathbf{M}^{-1} \cdot \mathbf{A} \cdot K_i$ they cannot recover A^{-1} or K^{-1} to retrieve the client's original query. Furthermore, since the nodes have been shuffled, the cloud server also does not know which node in the original tree tests a particular feature

value. As a result, the DT model can be hidden from the cloud servers.

Step 4: To evaluate the Bloom filter-based DT nodes, each cloud server C_i, $i \in \{1, 2\}$, performs the following steps. For each Bloom filter node BF_j in BF'_i, extract fv_j the jth element of $Q_{X,i}$. If fv_j is in the Bloom filter BF_j, C_i outputs 1, otherwise it outputs 0. Once all Bloom filter nodes have been processed by C_i, C_i should have output a bit string S_i of length X. This bit string S_i represents a shuffled heap whereby each of the elements in the heap is either 0 or 1. If the jth value in the bit string is 0, this means that the node represented by BF_j in the original split up tree would have sent the query down the left path of the tree if this node was reached, otherwise, if the jth value in the bit string is 1, node represented by BF_j would have sent the query down the right path of the tree if this node was reached. Finally C_i returns the bit string S_i back to the client.

5.3 Response Recovery

After receiving the two bit strings S_1 and S_2 from the cloud servers, the client must first unshuffle the bit strings using the keys K_1^{-1} and $K_2^{-}-1$ which it received from the model provider in the beginning of the scheme. To unshuffle each S_i the client simply has to compute $S'_i = S_i \cdot K_i^{-1}$, resulting in S'_1 and S'_2.

Once the client has unshuffled the two bitstrings S'_1 and S'_2 the client can calculate $S = S_1 \oplus S_2$ to determine the heap string, S, which represents how the initial tree created by the model provider would have classified the client's query. We can see that this is the case due to how we split up our Bloom filters in the model preparation phase. A single bit of S, $s_j = 1$ iff only one of the Bloom filters $BF_{i,j}$ contains fv_j, which corresponds to fv_j being less than or equal to the threshold value. Similarly $s_j = 0$ iff both $BF_{i,j}$'s contain fv_j or neither of them do, which corresponds to fv_j being greater than the threshold value.

The client then uses the heap string S to extract the path the original tree would taken to classify the client's query. This can be done by indexing the string S as follows:

- For the 0th step, set i (index) as 0, and output $S[0]$.
- While i is less than the length of S,
- Continuously set $i = 2 * i + 1 + S[i]$, and output $S[i]$.

The string output by this process, denoted as S_{class}, represents the binary value of the leaf node in the original tree with id S_{class}. To determine the final classification result of the query, the client simply checks which of the Bloom filters published by the model provider contains S_{class}. If a Bloom filter BF_l with the label l contains S_{class}, the client knows that the DT model predicts that their query would have class l.

Chapter 6

Security Analysis

In this chapter, we analyze the security of our proposed scheme under our defined security model.

6.1 Privacy-Preserving Decision Tree Model

In our proposed scheme, the original decision tree (DT) model is split into two Bloom filter-based DT models and respectively distributed to two cloud servers. As we consider the two cloud servers are non-collusive, each server cannot individually determine which feature a specific Bloom filter stands for. Each server may observe the number of 1's in a Bloom filter to guess whether it corresponds a boolean domain feature or not. For this attack, if we do not extend a boolean domain to a large domain, the server can correctly guess it with a high probability. Luckily, in our proposed scheme, we have considered this risk by transforming boolean features' domains into large-sized ones, which prevents the server from directly guessing the DT model from the Bloom filters.

Also from Equation (5.4), one server may first guess the matrix $\mathbf{A}$, and obtain the DT model by computing $(f_1, f_2, f_5, f_6, f_4, f_7, f_9) = (f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}) \cdot \mathbf{A}$. However, the server i cannot obtain $\mathbf{A}$, as it has been encrypted by the matrix $\mathbf{M}^{-1}$

as $\mathbf{M}^{-1} \cdot \mathbf{A} \cdot \mathbf{K_i}$.

Furthermore, as the client is honest in our security model, although they have the secret key $\mathbf{M}$, they will not collude with the server. Therefore, the server cannot obtain $\mathbf{M}$ through the client. With the above analysis, we can conclude the DT model is privacy-preserving in our proposed scheme.

6.2 Privacy-Preserving Client Query

In our proposed scheme, the client's query $Q = (fv_1, fv_2, \cdots, fv_N)$ is encrypted by $\mathbf{M}$ into $EQ = Q \cdot \mathbf{M}$. Without $\mathbf{M}$, one server cannot recover the query Q. From Equation (5.3), one server could obtain $Q_{X,i}$, ($Q_{X,1} = (fv_1, fv_4, fv_2, fv_7, fv_9, fv_5, fv_5)$ in our example). However, as the server does not know the original DT model, the nodes in the the tree have been shuffled, and all transformed features' domains are of a large size, knowing $Q_{X,i}$ does not provide more meaningful information to the server. Therefore, the client query is privacy-preserving in our proposed scheme.

6.3 Privacy-Preserving Client Query Result

In our proposed scheme, after obtaining S_1 from the server C_1 and S_2 from the server C_2, the client can compute the final classification result by first unshuffling the strings S_1 and S_2 using their keys K_1 and K_2, to produce S'_1 and S'_2. Finally they can compute the classification string $S = S'_1 \oplus S'_2$.

In order for a server C_i to reconstruct S they would first need to determine how to unshuffle their bitstring S_i. Since a bitstring contains X bits, of which $r < X$ are 1's, then there are $X!/(r!(X-r)!)$ ways to unscramble the bitstring. As such to correctly guess the unscrambling of a bitstring, the cloud server has a $1/X!/(r!(X-r)!)$ chance to correctly unscramble the bitstring.

If the cloud server i manages to unscamble their bitstring correctly, they still need

to correctly determine S'_{3-i} to recover the client's classification result. Since the two servers C_1 and C_2 are not collusive in our security model, each of them can only guess the correct S'_{3-i} with a probability of $1/2^X$, where X is the bit length of S'_{3-i}. Therefore, under our security model, the client query result is also privacy-preserving in our proposed scheme.

6.4 Privacy Preserving Threshold Values

In our proposed scheme, threshold values $t_j \in \mathcal{T} = \{t_1, t_2, \cdots, t_x\}$ are not disclosed to the client or the cloud servers. Unlike a normal decision tree which must dislose the value t_j to compare to a feature f_j at a particular decision node, our scheme does not need to disclose the value since it is inherently incorporated into how we create the Bloom filters $BF_{i,j}$ as described in Chapter 5. Since to the cloud, the Bloom filters simply look like a bit array filled with 1's and 0's, the cloud cannot determine what the threshold for a particular node j is. Furthermore, as the client does not see the decision tree model at all, the threshold values of our decision tree is also privacy-preserving in our scheme.

6.5 Prevention of Collusion Attacks

In our proposed scheme, we also prevent collusion attacks between two cloud servers. Normally we would consider that the two cloud servers, C_1 and C_2, would come from different cloud providers such as Microsoft Azure and Amazon Web Services like the researchers in [39, 40]. Such a scenario would make it unlikely that cloud servers would collude with each other since colluding would hurt their reputation as reputable cloud providers. However, for our scheme we can safely assume that even if the cloud servers collude they will not gain any valuable information.

The order of $BF_{1,j}$ and $BF_{2,j}$ are shuffled using the keys K_1 and K_2 prior to C_1 and C_2

receiving them. Thus, the cloud servers process the client query in a different order. When C_1 produces S_1 and C_2 produces S_2, both strings still need to be unshuffled before they can provide any useful information to the cloud servers. However since only the client, and the provider know the keys K_1^{-1} and K_2^{-1} which can unshuffle the strings, the cloud server cannot recover the final classification of the model on the client's query. Therefore, under our security model, the client query result is still privacy-preserving if the cloud servers collude with each other.

Furthermore, if the two cloud servers collude and share their Bloom filter models with one another, they cannot know which of their Bloom filters correspond to the same node in the original decision tree since their order is not preserved by the cloud servers. As such if a single feature f is tested p times in the decision tree, the cloud servers can only correctly guess a single Bloom filter's paired Bloom filter which also tests f with a rate of $1/p$.

Chapter 7

Experimental Analysis

In this section we perform experiments to analyze the efficiency of our scheme. We specifically look at how our model performs in its three main phases: model preparation, query request and response, and response recovery.

