Secure multiparty computation for privacy preserving data mining

Chen, P.

Award date:
2012

Master's Thesis

Secure Multiparty Computation for Privacy Preserving Data Mining

Ping Chen

EINDHOVEN UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Master's Thesis

Secure Multiparty Computation for Privacy Preserving Data Mining

by

Ping Chen

Supervisors:
dr.ir. L.A.M. (Berry) Schoenmakers
prof.dr.ir. H.A.M. (Hennie) Daniels
ir. S.J.A. (Sebastiaan) de Hoogh

August 2012

Acknowledgments

This thesis is the result of my internship at Erasmus University Rotterdam, as part of the the EU-FP7 project CASSANDRA. I would like to thank professor Hennie Daniels for giving such an opportunity to perform an interesting and challenging master's thesis project.

I am very grateful to my supervisor Berry Schoenmakers at Eindhoven University of Technology, for the guidance, questions and answers, and his humor. Special thanks go to Berry Schoenmakers and Sebastiaan de Hoogh, for their profound knowledge in cryptography and providing guidance and support to my work. And with gratitude to Hennie Daniels and Lingzhe Liu, for managing my internship, and inspiring me some ideas in data mining.

I enjoyed my study and life in the Netherlands, thanks to Eindhoven University of Technology and the Kerckhoffs Institute, for providing the 2-year master program in Information Security. And with special thanks to het Koninklijk Concertgebouworkest, for their fantastic and wonderful performances which give me serenity, comfort and enjoyment.

Writing thesis is not easy for me, I am thankful to Sebastiaan de Hoogh and Boris Škorić, for their helpful comments to my thesis. And I would like to thank my friend Tingting Cao, for always encouraging me during the project. Finally and most importantly, I want to thank my parents for all their love and support over the years.

Ping Chen
August 2012

Contents

Chapter 1

Introduction

Consider a supply chain, in which a number of parties collaborate with each other for moving a product or service from supplier to customer. For better supply chain management (optimizing the benefit of the supply chain), it is essential to share data among supply chain partners. Data sharing increases the visibility of a supply chain, hence allows for cost-efficient risk management for both business companies and customs authorities. However, as data is part of partners' trade secret, they are usually reluctant to disclose their data. Furthermore, if this data contains customers' sensitive information, it is prohibited by legal acts to disclose that information.

To enable collecting and analyzing large data sets from supply chain partners, without compromising the confidentiality of the data, privacy-preserving data mining solutions can be used. Usually the data collected is not for the sake of having the individual records, but rather for characterizing the whole data set and generating an overall result, by using various data mining algorithms. Privacy-preserving data mining considers the problem of running data mining algorithms on confidential data that is not supposed to be revealed, even to the party running the algorithm [LP09].

Randomization method and anonymization method are two common techniques for privacy-preserving data mining, which can help protect the confidentiality of data, however they introduce a trade-off between information loss and privacy [AY08]. In practise, it is difficult to strike the right balance to get satisfied result.

Secure Multiparty Computation (SMC) is a technique from modern cryptography that can be used to achieve privacy-preserving data mining. Secure multiparty computation refers to cryptographic protocols that allow a set of parties to perform a computation on their private inputs in such a way that the parties only learn the correct output and nothing else, hence their privacy is preserved.

This thesis is mostly about deploying secure multiparty computation for privacy-preserving data mining. As a case study, we address the problem of secure frequent itemset mining and secure decision tree learning. The research was part of the EU-FP7 project CASSANDRA which aims to make container security more efficient and effective through secure data sharing [CAS12].

1.1 Cryptographic Concepts

The concept of secure multiparty computation was formally introduced in 1982 by Yao as secure two-party computation through the millionaires problem, which allowed two parties to compare their wealth (to see who is the richer one) without revealing anything else about each other's wealth [Yao82]. In general, there are a number of parties $P_1, P_2, ..., P_n$, with respective input values $x_1, x_2, ..., x_n$, a SMC protocol enables them to jointly evaluate $f(x_1, x_2, ..., x_n)$ for some function f, in such a way that each party obtains the correct output value and no information leaks on the input values of the honest parties beyond what is implied logically from the output values and the input values of the corrupted parties [Sch11].

There are inherent limitations regarding the functions that can be computed without leaking information about their inputs. For example, to securely compute the average over distributed inputs: $f(x_1, x_2, ..., x_n) = (\sum_{i=1}^{n} x_i)/n$, if $n = 2$, then each party can determine the input of the other party, so input privacy is not possible. Also, if $n = 3$, two parties can collude and learn the input of the third one.