7.1 Experimental Settings

We evaluate our scheme on a Windows 11 laptop with an Intel i7 processor and 32GB of ram. The code is implemented in Python using various packages including *numpy, scikit-learn, pandas* and *pybloom*. A listing of the module code can be seen in Appendix A.

In our experiments we use Slate's Letter Recognition dataset [30] to create our DT models. This dataset aims to identify rectangular pixel displays as a particular capital letter. This dataset contains 20000 rows of data, 16 features, and 26 possible classification values. Table 7.1 contains a more detailed description about each of the features in the dataset.

During model training we use 80% of the data for training data, and reserve the remaining 20% of data for testing our models. During our testing process we did not notice any difference in model accuracy between the original DT model and the

Table 7.1: Slate's Letter Recognition Dataset Features

Column Number	Feature Information	Domain
1	Horizontal Position of Box	$\mathbb{Z}$
2	Vertical Position of Box	$\mathbb{Z}$
3	Width of Box	$\mathbb{Z}$
4	Height of Box	$\mathbb{Z}$
5	Total Number of "On" Pixels	$\mathbb{Z}$
6	Mean X Position of "On" Pixels	$\mathbb{Z}$
7	Mean Y Position of "On" Pixels	$\mathbb{Z}$
8	Mean X Variance	$\mathbb{Z}$
9	Mean Y Variance	$\mathbb{Z}$
10	Mean X Y Correlation	$\mathbb{Z}$
11	Mean of $X^2 * Y$	$\mathbb{Z}$
12	Mean of $X * Y^2$	$\mathbb{Z}$
13	Mean Edge Count From Left to Right	$\mathbb{Z}$
14	Mean Correlation of Mean Edge Count From Left to Right With Y	$\mathbb{Z}$
15	Mean Edge Count From Top to Bottom	$\mathbb{Z}$
16	Mean Correlation of Mean Edge Count From Top to Bottom With X	$\mathbb{Z}$
17	Letter (class)	0-25, (Letters encoded as integers with 0 = A, 1 = B ... 25 = Z)

Bloom filter based model produced by our scheme. However we do note that this is likely due to us setting the false positive rate fp to be 0.001 for each of our Bloom Filters.

A summary of the time to complete our scheme for a tree with 8, 128, 1024, and 8192 nodes can be seen in Table 7.2. A comparison of the amount of time needed for classification for these trees using our scheme and the original trees are shown in Table 7.3.

7.2 Model Preparation

The model preparation phase of our scheme can be broken down into several main steps of work. Namely the model preparation phase of our scheme carries out decision tree conversion where it converts the DT model to the Bloom filter based model, key generation where the cryptographic keys are generated, and the shuffling of Bloom filters. We detail various performance trials related to each of these steps in the following subsections.

7.2.1 Decision Tree Conversion

For each decision node j in the model provider's original DT, the provider creates two Bloom filters based on the feature f_j that j tests. As discussed above in Subsection

5.1, the model provider must insert all elements $u \in U_j$, the set of allowable values for f_j, into its appropriate Bloom filter. If all Bloom filters use the same of number of hash functions k, and a similar sized domain for U_j's, this operation has $O(X*k*|U|)$ time complexity. As such increasing any of X, k or $|U|$ individually will linearly affect the performance of the tree's conversion to Bloom filters.

In Figure 7.1 we can see the effect that increasing the number of nodes has on the time needed to convert the decision tree model to Bloom filters. In these trials $|U|$ was set to 100 and k was set to 10.

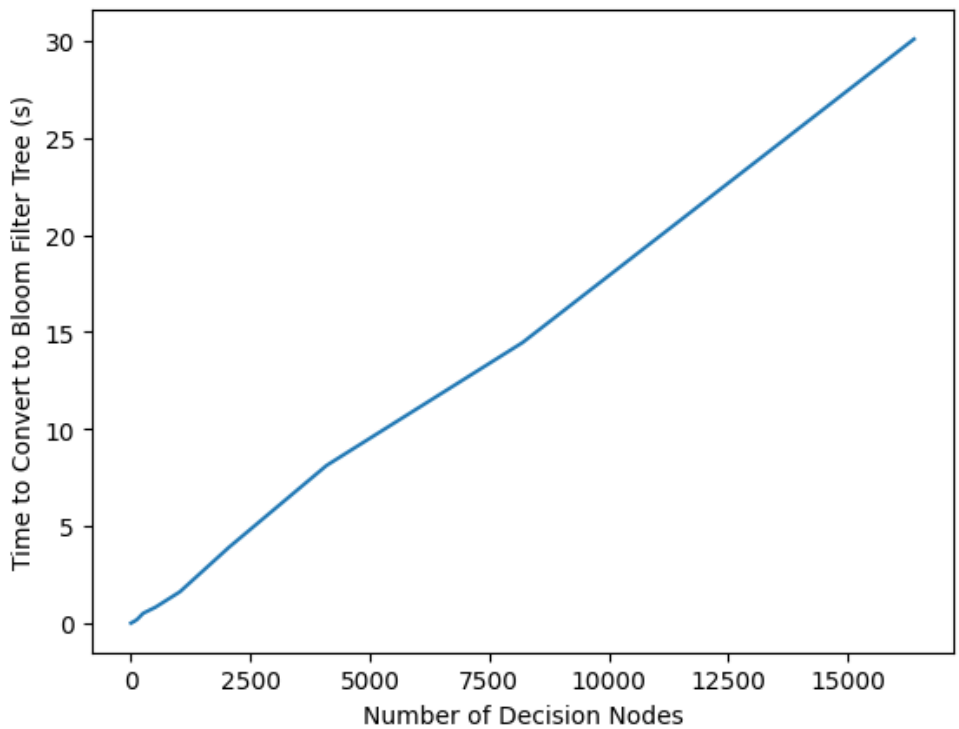

Figure 7.1: Number of Decision Nodes Versus Time to Convert Tree to Bloom Filters (s)

However, if we recall in the earlier sections we state that decision trees trained using our scheme must be either full binary trees, or they must be extended to a full binary tree before we can convert them using our scheme. As such the true complexity of converting the decision tree to Bloom filters is $O(2^d * k * |U|)$ for a tree with depth d. Thus while number of hash functions k and the domain size $|U|$

affect the scheme's performance linearly, the depth of the tree affects the scheme's peformance exponentially. This effect can be seen in Figure 7.2. $|U|$ and k were kept the same as the previous set of trials.

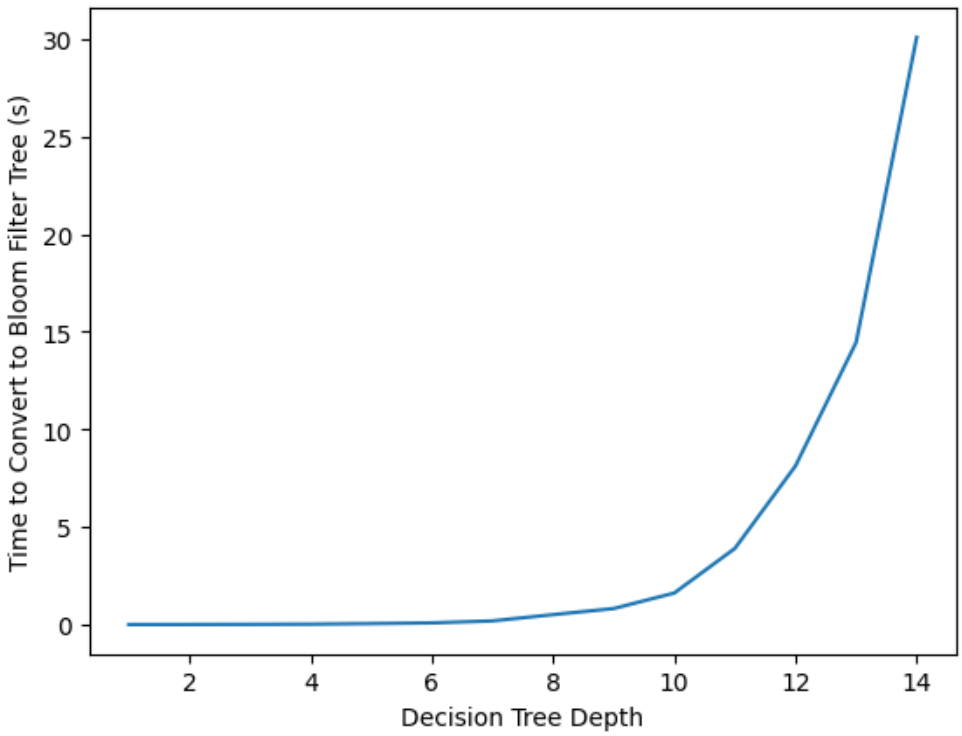

Figure 7.2: Decision Tree Depth Versus Time to Convert Tree to Bloom Filters (s)

Producing the Bloom filters for the leaf nodes however only takes $O((X+1)*k)$ time complexity since each leaf node only needs to be inserted into one Bloom filter. In Figure 7.3 we can see the effect that increasing the number of leaf nodes $X+1$ has on the amount of time needed to create the leaf node bloom filters. Similar to earlier, we keep k set to 10, in these trials.