Secure multiparty computation protocols can be designed to be secure against either passive adversaries or active adversaries. The adversary represents the coalition formed by an attacker and/or one or more of the parties taking part in the protocol. The parties under the control of the adversary are said to be corrupted. An adversary is passive or semi-honest if the corrupt parties do not interfere with the execution of the protocol, but eavesdrop on the communication between the parties and try to learn as much as possible from the data collected. An active adversary can make the corrupted parties to behave arbitrarily, and interfere with the communication by deleting, injecting, or modifying messages [Sch12].

An adversary can also be categorized as static or adaptive, from the point of view of the corruption strategy. A static adversary cannot corrupt parties during protocol execution, hence the set of corrupted parties is fixed from the start of the protocol. The adversary is adaptive if the parties can be corrupted during protocol execution. Logically, designing secure computation protocols against an active and adaptive adversary is much harder than against a passive and static adversary.

An important aspect of secure multiparty computation is the underlying communication model, which describes how messages are transmitted among parties and what an adversary is allowed to observe. Generally, there are two different settings. In the cryptographic setting, the parties communicate via a broadcast channel, where all parties are able to see the messages exchanged, but the adversary has bounded computational resources (restricted to be a probabilistic polynomial time algorithm). In the information theoretic setting, the parties are connected via private channels, which allow the parties to send each other messages that are not visible to the adversary, and the adversary may be unlimited powerful.

1.2 Related Work

Although the general theory of secure multiparty computation has been created in the 1980s, and it was proven that secure multiparty computation is feasible for any computable function [Yao82, CGMA85, GMW87, BGW88, RB89], secure multiparty computation was not efficient to be able to solve complex functions in a reasonable amount of time. To apply the theoretical result in real life, protocols must be designed that are highly efficient. Nowadays, efficiency of secure multiparty computation is an important topic of cryptographic research, and there are a number of practical results appeared in [Tof07, BCD$^+$09, Gei10, DK10, Hoo12].

The first real life application of secure multiparty computation is presented in [BCD$^+$09], where about 1200 Danish farmers participated in a auction system built using secure multiparty computation protocols to determine the market price of their sugarbeets. The computation took about 15 minutes using three laptops connected by a LAN.

Secure multiparty computation solution for secure linear programming is reported in [Hoo12], where they designed large scale application to solve linear programming problem in the context of collaborative supply chain management. The computation for solving linear programs with about 300 variables and 200 constraints requires about 1 day on an ordinary PC.

The idea to use secure multiparty computation for privacy-preserving data mining was first appeared in [LP00], where Lindell and Pinkas proposed a secure two-party protocol for building decision tree over horizontally partitioned data, based on the ID3 algorithm and oblivious transfer protocol. The core block of the protocol is securely computing logarithm function, using Taylor approximation and oblivious polynomial evaluation.

After Lindell and Pinkas's work, there has been growing interest in using secure multiparty computation for privacy-preserving data mining. A generalized privacy-preserving variant of the ID3 algorithm for vertically partitioned data is presented in [VCKP08], and privacy-preserving classification using Naïve Bayes classifier for both horizontally partitioned and vertically partitioned data is proposed in [VKC08]. Other privacy-preserving data mining solutions for horizontally partitioned data using secure multiparty computation includes association rule mining [KC04], and clustering [JW05, IKS+07].

Most of the above solutions are based on oblivious transfer protocol [GMW87, Kil88], and the $\ln x$ protocol proposed by Lindell and Pinkas in [LP00] is reused as a building block. In these solutions, a two-party case was considered first, and the multiparty case with more than two parties is achieved by reducing the problem to two-party cases, thus they are not efficient for large scale applications.

Privacy-preserving data mining using secure multiparty computation for solving real-life problems is first demonstrated in [BTW12], where a secure data aggregation system was built for jointly collecting and analyzing financial data from a number of Estonian ICT companies. The application was deployed in the beginning of 2011 and is still in continuous use. However, their data analysis are limited to basic data mining operations, such as sorting, filtering.

1.3 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Secure Multiparty Computation**
  This chapter gives a brief introduction of the security model for secure multiparty computation. We distinguish 'perfect' security and 'statistical' security for our protocols. With respect to secure multiparty computation, our protocols are based on threshold secret sharing schemes, and we use the VIFF to implement our protocols.

- **Chapter 3: Basic Protocols**
  This chapter presents an overview of basic protocols that are served as building blocks for the protocols in Chapter 4 and Chapter 5. Most of the protocols are standard in literature, as a small contribution, we proposed new protocols for equality test, and symmetric AND/OR function, which is more efficient in some cases than existing protocols.

- **Chapter 4: Secure Frequent Itemset Mining**
  This chapter proposes two privacy preserving itemset mining solutions using secure multiparty computation. We base our protocols on the Apriori algorithm. A secure Apriori protocol which outputs public frequent itemsets, and a fully secure Apriori protocol which outputs secret result are given.

- **Chapter 5: Secure Decision Tree Learning**
  This chapter proposes a secure ID3 protocol for privacy preserving decision tree learning, based on the ID3 algorithm. To avoid the secure computation of logarithm function, we use the Gini Index and χ^2 statistic as splitting measures in secure ID3 protocol. In our benchmark experiment, we compare the performance of the Gini Index and χ^2 statistic. In addition, we introduce double field setting to improve the performance of secure ID3 protocol.

Chapter 2

Secure Multiparty Computation

In this chapter, we first introduce some basic concepts about the security of cryptographic protocols that are used in this thesis, and then present the secret sharing schemes based secure multiparty computation framework in Section 2.2. Section 2.3 introduces the Virtual Ideal Functionality Framework that are used to implement our protocols, and the experiment settings for benchmark analysis.

2.1 Security Model

Defining the security of protocols is a fundamental and complicated problem in cryptography, which involves a lot of technical details. We refer to [MR92, Bea92, LP09] for the formal definition of security. Roughly speaking, security against some adversary means that all information gained by the adversary can be simulated by a simulator using only information that the adversary is allowed to know. In other words, the protocol is said to be secure if the adversary cannot distinguish the simulated messages from messages it would have received from the parties that are not corrupt during protocol execution.

In this thesis, we use 'perfect' secure and 'statistical' secure to describe the security of secure multiparty computation protocols. The concepts are based on the definition of indistinguishability.

2.1.1 Indistinguishability

Definition 2.1 *Let X and Y be two random variables, both taking values in some finite set V. The* ***statistical distance*** *between X and Y is defined as*

$$\Delta(X;Y) = \frac{1}{2}\sum_{v\in V} |\Pr[X = v] - \Pr[Y = v]|$$

Definition 2.2 *A non-negative function $f : \mathbb{N} \to \mathbb{R}$ is called* ***negligible*** *if for every positive polynomial p there exists a $k_0 \in \mathbb{N}$ such that for all $k \geqslant k_0, f(k) \leqslant 1/p(k)$.*

Definition 2.3 *Let $X = \{X_i\}_{i\in I}$ and $Y = \{Y_i\}_{i\in I}$ be two probability ensembles, indexed by I. A probability ensemble is a set of probability distributions or random variables. Suppose that $|X_i| = |Y_i|$ for all $i \in I$, and that these sizes are polynomial in $|i|$. Then X and Y are said to be:*

perfectly indistinguishable *if $\Delta(X_i; Y_i) = 0$ for all $i \in I$ (hence identically distributed).*

statistically indistinguishable *if $\Delta(X_i; Y_i)$ is negligible as a function of $|i|$.*

computationally indistinguishable *if for all probabilistic polynomial time algorithms D, we have that*

$$|\Pr[D(X_i) = 1] - \Pr[D(Y_i) = 1]|$$

is negligible as a function of $|i|$.

Based on the above definitions, we say a protocol is ‘perfect’ secure if the the simulated messages and messages obtained by the adversary are perfectly indistinguishable, and ‘statistical’ secure means that the simulated messages and messages obtained by the adversary are statistically indistinguishable.

2.1.2 Composability

Usually the design of protocols for a complex task uses the so called modular protocol composition, in which the complex task are divided into several simpler subtasks that are realized by small sub-protocols. Several sub-protocols can be secure in the stand-alone setting, however the protocol composed by the sub-protocols may not be secure. To prove the security of a complex protocol, the framework of Universal Composability (UC) can be used [Can01].

Without going into too much technical details, the main UC theorem says that a protocol remains secure, if it is composed with an arbitrary set of universal composable protocols. Hence to design a protocol that securely realizes a complex task, we can divide the task into subtasks, and design secure sub-protocols which are universal composable for each subtask as building blocks.

2.1.3 Complexity Analysis

To analyze the performance of secure multiparty computation protocols theoretically, we calculate the round complexity and communication complexity.

- **Round Complexity.** A protocol can be divided logical units called rounds. A round is a phase where parties need to wait for messages in order to be able to continue computation. For example, in a protocol round r, each party receives messages from the other parties (sent in round $r-1$), performs a local computation, and then sends messages to the other parties (to be processed in round $r+1$). The round complexity of a protocol is the number of rounds necessary to complete a protocol run.