Similar to our note above, since we consider that decision trees trained using our scheme must be either full binary trees, or they must be extended to a full binary tree before we can convert them using our scheme we can also consider that the operation of creating the leaf node Bloom filters has $O((2^d+1)*k)$ complexity since a full binary tree with depth d has 2^d+1 leaf nodes. This effect can be seen in Figure 7.4.

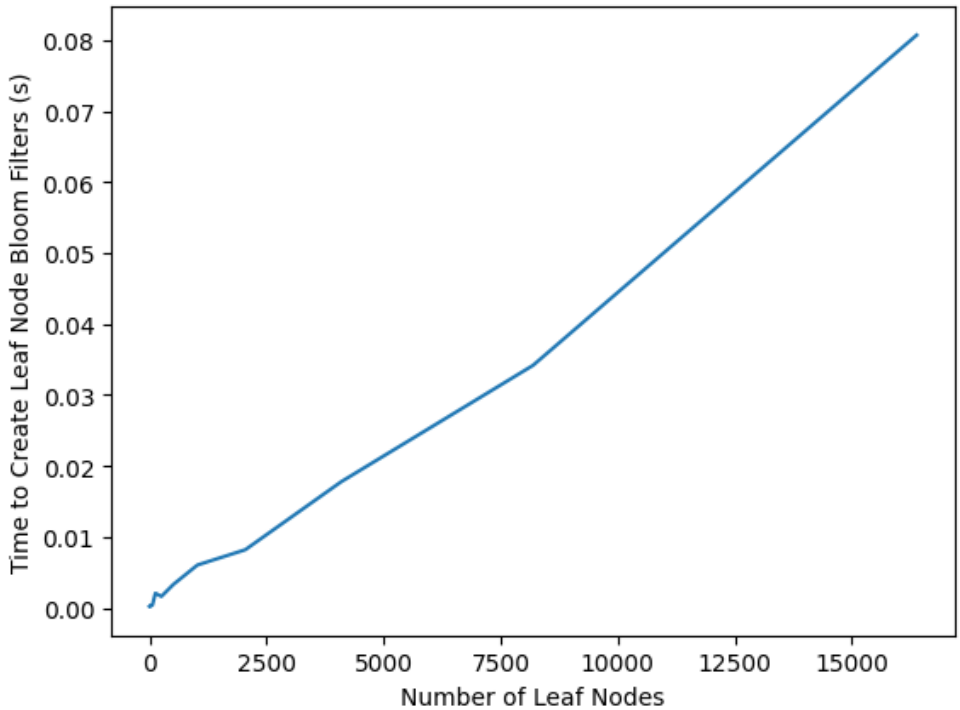

Figure 7.3: Number of Leaf Nodes Versus Time to Create Leaf Node Bloom Filters (s)

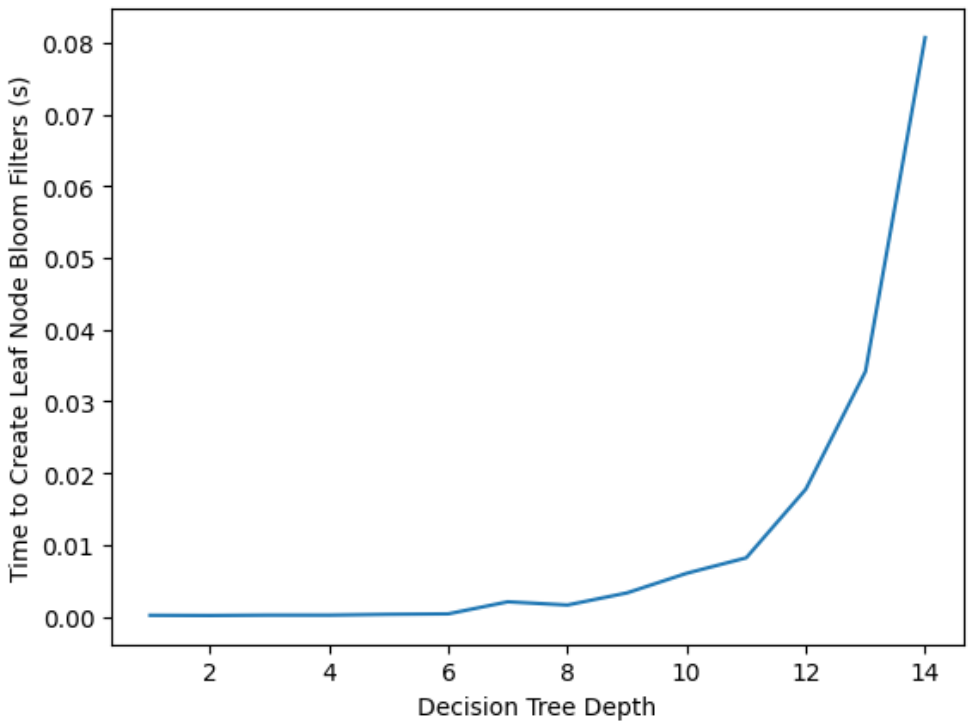

Figure 7.4: Decision Tree Depth Versus Time to Create Leaf Node Bloom Filters (s)

7.2.2 Key Generation

The client's secret key $\mathbf{M}$ can be generated by randomly creating a diagonally dominant $N \times N$ matrix, since diagonally dominant matrices are invertible. Next the feature selection matrix A of size $N \times X$ is generated to correctly extract the features needed from the client's feature vector. Next the keys, K_1 and K_2 for shuffling the Bloom filter vectors BF_1 and BF_2 respectively are generated. Each of these keys is of size $X \times X$.

Finally, once the initial matrices have been generated the provider must calculate the two cloud servers' keys $SK_i = \mathbf{M}^{-1} \cdot \mathbf{A} \cdot K_i$, as well as client's keys K_1^{-1} and K_2^{-1}, for unshuffling the classification result from each cloud server.

In this step of our scheme, the most time consuming operations are the matrix multiplications required to produce SK_i's and inverting M, K_1, and K_2. During our tests we analyzed the time taken to produce the keys in our scheme for various values of X while N is kept constant at 16. A summary of these experiments can be seen in Figure 7.5.

7.2.3 Shuffling Bloom Filters

Shuffling the Bloom filter vectors BF_1 and BF_2 to produce BF_1' and BF_2' depends only on the number of decision nodes in the full binary tree being converted in this scheme. Since we consider that we use matrix multiplication to produce $BF_i' = BF_i \cdot K_i$ for $i = 1, 2$, this shuffling operation has $O(X^2)$ time complexity. During our trials we analyze the time taken to shuffle the Bloom filters when X has various values. The results of these experiments can be seen in Figure 7.6.