- **Communication Complexity.** The communication complexity defines the amount of data transmitted during a protocol run. We call the amount of data sent by each party in a multiplication protocol (see Protocol 3.1) an invocation, and the communication complexity is measured by the number of invocation during a protocol run

In general, the overall protocol running time is dominated by the communication time, an estimate of the overall communication time can be obtained by combining the Communication Complexity (the amount of data sent) and Round Complexity (the number of rounds).

2.2 Basic Framework

In general, there are three different frameworks to achieve secure computation, Secret Sharing Schemes [Sha79, RB89], Threshold Homomorphic Cryptosystems [CDN01, DN03], Oblivious

Transfer Protocols [GMW87, Kil88]. This section introduces the secure multiparty computation framework based on Secret Sharing Schemes.

2.2.1 Secret Sharing Scheme

A secret sharing scheme allows one party, called dealer, to share a secret among a set of parties, in such a way that only some specified subsets of parties can recover the secret, while others have no information about it. More formally, let $P = \{P_1, ..., P_n\}$ be the set of all parties, including the dealer, the idea is to split a secret s into several shares, such that each party P_i get a share denoted by $[s]_i$ ($[s]$ will be used to indicate a shared secret s), and the secret can be reconstructed whenever a sufficient number of shares is available, but it is not possible to get any information about the secret if an insufficient number of shares is available.

Secret sharing was introduced independently by A. Shamir [Sha79] and G.R. Blakley [Bla79]. Let $P = \{P_1, ..., P_n\}$ be the set of all parties, including a dealer D, and $\mathcal{P}(P)$ be the power set of P. A secret sharing scheme consists of two phases (and protocols):

- **Distribution.** A protocol in which the dealer D distributes a secret s by computing the sharing function to obtain a list of shares $([s]_1, ..., [s]_n)$, and sending to each party P_i its share $[s]_i$ on a private channel, for $i \leqslant n$.

- **Reconstruction.** A protocol in which the secret s is recovered by collecting the shares from a qualified set, and computing the reconstruction function to obtain the secret (this is often done in parallel by all parties).

A qualified set is a set of parties that is allowed to reconstruct the secret together. The collection of all qualified sets $\Gamma \subseteq \mathcal{P}(P)$ is called access structure. An access structure is monotone if for every $A \in \Gamma$, if $A \subset B \subseteq P$ then $B \in \Gamma$ (i.e., closed under taking supersets).

A forbidden set is a set of parties that should not obtain any information by combining their shares. The collection of all forbidden sets $\Delta \subseteq \mathcal{P}(P)$ is called adversary structure. An adversary structure is monotone if for every $A \in \Delta$ and $B \subset A$ it holds that $B \in \Delta$ (i.e., closed under taking subsets).

If a secret sharing scheme has a monotone access structure Γ, and a monotone adversary structure Δ, it is called perfect. A perfect secret sharing scheme satisfies: $\Delta = \bar{\Gamma} = \mathcal{P}(P) - \Gamma$. This is the typical situation in practice, meaning that the adversary can corrupt any set of parties $A \in \Delta$, while the other parties $\bar{A} \in \Gamma$ are honest.

A secret sharing scheme satisfies the privacy and correctness property:

- **Privacy.** If the dealer D remains honest, the parties of any set $A \in \Delta$ learn nothing about the secret s after the distribution phase.

- **Correctness.** In the reconstruction phase, the parties of any set $A \in \Gamma$ can recover the secret.

A secret sharing scheme is called linear if the shares are a linear function of the secret and randomly chosen values. A Linear Secret Sharing (LSS) Scheme has the nice property that any linear combination of shared secrets can be locally computed as a linear combination of their individual shares, hence there is no communication cost. In the next section, we will introduce an efficient LSS scheme: Shamir's Secret Sharing.

2.2.2 Shamir's Secret Sharing

A. Shamir proposed a simple and efficient perfect linear (t, n)-threshold secret sharing scheme in [Sha79], where n is the number of parties, $t(1 \leq t < n)$ is the threshold. It allows any set of $t + 1$ parties to reconstruct the secret, while a set of t or less parties can not obtain any information about the secret, thus all qualified sets have size at least $t + 1$:

$$\Gamma = \{A \subseteq \{P_1, ..., P_n\} || A| > t + 1\}, 1 \leqslant t < n$$

Shamir's Secret Sharing Scheme is based on Lagrange polynomial interpolation over finite fields. It consists of three phases:

- **Setup:** All parties agree on the following public parameters: a finite field $\mathbb{F}_q$ with size $q > n$, the threshold is t, with $t < n$. Each party P_i is assigned a public $z_i \in \mathbb{F}_q$, with $z_i \neq 0$ and $z_i \neq z_j$ for $i \neq j$.
- **Distribution:** The dealer picks t elements from $\mathbb{F}_q$: $a_1, a_2, ..., a_t$ randomly and independently. He defines a polynomial $f(z)$:

  $$f(z) = s + a_1 z + a_2 z^2 + ... + a_t z^t$$

  Note that $f(0) = s$. After that the dealer sends to each party P_i a share $[s]_i = f(z_i)$ on a private channel.
- **Reconstruction:** Any $t+1$ parties can reconstruct s with their shares by using Lagrange interpolation:

  $$s = f(0) = \sum_{i \in I} f(z_i) L_i(0)$$

  where

  $$L_i(z) = \prod_{j \in I \setminus \{i\}} \frac{z - z_j}{z_i - z_j}, I \subseteq \{1, 2, ..., n\}, |I| = t + 1$$

The setup phase consists of agreement on a finite field $\mathbb{F}_q$ with size q and corruption threshold $t < n$, and a list of public, distinct, non-zero value $z_i \in \mathbb{F}_q$ for $1 \leqslant i \leqslant n$. All computations are done in the field $\mathbb{F}_q$. A typical setting is $\mathbb{F}_q = \mathbb{Z}_p$, with p prime, and $z_i = i$.

In Shamir's (t, n)-threshold secret sharing scheme, the secret s can be reconstructed for any set of $t + 1$ shares, hence the correctness property holds for a passive adversary. While for t or less shares, the computed secret can not be distinguished from a random field element, so the privacy property holds perfectly. The share size in the scheme is optimal for perfect security: each party receives a share whose size is equal to that of the secret, i.e., a field element.

In this thesis, we use Protocol 2.1 and Protocol 2.2 to denote the Shamir's (t, n)-threshold secret sharing scheme, assuming party P_i is the dealer.

Protocol 2.1 Generate Shamir's Share $[s] \leftarrow \mathit{ShamirShare}(\mathbb{Z}_p, t, n, s)$

```
1: party $P_i$ randomly picks $a_1, a_2, ..., a_t$ from $\mathbb{Z}_p$
2: foreach $j \in \{1, 2, ..., n\}$ do
3:     $[s]_j \leftarrow s + \sum_{\ell=1}^{t} a_\ell j^\ell$
4:     send $[s]_j$ to party $P_j$
5: end for
6: return $[s]$
```

After the execution of Protocol *ShamirShare*, each party P_i will get its share denoted by $[s]_i$. To recover the secret, a qualified set with at leat $t+1$ parties is required. Let $A \in \Gamma$ be a qualified set with size $t+1$, the reconstruction step of Shamir's (t, n)-threshold secret sharing scheme is shown as follows:

Protocol 2.2 Open Shamir's Share $s \leftarrow \mathit{OpenShare}(A, [s])$

```
1: for each party $P_i \in A$, send $[s]_i$ to all parties
2: all parties compute $s \leftarrow \sum_{P_i \in A} [s]_i \prod_{P_j \in A, j \neq i} \frac{-j}{i-j}$
3: return $s$
```

Notation. For convenience, we assume there is a default setting for Sharmir's secret sharing scheme, i.e., $\mathbb{Z}_p, t, n, A$ is known to all parties, and skip these arguments for protocol *ShamirShare* and *OpenShare*, hence the above two protocols are called like this:
$[s] \leftarrow \mathit{ShamirShare}(s)$ and $s \leftarrow \mathit{OpenShare}([s])$

2.2.3 Replicated Secret Sharing

Replicated Secret Sharing is another secret sharing scheme, it is not efficient, compared to Shamir's Secret Sharing. However, it is quite useful for noninteractively generating random secrets. Assuming there are n parties $P = \{P_1, ..., P_n\}$, and a threshold monotone access structure Γ with threshold $t < n$. Let $\mathcal{T} = \{T_1, T_2, ..., T_w\}$ be the set of all maximal forbidden set, i.e., $\mathcal{T} = \{T \subset P || T| = t\}, w = \binom{n}{t}$, and $\mathcal{T} \subset \mathcal{P}(P) - \Gamma$. The scheme for Γ over a finite field $\mathbb{F}_q$ is defined as follows [CDI05]:

- **Share Distribution.** To share a secret $s \in \mathbb{F}$, the dealer generates random element $[s]_i^R$ for $1 \leqslant i \leqslant w-1$ from $\mathbb{F}_q$, and set $[s]_w^R$ to be $s - \sum_{i=1}^{w-1} [s]_i^R$. To distribute the shares, the dealer send $[s]_i^R$ to $P \backslash T_i$ for $1 \leqslant i \leqslant w-1$.
- **Share Reconstruction.** Let $A \subseteq P$ be a set of minimal qualified set, i.e., $|A| = t+1$. Parties in A poll their shares to get the entire vector $([s]_1^R, ..., [s]_w^R)$, and compute $s = \sum_{i=1}^{w} [s]_i^R$.