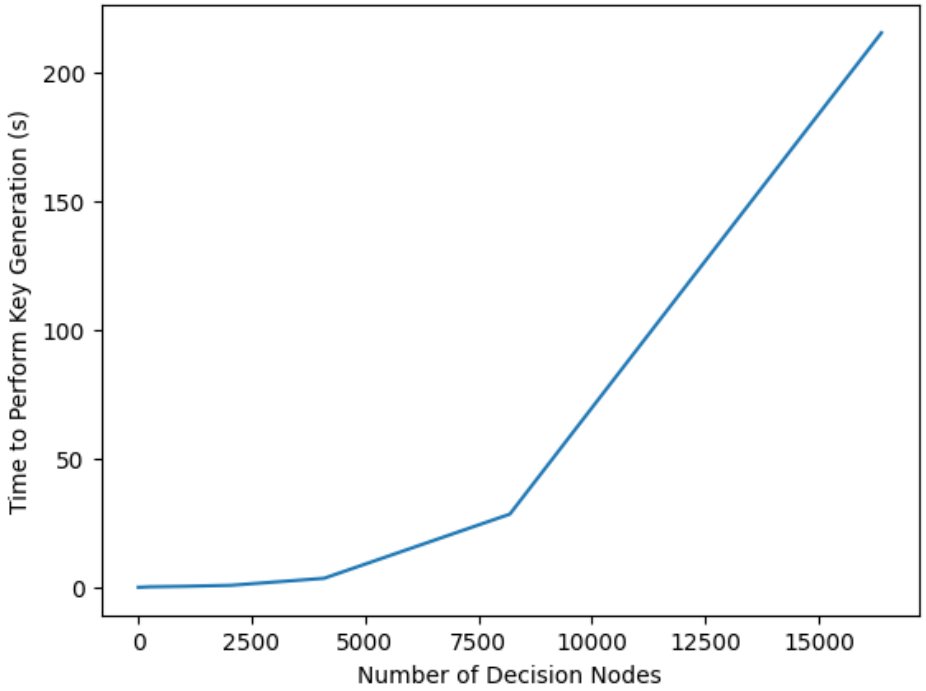

Figure 7.5: Number of Decision Nodes Versus Time to Generate Cryptographic Keys

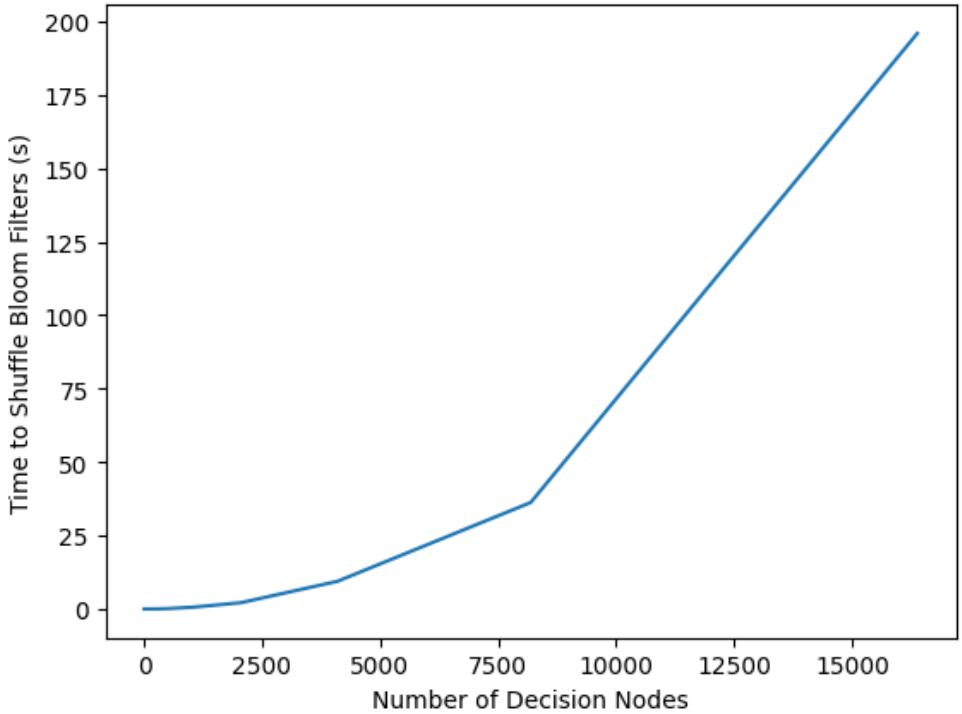

Figure 7.6: Number of Decision Nodes Versus Time to Shuffle Bloom Filters

7.3 Query Request and Response

For the query request and response phase of our scheme, there are three main steps which are carried out. At this stage all crytographic keys have been distributed to the cloud servers as well as the client, and the Bloom filter model has already been uploaded to the cloud servers for hosting. As such, this phase's main work comes from the encryption of the client's query, the creation of $Q_{X,i}$, and the classification by the cloud servers. In this section we once again perform various tests to see how our scheme performs for various sized decision trees.

7.3.1 Query Encryption

Encryption of the client's feature vector $Q = (fv_1, fv_2, \cdots, fv_N)$ requires multiplication of the client's feature vector of size N and the matrix M of size N by N to produce $EQ = Q \cdot M$. Thus encryption of the client's feature vector has $O(N^2)$ time complexity. In our tests N was set to 16 as there were 16 features in our dataset which the model was trained on. Thus while our tests vary the amount of decision nodes in the decision tree being converted, the amount of time needed to perform query encryption remained largely constant throughout our testing.

If we average the amount of time needed to perform query encryption across our initial trials we can estimate how long this operation takes for $N = 16$. Throughout our trials we consider converting trees with depths $d = \{1, 2, ...14\}$. The average amount of time needed to perform query encryption in our trials was $6.33000 \cdot 10^{-5}$ s.

7.3.2 $Q_{X,i}$ Creation

Similarly to the query encryption step above, creating $Q_{X,1}$ and $Q_{X,2}$, the vectors used by the cloud servers to perform classification depends on the complexity of

multiplication between the ciphertext vector EQ of length N, and the secret keys $SK_i = \mathbf{M}^{-1} \cdot \mathbf{A} \cdot K_i$, a matrix of size N by X. This operation has $O(NX)$ time complexity. Similar to the previous trials, $N = 16$ throughout our trials and we consider converting trees with depths $d = \{1, 2...14\}$. Since the cloud servers C_1 and C_2 can create their respective $Q_{X,i}$'s in parallel to each other, we simply average the amount of time taken for both servers to create their respective $Q_{X,i}$'s. The results of these tests can be seen in Figure 7.7.

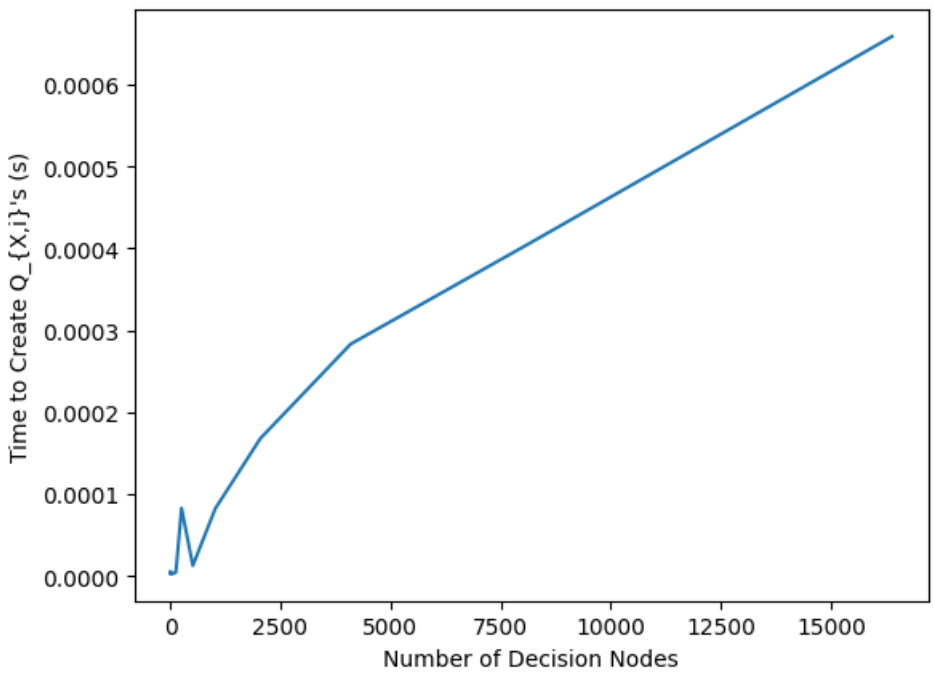

Figure 7.7: Number of Decision Nodes Versus Time to Create $Q_{X,i}$ (s)

7.3.3 Classification by Cloud Servers

Once the cloud servers have created their vectors $Q_{X,1}$ and $Q_{X,2}$ to be used in their classification process, each cloud server processes each Bloom filter node in their BF_i'. Thus for each cloud server C_i to create its bitstring S_i, it requires $O(X * k)$ time complexity. Similar to our previous trials, we vary the amount of nodes in the decision tree during our testing and have k set to 10. Since the cloud servers can also perform their classifications independently from each other, we once again average

the amount of time taken for both servers to create their respective S_i's in our trials. The results of these tests can be seen in Figure 7.8.

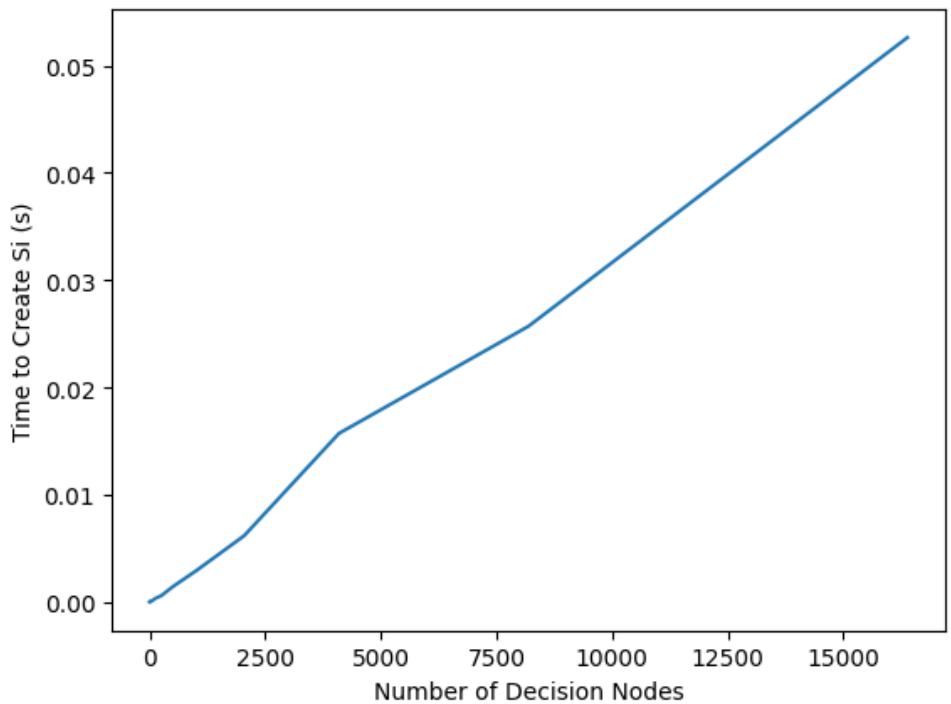

Figure 7.8: Number of Decision Nodes Versus Time to Create S_i (s)

Although we note that the time complexity for this operation was $O(X * k)$ it is possible that a cloud server can increase its performance by processing its Bloom filters in parallel since unlike a normal decision tree model, our Bloom filter based model allows for nodes to be processed in any order without the need to know what a previous node has output. Thus it is possible that the cloud servers may be able to gain additional performance gains by implementing parallelization of the processing of their Bloom filters. We save this improvement for future work.

7.4 Response Recovery

Once the client has received two bit strings from the cloud servers, they must first unshuffle the S_i's to produce $S_i' = S_i \cdot K_i^{-} - 1$. This operation has $O(X^2)$ time

complexity similar to shuffling the Bloom filters initially.

Next the client computes $S = S'_1 \oplus S'_2$, and determines the path the original decision tree took, which will take $O(\log_2(X))$ time complexity, since heap based path reconstruction for a decision tree only requires determining the path the tree would take from the root node of the tree to a leaf node. From the path the decision tree took, the client can determine the leaf node id S_{class} that the original decision tree would have reached in its classification of the client's query.

Finally, the client is able use the leaf node id, S_{class}, to determine the final classification of their query by seeing which Bloom filter BF_l contains S_{class}. This final step would take $O(|L| * k)$ for a decision tree with L distinct classes and k hash functions per Bloom filter BF_l. In our trials $|L| = 26$ since there are 26 possible labels that the decision tree can predict.

Similar to our earlier experiments we analyze the amount of time needed to perform response recovery for trees of various sizes. The results of these experiments can be seen in Figure 7.9.

7.5 Performance Summaries

After performing all of the experiments above, we summarize some of the trials in Table 7.2. This table details the amount of time needed to perform key steps of our scheme for trees of various sizes ranging from a tree with 8 nodes to a tree with 8192 nodes. During these trials, $fp = 0.001$, $k = 10$, $N = 16$, and $|L| = 26$.

Table 7.2: Time (s) to perform scheme steps for various sized decision trees

		8	16	128	1024	8192
Model Preparation	Decision Node Conversion	0.01001	0.02072	0.18701	1.61468	14.4492
	Leaf Node Conversion	0.00021	0.00021	0.00206	0.00604	0.03418
	Key Generation	0.00200	0.00962	0.10539	0.40252	28.4460
	Shuffling Bloom Filters	4.80000e-05	0.00011	0.00777	0.58652	36.2400
Query Request and Response	Query Encryption	9.19995e-06	8.19995e-06	2.24001e-05	2.69000e-05	1.88000e-05
	$Q_{X,i}$ Creation	7.19994e-06	4.99992e-06	9.10007e-06	0.00016	0.00081
	Classification by Cloud Servers	3.35500e-05	4.66750e-05	0.00036	0.00299	0.02571
Response Recovery	Response Recovery	0.00013	0.00015	0.00629	0.48088	37.9092
	Total	0.01392	0.03178	0.31090	3.09477	117.107

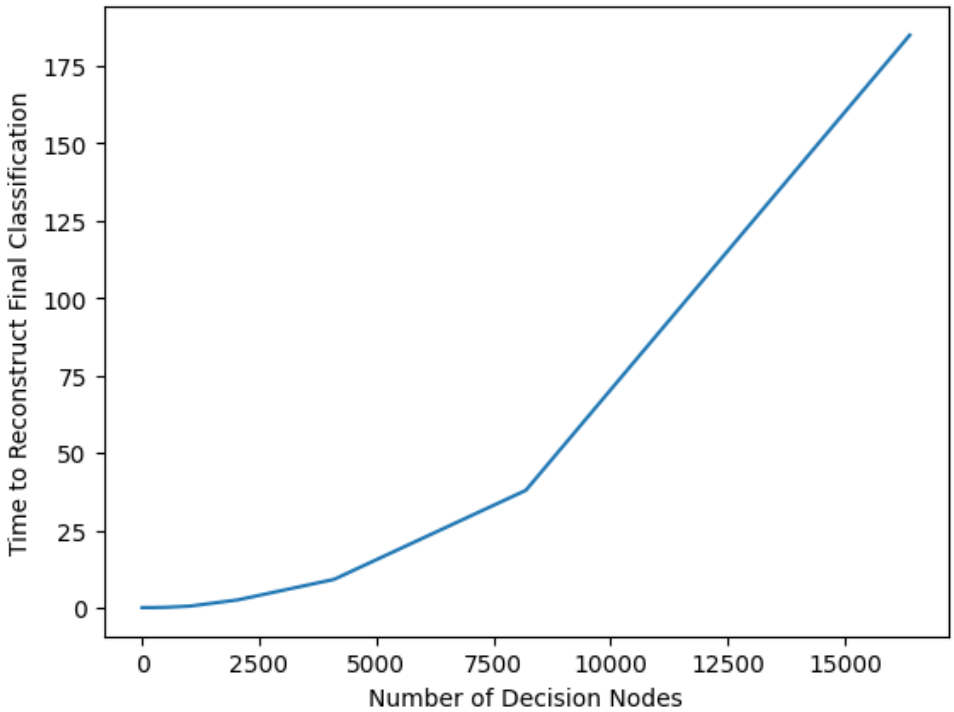

Figure 7.9: Number of Decision Nodes Versus Time to Perform Response Recovery (s)

We also compare the amount of time needed to perform classification for our scheme as well as for the original decision tree. For our scheme, the amount of time needed for classification is determined as the total amount of time taken for the query request and response phase as well as the response recovery phase. Overall we can see that from from our above analysis, the overall classification complexity is $O(N^2 + NX + Xk + X^2 + \log_2(X) + |L| * k)$. Within this the dominant terms are NX, N^2 and X^2, since Xk, $\log_2(X)$, and $|L| * k$ will generally be less than these terms. Thus classification complexity is more accurately $O(N^2 + NX + X^2)$. From this, the cloud servers have $O(NX)$ time complexity for classification and the client has $O(N^2 + X^2)$ time complexity for classification. The results of these tests can be seen in Table 7.3.

Table 7.3: Classification Times (s)

Number of Decision Nodes	Original Decision Tree	Our Scheme
8	0.00147	0.00018
16	0.00091	0.00021
128	0.00197	0.00669
1024	0.00095	0.48406
8192	0.00161	37.9357

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we have proposed a novel scheme to preserve decision tree privacy under a two-cloud server model. Unlike many other state of the art schemes [18, 19, 39, 40] our scheme does not make use of homomorphic encryption or additive secret sharing. We are able to preserve the privacy of the user's query, classification result, and the decision tree model with only Bloom filters and simple symmetric encryption. Furthermore we were able to keep the decision tree classification complexity to $O(NX)$ for the cloud servers and $O(N^2 + X^2)$ for the client while using our scheme.