We use $[s]^R$ to denote replicated share of s, and the shares hold by parties are denoted by $([s]_1^R, ..., [s]_w^R)$. The above scheme is shown as Protocol 2.3 and Protocol 2.4, assuming party P_i is the dealer, A is a qualified set with size $t+1$.

Protocol 2.3 Generate Replicated Share $[s]^R \leftarrow \mathit{RShare}(\mathbb{Z}_p, t, n, s)$

```
1: party $P_i$ picks $[s]_i^R$ for $1 \leqslant i \leqslant w-1$ from $\mathbb{Z}_p$
2: $[s]_w^R \leftarrow s - \sum_{k=1}^{w-1} [s]_k^R$
3: foreach $j \in \{1, 2, ..., w\}$ do
4:     party $P_i$ send $[s]_j^R$ to $P \backslash T_j$
5: end for
6: return $[s]^R$
```

After the distribution phase, each party P_j obtains a share vector $V_j = ([s]_i^R)_{P_j \notin T_i}$, with $\binom{n-1}{t}$ elements. For parties in $T_i \in \mathcal{T}$, they cannot recover s, as exactly one share $[s]_i^R$ is missing. And for smaller forbidden set, they miss miss at least one share to reconstruct the secret. While for a qualified set, the secret can be recovered as $s = \sum_{i=1}^{w} [s]_i^R$, since they can combine their shares, and jointly obtain the entire vector $([s]_1^R, ..., [s]_w^R)$. Thus the correctness and privacy property hold for the scheme.

Protocol 2.4 Open Replicated Share $s \leftarrow \mathit{ROpen}(A, [s]^R)$

```
1: foreach party $P_i \in A$ do
2:     for each $j$ s.t. $P_i \notin T_j$, send $[s]_j^R$ to all parties
3: end for
4: all parties compute $s \leftarrow \sum_{\ell=1}^{w} [s]_\ell^R$
5: return $s$
```

Replicated sharing is very inefficient, but its shares can be locally converted to shares of any LSS scheme [CDI05]. Let $[s]^R = ([s]_1^R, ..., [s]_w^R)$ be the RSS share of a secret $s \in \mathbb{Z}_p$, Protocol 2.5 shows how to convert a RSS share $[s]^R$ to Shamir's share $[s]$ for the same access structure.

Protocol 2.5 Convert Replicated share to Shamir's Share $[s] \leftarrow \mathit{RSS2Shamir}([s]^R)$

```
1: foreach $i \in \{1, 2, ..., n\}$ in parallel do
2:     party $P_i$ compute $[s]_i \leftarrow \sum_{j=1, P_i \notin T_j}^{w} ([s]_j^R \prod_{P_\ell \in T_j} \frac{\ell - i}{\ell})$
3: end for
4: return $[s]$
```

For the correctness of Protocol *RSS2Shamir*, the reader can refer to [CDI05, Hoo12]. Privacy follows from the locality of conversion, and the converted shares cannot leak more information than the original shares, as all computation are done locally.

2.3 Performance Analysis

To analyze the performance of our protocols in a practical setting, we build applications using Virtual Ideal Functionality Framework (VIFF). VIFF is a general software framework for doing secure multiparty computation [Gei10], which provides researchers and programmers with the basic building blocks (or sub-protocols) as APIs to allow rapid prototyping of new protocols and building practical applications.

This section provides a brief introduction to VIFF, and presents our experiment settings for performance analysis.

2.3.1 Virtual Ideal Functionality Framework

The interface offered by VIFF is viewed as an ideal functionality $\mathcal{F}_{VIFF}$ within the UC framework. An implementation of this ideal functionality is called a runtime class in VIFF. There are several runtime classes in VIFF, dealing with different security settings:

- **Paillier Runtime.** This is a special two-player runtime based on the homomorphic Paillier cryptosystem [Pai99], which is secure against a passive (semi-honest) adversary.
- **Passive Runtime.** This runtime implements the BGW protocol [BGW88] for n players ($n > 2$). It is secure against passive adversaries as long no more than $n/2$ of the parties are corrupted.

- **Active Runtime.** This runtime offers security against active adversaries for n players ($n > 2$), based on [DGKN09]. It can tolerate at most $n/3$ maliciously corrupted parties.

This thesis focus on passive adversaries, thus the Passive Runtime is used as the base (or start point) for building privacy preserving data mining applications. Besides the basic functionality implemented in these runtime classes, VIFF also provide extra useful functions that can be incorporate into a runtime, for example, the Pseudorandom Secret Sharing (PRSS) module for random secret generation, the mixin classes for secure integer comparison. Chapter 3 reviews the basic protocols in Passive Runtime, together with other useful protocols.

VIFF was created at the University of Aarhus in Denmark in 2008, and it has been used for several large applications which demonstrated its capability for secure multiparty computation [BCD⁺09, Mau09, DK10, Hoo12]. This is the main reason that we choose VIFF to design and build applications for privacy preserving data mining.

The notable application built using VIFF is the Nordic Sugar auction system for Danish sugar beet farmers. With this system, trading sugar beets was possible without finding and paying a trusted third party to manage the auction. The application was first implemented in Java language by the SIMAP research project, and was successfully carried out in Denmark in 2008 [BCD⁺09]. Later, it was rewritten in Python language using VIFF, and was successfully repeated in 2009 [Gei10].

2.3.2 Experiment Settings

In VIFF, there are several parameters that are used to config how a runtime class should work. For performance analysis, we are mainly interested in the following runtime options:

- **Bit Length.** This parameter defines maximum bit length of input numbers for our protocols. Usually a single field $\mathbb{Z}_p$ is used for all protocols in an application, thus to avoid integer overflow in $\mathbb{Z}_p$, the prime p is chosen to be large than $(2^{\ell+1} + 2^{\ell+k+1})$, where l is the bit length, k is a security parameter. The security security parameter is used to ensure the integer comparison protocols (see Section 3.5) work correctly, and guarantee statistical security (see Chapter 3).

- **Players Setting.** Players are the parties that run the secure multiparty computation protocols. In our experiment, we have three players, and the threshold for secret sharing scheme is set to 2, i.e., it is a linear $(2, 3)$-threshold secret sharing scheme. These three players run the protocols on different network ports of the same computer, which is a HP Elitebook 8540w laptop, with Intel Core i5 M540 CPU @2.53GHz (2 cores, 4 threads), 8GB memory, and a Ubuntu 10.04 64bit OS.

For efficient implementation, we used a boost extension to VIFF, which greatly improves the performance of VIFF applications [Kel10].

Chapter 3

Basic Protocols

This chapter reviews some basic SMC protocols, which are used as building blocks for privacy preserving data mining solutions in Chapter 4 and Chapter 5. We consider these protocols under the typical setting of Shamir's secret sharing scheme as described in Section 2.2, though some of them can be also applied in the Threshold Homomorphic Cryptosystems. Most of the protocols in this chapter are already implemented in VIFF.

We start with the basic SMC protocols for arithmetic operations in Section 3.1. The second and third section shows how to generate random secret in an interactive way and non-interactive way respectively. Some basic constructions for round efficiency are presented in Section 3.4. Section 3.5 reviews several protocols for integer comparison. In Section 3.6, we propose a new protocol for symmetric AND/OR function. A summary of protocols introduced in this chapter is given in Section 3.7.

3.1 Secure Arithmetic

This section reviews SMC protocols for basic arithmetic operations, including linear combination, multiplication, inner product, and inverse. These protocols provide perfect privacy against a passive adversary, for the correctness proof and security analysis, we refer readers to [BGW88, Tof07, Hoo12]. The protocols in this section are already implemented in VIFF.

3.1.1 Linear Combination

Shamir's (t, n)-threshold secret sharing scheme is a LSS scheme, it allows each party to locally compute linear combinations of secrets and public values. Linear combination of secrets includes the following operations:

- **Addition of secrets ($[c] \leftarrow [a] + [b]$):** Each party P_i locally computes its share of the result $[c]_i = [a]_i + [b]_i$.
- **Addition of secret and public value ($[c] \leftarrow [a] + \alpha, \alpha \in \mathbb{F}_q$):** Each party P_i locally computes its share of the result $[c]_i = [a]_i + \alpha$.
- **Multiplication of secret and public value ($[c] \leftarrow [a] \cdot \alpha, \alpha \in \mathbb{F}_q$):** Each party P_i locally computes its share of the result $[c]_i = [a]_i \cdot \alpha$.

Since the above operations can all be done locally, we simply use the $+$ to denote addition operation, and the $\cdot$ to denote the multiplication of secret and public value in this thesis.