Specifically, we developed a scheme which first converted the decision tree into two series of Bloom filters which mimicked the original decision tree structure and allowed for the classification process to be split amongst two servers. By doing so we ensured that even if a single cloud server decides to act maliciously, it cannot reconstruct the initial decision tree or retrieve a client's classification or query. After the servers process the client's encrypted query, the client can retrieve the final classification of their query without either cloud server knowing the final classification. So while our scheme's classification complexity is not as good as a traditional decsion tree, we are

able to prevent collusion attacks as well with our scheme while offering secure model inference.

Finally, we verified that our scheme was efficient through rigorous testing, and confirmed that the scheme is secure through security analysis. To our knowledge, our scheme is one of the first privacy preserving decision tree schemes which does not make use of additive secret sharing or homomorphic encryption to ensure model privacy, query privacy, and classification privacy.

8.2 Future Work

While our work was effective in preserving the privacy of a decision tree model while maintaining model accuracy, additional extensions to this work can be considered. In the following chapter we consider some of the limitations of our work and propose future research that could be caried out to tackle these limitations.

8.2.1 Machine Learning Model Attacks

In this thesis we assume that the client is an honest party. Due to this, we do not consider machine learning attacks such as model extraction attacks which sees clients try to continuously probe a machine learning model to see what data it is trained on or how the model is constructed. While we know that a client does not gain any useful information about the model when it makes a single prediction, we have not considered whether a client making repeated requests to the model could divulge useful information about the hosted model.

For example membership inference attacks may be possible in our scheme. In a membership inference attack, attackers hope to discover whether a particular query is part of the data that a model was trained on [14]. It is noted in [14, 36], that with enough membership inference attacks, an attacker could recreate the dataset that a

model was trained on with a high probability, which may allow them to recreate a machine learning model.

While we do mitigate against some of these membership inference attacks by only providing the client with the final classification result of a query and not with confidence scores detailing our confidence between possible prediction classes, [16] and [8] note that even when only label predictions are provided, model inference attacks are still possible.

In future work we could look at the information gained when a client repeatedly makes request to the cloud hosted model. Based on our findings we can make modifications to our scheme to prevent or reduce the risk of such attacks.

8.2.2 Allowing Float Values

In our scheme, we assume that all allowable values of all features is integer based, or if they are binary, or string based we first convert them to be integers. As such if a dataset has floating point valued features, we must first convert them to be interger based, which may result in some loss of precision or exceedingly large Bloom filters if the number of values in the domain gets too large. Future work could investigate whether it is possible to maintain the privacy preserving properties of our scheme while also allowing for floating point values to be acceptable.

8.2.3 Key Distribution

In our scheme we consider that all clients share a single set of keys. However, if these keys get leaked to the cloud servers, then the cloud server would be able to reconstruct our decision tree model if it can also collude with the other cloud server. An improvement to our scheme could involve making it possible for the decision tree model to be able to be encrypted using multiple keys which could be unique between users. By doing so we could make our model more secure, however a new problem

would be introduced as we would need to consider the task of key distribution which may make our scheme less efficient.

8.2.4 Model Explainability

In this work we noted that decision tree classifiers are often chosen for machine learning applications today since they are easily understood by the public. However by adding privacy preserving machine learning properties to our model, the decision tree model has lost its ability to provide model explainability. A future paper could investigate whether model explainability is possible with the use of our scheme or if model explainability simply contradicts our ability to provide privacy preserving decision tree models.

8.2.5 Extensions to Non-Full Trees

In this work we consider that all the decision tree classifiers must be full binary trees. However for many decision tree models, this may not be the case. Thus, by extending all decision trees to full binary trees in our scheme we must add additional dummy nodes which are not present in the original decision tree model. Thus our model adds additional complexity to decision trees which are not full binary trees. In future work, we could explore whether it is possible to extend our scheme to allow for decision tree models to be non-full binary trees. While it is likely that we could easily extend our scheme to allow for complete binary trees since heaps can allow us to store data about complete binary trees, it may require the cloud servers, and the client knowing additional structural information about the tree.

8.2.6 Extensions to Other Tree Based Models

In this work we consider using the decision tree classifiers due to their robustness and ability to provide quick classifications. However, since decision trees, and decision

nodes are the basis for many machine learning models such as Random Forest and AdaBoost, it is possible that we would be able to apply our scheme to these models as well.

However, due to the additional pieces of information required for these models, such as the weights of each tree in an AdaBoost model, we would need to modify our scheme to allow for such additional information. Similarly for Random Forest models, our work would need to consider the amount of space required to store each decision tree in the forest as Bloom filters. While this is important for a single decision tree as well, space efficiency becomes increasingly important for Random Forest models which contain multiple decision trees of various lengths. Future work could evaluate whether our scheme would still be efficient for privacy preservation of Random Forest, AdaBoost and other decision tree based models.

Bibliography

[1] Adi Akavia, Max Leibovich, Yehezkel S. Resheff, Roey Ron, Moni Shahar, and Margarita Vald, *Privacy-preserving decision trees training and prediction*, ACM Trans. Priv. Secur. **25** (2022), no. 3.

[2] Amazon, *What is machine learning*, https://aws.amazon.com/what-is/machine-learning, Accessed: 2023-5-25.

[3] John R. Anderson, Qingqing Huang, Walid Krichene, Steffen Rendle, and Li Zhang, *Superbloom: Bloom filter meets transformer*, CoRR **abs/2002.04723** (2020).

[4] Nima Asadi and Jimmy Lin, *Fast candidate generation for two-phase document ranking: Postings list intersection with bloom filters*, Proceedings of the 21st ACM International Conference on Information and Knowledge Management (New York, NY, USA), CIKM '12, Association for Computing Machinery, 2012, p. 24192422.

[5] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Commun. ACM **13** (1970), no. 7, 422426.

[6] Oswald Campesato, *Artificial intelligence, machine learning, and deep learning.*, Mercury Learning and Information, 2020.

[7] S. Carpov, N. Gama, M. Georgieva, and D. Jetchev, *Genoppml a framework for genomic privacy-preserving machine learning*, 2022 IEEE 15th International Conference on Cloud Computing (CLOUD) (Los Alamitos, CA, USA), IEEE Computer Society, jul 2022, pp. 532–542.

[8] Christopher A. Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot, *Label-only membership inference attacks*, Proceedings of the 38th International Conference on Machine Learning (Marina Meila and Tong Zhang, eds.), Proceedings of Machine Learning Research, vol. 139, PMLR, 18–24 Jul 2021, pp. 1964–1974.

[9] Zhenwei Dai and Anshumali Shrivastava, *Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier with application to real-time information filtering on the web*, Advances in Neural Information Processing Systems (H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, eds.), vol. 33, Curran Associates, Inc., 2020, pp. 11700–11710.

[10] Haokun Fang and Quan Qian, *Privacy preserving machine learning with homomorphic encryption and federated learning*, Future Internet **13** (2021), no. 4.

[11] IBM, *What is reinforcement learning*, https://developer.ibm.com/learningpaths/get-started-automated-ai-for-decision-making-api/what-is-automated-ai-for-decision-making, Accessed: 2023-5-25.

[12] ———, *What is unsupervised learning?*, https://www.ibm.com/topics/unsupervised-learning, Accessed: 2023-5-25.

[13] IEEE, *Types of homomorphic encryption*, https://digitalprivacy.ieee.org/publications/topics/types-of-homomorphic-encryption, Accessed: 2023-5-25.

[14] Md Shohidul Islam, Behnam Omidi, Ihsen Alouani, and Khaled N. Khasawneh, *Vpp: Privacy preserving machine learning via undervolting*, 2023 IEEE Inter-

national Symposium on Hardware Oriented Security and Trust (HOST), 2023, pp. 315–325.

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis, *The case for learned index structures*, Proceedings of the 2018 International Conference on Management of Data (New York, NY, USA), SIGMOD '18, Association for Computing Machinery, 2018, p. 489504.

[16] Zheng Li and Yang Zhang, *Membership leakage in label-only exposures*, Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA), CCS '21, Association for Computing Machinery, 2021, p. 880895.

[17] Jinwen Liang, Zheng Qin, Sheng Xiao, Lu Ou, and Xiaodong Lin, *Efficient and secure decision tree classification for cloud-assisted online diagnosis services*, IEEE Transactions on Dependable and Secure Computing **18** (2021), no. 4, 1632–1644.

[18] Qianying Liao, Bruno Cabral, Joo Paulo Fernandes, and Nuno Loureno, *Herb: Privacy-preserving random forest with partially homomorphic encryption*, 2022 International Joint Conference on Neural Networks (IJCNN), 2022, pp. 1–10.

[19] Qianying Liao, Alexandre Cortez Santos, Bruno Cabral, Joo Paulo Fernandes, and Nuno Loureno, *Herb+: Evolving an industrial-strength privacy-preserving machine learning framework*, 2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC), 2022, pp. 212–223.

[20] MathWorks, *What is reinforcement learning?*, https://es.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html, Accessed: 2023-5-25.

[21] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan, *A survey on bias and fairness in machine learning*, ACM Comput. Surv. **54** (2021), no. 6.

[22] Microsoft, *Visually represent your azure architecture using the latest shapes in visio for the web*, https://techcommunity.microsoft.com/t5/microsoft-365-blog/visually-represent-your-azure-architecture-using-the-latest/ba-p/1634509, Accessed: 2023-5-20.

[23] _____, *What is cloud computing*, https://azure.microsoft.com/en-ca/resources/cloud-computing-dictionary/what-is-cloud-computing, Accessed: 2023-5-20.

[24] Michael Mitzenmacher, *A model for learned bloom filters, and optimizing by sandwiching*, Proceedings of the 32nd International Conference on Neural Information Processing Systems (Red Hook, NY, USA), NIPS'18, Curran Associates Inc., 2018, p. 462471.

[25] Payman Mohassel and Yupeng Zhang, *Secureml: A system for scalable privacy-preserving machine learning*, 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 19–38.

[26] Kseniia Nikolskaia and Victor Naumov, *Ethical and legal principles of publishing open source dual-purpose machine learning algorithms*, 2020 International Conference Quality Management, Transport and Information Security, Information Technologies, 2020, pp. 56–58.

[27] Ripon Patgiri, Anupam Biswas, and Sabuzima Nayak, *deepbf: Malicious url detection using learned bloom filter and evolutionary deep learning*, Computer Communications **200** (2023), 30–41.

[28] Junjie Peng, Elizabeth Jury, Pierre Dnnes, and Coziana Ciurtin, *Machine learning techniques for personalised medicine approaches in immune-mediated chronic inflammatory diseases: Applications and challenges*, Frontiers in Pharmacology **12** (2021).

[29] Saman Rizvi, Bart Rienties, and Shakeel Ahmed Khoja, *The role of demographics in online learning; a decision tree based approach*, Computers and Education **137** (2019), 32–47.

[30] David Slate, *Letter Recognition*, UCI Machine Learning Repository, 1991, DOI: https://doi.org/10.24432/C5ZP40.

[31] Statista, *Total data volume worldwide 2010-2025*, https://www.statista.com/statistics/871513/worldwide-data-created/, Accessed: 2023-5-25.

[32] Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow, *Privacy-preserving decision trees evaluation via linear functions*, Computer Security – ESORICS 2017 (Cham) (Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, eds.), Springer International Publishing, 2017, pp. 494–512.

[33] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu, *Cryptgpu: Fast privacy-preserving machine learning on the gpu*, 2021.

[34] David J. Wu, Tony Feng, Michael Naehrig, and Kristin Lauter, *Privately evaluating decision trees and random forests*, Cryptology ePrint Archive, Paper 2015/386, 2015.

[35] Yingying Yao, Zhendong Zhao, Xiaolin Chang, Jelena Mii, Vojislav B. Mii, and Jianhua Wang, *A novel privacy-preserving neural network computing approach for e-health information system*, ICC 2021 - IEEE International Conference on Communications, 2021, pp. 1–6.

[36] Zuobin Ying, Yun Zhang, and Ximeng Liu, *Privacy-preserving in defending against membership inference attacks*, Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice (New York, NY, USA), PPMLP'20, Association for Computing Machinery, 2020, p. 6163.

[37] Shuai Yuan, Hongwei Li, Rui Zhang, Meng Hao, Yiran Li, and Rongxing Lu, *Towards lightweight and efficient distributed intrusion detection framework*, 2021 IEEE Global Communications Conference (GLOBECOM), 2021, pp. 1–6.

[38] Mingwu Zhang, Yu Chen, and Willy Susilo, *Decision tree evaluation on sensitive datasets for secure e-healthcare systems*, IEEE Transactions on Dependable and Secure Computing (2022), 1–14.

[39] Yifeng Zheng, Huayi Duan, Cong Wang, Ruochen Wang, and Surya Nepal, *Securely and efficiently outsourcing decision tree inference*, IEEE Transactions on Dependable and Secure Computing **19** (2022), 1841–1855.

[40] Yifeng Zheng, Cong Wang, Ruochen Wang, Huayi Duan, and Surya Nepal, *Optimizing secure decision tree inference outsourcing*, IEEE Transactions on Dependable and Secure Computing (2022), 1841–1855.

Appendix A

Code

A.1 Bloom Tree Module

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from numpy.linalg import inv
import math
import numpy as np
from numpy.linalg import inv
import math
import random
import pybloom_live
import random

def coin_flip():
    '''Represents a coin flip. Returns False for tails, True
        for heads.
    '''
    random.seed()
    return bool(random.randint(0,1))

def bloomify_node(min_val, max_val, threshold, error_rate):
    '''Creates two Bloom Filters as per the description in
       the algorithm presented in the paper.
    '''
```

```
    capacity = max_val - min_val + 1

    bloom_filter_1 = pybloom_live.BloomFilter(capacity=capacity, error_rate=error_rate)
    bloom_filter_2 = pybloom_live.BloomFilter(capacity=capacity, error_rate=error_rate)

    added_to_bf1 = []
    added_to_bf2 = []

    for i in range(int(min_val), int(threshold + 1)):
        if coin_flip():
            pass
        else:
            bloom_filter_1.add(i)
            bloom_filter_2.add(i)
            added_to_bf1.append(i)
            added_to_bf2.append(i)


    for i in range(int(threshold + 1), int(max_val)):
        if coin_flip():
            bloom_filter_1.add(i)
            added_to_bf1.append(i)
        else:
            bloom_filter_2.add(i)
            added_to_bf2.append(i)

    return [bloom_filter_1, bloom_filter_2]


def bloomify_leaf_nodes(leaf_node_class_dict, error_rate, classifier):
    '''Creates the bloom filters required for final classification {BF_L}. Here the list is returned as a list of tuples where the
    first element in the tuple is the label L_i and the second element in the tuple is BF_L.
    '''
    max_depth = classifier.tree_.max_depth
    capacity = 2 ** (max_depth)


    bloom_list = []
    for key in leaf_node_class_dict:
        bloom_list.append((key, bloomify_leaf_node(leaf_node_class_dict[key], capacity, error_rate))
        )
```

```
    return bloom_list


def bloomify_leaf_node(leaf_node_ids, capacity, error_rate):
    '''Converts leaf nodes into bloom filters as described in the paper.
    This is done by checking the leaf node ids in the tree and checking the label associated with it to create a bloom filter.
    '''
    leaf_filter  = pybloom_live.BloomFilter(capacity, error_rate)
    for id in leaf_node_ids:
        leaf_filter.add(id)
    return leaf_filter


def bloomify_tree(classifier, min_vals, max_vals, error_rate, classes):
    '''Converts all of the nodes in the original classifier into bloom filters as described in the paper. As well,
    generates the set of bloom filters {BF_L} for classification.
    Returns a dictionary containing BF_i the bloom filters to be sent to each cloud server C_i and the LeafNodeDict the dictionary containing which leaf nodes provide which classification.
    '''
    num_nodes = classifier.tree_.node_count
    max_depth = classifier.tree_.max_depth
    decision_nodes = 2 **(max_depth) - 1
    classes = classifier.classes_
    children_left = classifier.tree_.children_left
    children_right = classifier.tree_.children_right

    leaf_node_class_dict = {}
    for label in classes:
        leaf_node_class_dict[label] = []


    stack = [(0,0, None)]
    bloom_filters_1 = [None] * decision_nodes
    bloom_filters_2 = [None] * decision_nodes
    feature_heap = [None] * decision_nodes
    leaf_node_id = 0
    heap_id = 0
```

```
103
104     while len(stack) != 0:
105         node_id, depth, label = stack.pop(0)
106         if node_id is not None:
107             is_split_node = children_left[node_id] !=
                    children_right[node_id]
108
109             # If a split node, append left and right
                    children and depth to `stack`
110             # so we can loop through them
111             if is_split_node:
112                 stack.append((classifier.tree_.children_left
                        [node_id], depth + 1, None))
113                 stack.append((classifier.tree_.
                        children_right[node_id], depth + 1, None)
                        )
114
115                 #Feature tested
116                 feature = classifier.tree_.feature[node_id]
117
118                 #Create Bloom Filters
119                 filters = bloomify_node(min_vals[feature],
                        max_vals[feature], classifier.tree_.
                        threshold[node_id], error_rate)
120                 bloom_filters_1[heap_id] = filters[0]
121                 bloom_filters_2[heap_id] = filters[1]
122                 feature_heap[heap_id] = feature
123                 heap_id+=1
124             else:
125                 #Add node to the list of classifications
126                 value = classifier.tree_.value[node_id]
127                 label = classifier.classes_[np.argmax(value)
                        ]
128                 if depth < max_depth:
129                     dummy_min_max = random.randint(0, len(
                            min_vals)-1)
130                     dummy_threshold = random.randint(
                            min_vals[dummy_min_max], max_vals[
                            dummy_min_max])
131                     filters = bloomify_node(min_vals[
                            dummy_min_max], max_vals[
                            dummy_min_max], dummy_threshold,
                            error_rate)
132                     bloom_filters_1[heap_id] = filters[0]
133                     bloom_filters_2[heap_id] = filters[1]
134                     heap_id += 1
135                     #Extend tree with dummy nodes
136                     #Check classification and put it into a
```

```
                        list
                    stack.append((None, depth + 1, label))
                    stack.append((None, depth + 1, label))
                else:
                    leaf_node_class_dict[label].append(leaf_node_id)
                    leaf_node_id += 1
        else:
            if depth < max_depth:
                dummy_min_max = random.randint(0, len(min_vals)-1)
                dummy_threshold = random.randint(min_vals[dummy_min_max], max_vals[dummy_min_max])
                filters = bloomify_node(min_vals[dummy_min_max], max_vals[dummy_min_max], dummy_threshold, error_rate)
                bloom_filters_1[heap_id] = filters[0]
                bloom_filters_2[heap_id] = filters[1]
                heap_id += 1
                #Extend tree with dummy nodes
                #Check classification and put it into a list
                stack.append((None, depth + 1, label))
                stack.append((None, depth + 1, label))
            else:
                leaf_node_class_dict[label].append(leaf_node_id)
                leaf_node_id += 1


    return {"BF1": bloom_filters_1,
            "BF2": bloom_filters_2,
            "FeatHeap": feature_heap,
            "LeafNodeDict": leaf_node_class_dict}

def encrypt_query(query, M):
    '''Encrypts a query using a key M. Returns the encrypted value query*M.
    '''
    return np.dot(query, M)

def create_f_prime(C, key):
    '''Creates the new feature vector f' using the ciphertext C and a key. Returns the C*key.
    '''
    return np.dot(C, key)

def generate_keys(feature_heap, k, num_features, seed=None):
    ''' Generates the keys M, and M^-1*A based on the
```

```
        feature heap created during the initial
        bloomification process.
    If there are any dummy values such that the feature
        vector being asked for by the provider then k
        represents the number of dummy features
    in additon to the normal n features that would be asked
        for.
    '''
    M = np.random.rand(num_features + k, num_features + k)
    mx = np.sum(np.abs(M), axis=1)
    np.fill_diagonal(M, mx)
    M_inverse = inv(M)

    if seed == None:
        random.seed()
    else:
        random.seed(seed)

    # Creates A matrix to extract information such that the
        correct features would be extracted from the original
        feature vector.
    A = np.zeros((num_features + k, len(feature_heap)))
    for i in range(0, len(feature_heap)):
        if(feature_heap[i] is not None):
            A[feature_heap[i]][i] = 1
        else:
            #Use parent value for the feature being
                extracted if the node was a leaf node.
            A[:, i] = A[:, int(math.floor(i-1)/2)]


    K1 = np.zeros((len(feature_heap), len(feature_heap)))
    columns_to_insert = list(range(0,len(feature_heap)))
    column = 0
    while (len(columns_to_insert) != 0):
        removed = random.choice(columns_to_insert)
        columns_to_insert.remove(removed)
        K1[removed][column] = 1
        column+=1


    K2 = np.zeros((len(feature_heap), len(feature_heap)))
    columns_to_insert = list(range(0,len(feature_heap)))
    column = 0
    while (len(columns_to_insert) != 0):
        removed = random.choice(columns_to_insert)
        columns_to_insert.remove(removed)
        K2[removed][column] = 1
```

```
        column+=1

    return {"M":M,
            "K1":K1,
            "K2":K2,
            "K1_inv":inv(K1),
            "K2_inv":inv(K2),
            "Cloud_Key_1":np.dot(np.dot(M_inverse, A), K1),
            "Cloud_Key_2":np.dot(np.dot(M_inverse, A), K2),
            "Cloud_Key": np.dot(M_inverse, A),
            }


def shuffle_BF(BF_list, K):
    ''' Shuffles the Bloom Filters based on the key K and returns the shuffled Bloom Filters
    '''
    BF_shuffle = [None] * len(BF_list)
    #Go through each column of the matrix and find the one, then select correct one
    for j in range(K.shape[1]):
        for i in range(K.shape[0]):
            if K[i][j] == 1:
                BF_shuffle[j] = BF_list[i]
    return BF_shuffle



def classify(BF_list, f_prime):
    ''' Generates a string s that is the output of the Bloom Filter based classification
    '''
    s_i = []
    for i in range(0, len(BF_list)):
        if BF_list[i] is not None and int(round(f_prime[i])) in BF_list[i]:
            s_i.append(1)
        else:
            s_i.append(0)
    return s_i

def reconstruct_classification(s1, s2, BFL, K1_inv, K2_inv):
    ''' Reconstructs the classification that would have been output by the original tree based on the strings s1 and s2 output by the servers
    and the 2 secret keys which are used for decryption.
    '''
    unshuffled_s1 = [None] * K1_inv.shape[1]
```

```
    for j in range(K1_inv.shape[1]):
        for i in range(K1_inv.shape[0]):
            if K1_inv[i][j] == 1:
                unshuffled_s1[j] = s1[i]
    unshuffled_s2 = [None] * K2_inv.shape[1]
    for j in range(K2_inv.shape[1]):
        for i in range(K2_inv.shape[0]):
            if K2_inv[i][j] == 1:
                unshuffled_s2[j] = s2[i]


    s = [b1 ^ b2 for (b1, b2) in zip(unshuffled_s1,
       unshuffled_s2)]
    heap_string = []

    # Go through s in heap order
    index = 0
    while(index < len(s)):
        if(index==0):
            heap_string.append(s[0])
        else:
            heap_string.append(s[index])
        index = 2*index+1+s[index]

    leaf_id = sum([2**(len(heap_string)-i-1)*heap_string[i]
       for i in range(0, len(heap_string))])

    for i in range(0, len(BFL)):
        if leaf_id in BFL[i][1]:
            return BFL[i][0]
    return f"Failure: {leaf_id} was the leaf calculated"
```

Vita

Candidate's full name: Sean Richard Dimalouw Lalla

Bachelor of Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada, 2022.

Bachelor of Science, University of New Brunswick, Fredericton, New Brunswick, Canada, 2022.

Master of Computer Science, University of New Brunswick, Fredericton, Canada, Expected May 2024.

Publications:

1. S. Zhang, S. Ray, R. Lu, Y. Guan and S. Lalla, Traceable and Privacy-Preserving Worker Selection Scheme with Arbitrary Spatial Ranges in MCS, Proc. IEEE ICC'23, Rome, Italy, May 28-June 1, 2023.

2. Y. Guan, S. Zhang, R. Lu, and S. Lalla, Efficient and Privacy-Preserving Subgraph Matching Queries in Graph Federation, Proc. IEEE ICC'23, Rome, Italy, May 28-June 1, 2023.

3. S. Lalla, R. Lu, Y. Guan, S. Zhang, Improving Decision Tree Privacy with Bloom Filters, Proc. IEEE GlobeCom'23, Kuala Lumpur, Malaysia, December 4-8, 2023.

Conference Presentations:
N/A

Posters:

1. S. Lalla, R. Lu, Y. Guan, S. Zhang, Improving Decision Tree Privacy with Bloom Filters, Research Exposition 2023, Fredericton, Canada, April 14, 2023.