3.1.2 Multiplication

Multiplication of secrets requires an interactive protocol, which is shown as Protocol 3.1. The idea is from the multiplication protocol proposed in [BGW88]. Given two secrets $[a], [b]$, each party P_i computes a share $[a]_i[b]_i$. As a secret $[a]$ is represented by a uniformly random t-degree polynomial f in Shamir's secret sharing scheme, the result polynomial of multiplication will have a degree of $2t$, and not uniformly random, thus $[a]_i[b]_i$ is not the correct share of $[ab]$.

To ensure the result polynomial has a degree of t, and be uniform random, the parties need to interactively compute a new random t-degree polynomial h, where $h(0) = ab$. The restriction $2t+1 \leqslant n$ must hold, otherwise it would be impossible to reconstruct the result from the $2t$-degree polynomial.

Protocol 3.1 Multiplication of secrets $[c] \leftarrow Mul([a], [b])$

```
Input: $[a], [b]$, where $a, b \in \mathbb{Z}_p$
Output: $[c]$, where $c = a \cdot b$, $c \in \mathbb{Z}_p$
1: foreach $i \in \{1, 2, ..., 2t+1\}$ in parallel do
2:     $P_i$ computes $d_i \leftarrow [a]_i \cdot [b]_i$
3:     $P_i$ shares $d_i$ into $[d_i] \leftarrow ShamirShare(d_i)$
4: end for
5: foreach $j \in \{1, 2, ..., n\}$ in parallel do
6:     $P_j$ computes $[c]_j \leftarrow \sum_{i=1}^{2t+1} ([d_i]_j \prod_{\ell=1,\ell\neq i}^{2t+1} \frac{-\ell}{i-\ell})$
7: end for
```

Complexity Analysis. Protocol 3.1 for the multiplication of secrets requires one round and one invocation. Note that only the first $2t+1$ parties are involved in the computation of d_i, this can save some communication and computation cost. To further optimize communication and computation load, it is possible to select any arbitrary subset of $2t+1$ parties.

3.1.3 Inner Product

Let $[a] = ([a_1], [a_2], ..., [a_m])$ be a shared m-vector, which contains a list of m shared secrets. To compute the inner product of two shared m-vector $[a], [b]$, a naive way would to use protocol *Mul* to compute a new m-vectors $([a_1 \cdot b_1], [a_2 \cdot b_2], ..., [a_m \cdot b_m])$, and then add all elements in the vector locally to get the final result. Protocol 3.2 shows how to extend the multiplication protocol in [BGW88] to be able to compute any generalized inner product by adding local computations only [Hoo12].

Protocol 3.2 Inner Product of secret $[c] \leftarrow Inner([\boldsymbol{a}], [\boldsymbol{b}])$

```
Input: $[\boldsymbol{a}] = ([a_1], [a_2], ..., [a_m]), [\boldsymbol{b}] = ([b_1], [b_2], ..., [b_m])$
Output: $[c]$, where $c = \sum_{j=1}^{m} a_j \cdot b_j$
1: foreach $i \in \{1, 2, ..., 2t+1\}$ in parallel do
2:     $P_i$ computes $d_i \leftarrow \sum_{j=1}^{m} [a_j]_i \cdot [b_j]_i$
3:     $P_i$ shares $d_i$ into $[d_i] \leftarrow ShamirShare(d_i)$
4: end for
5: $[c] \leftarrow \sum_{i=1}^{2t+1} ([d_i] \cdot \prod_{\ell=1,\ell\neq i}^{2t+1} \frac{-\ell}{i-\ell})$
```

Complexity Analysis. While the naive solution using protocol *Mul* requires m rounds and m invocations, protocol *Inner* only needs one round and one invocation, which is much more efficient than the naive solution.

3.2 Interactive Generation of Random Secret

Many SMC protocols require the ability to generate shared random secrets. There are two different techniques for the generation of random secrets. This section presents the interactive way to generate random secret [DFK+06, Tof07]. The non-interactive solutions based on pseudorandom secret sharing (PRSS) is introduce in Section 3.3 [CDI05].

3.2.1 Shared Random Element

Protocol 3.3 is used to jointly compute a shared secret $[r]$ for some unknown $r \in \mathbb{Z}_p$. The idea is to compute $r = \sum\limits_{i=1}^{n} x_i$, where $x_i \in \mathbb{Z}_p$ is chosen uniformly random by P_i. If at least one party P_i is honest, then at least one x_i is uniformly random, thereby r is uniformly random.

Protocol 3.3 Shared random element $[r] \leftarrow \mathit{Rand}(\mathbb{Z}_p)$

```
Input: $\mathbb{Z}_p$
Output: $[r]$, where $r \in \mathbb{Z}_p$
1: foreach $i \in \{1, 2, ..., n\}$ in parallel do
2:     $P_i$ picks a random $x_i \in \mathbb{Z}_p$
3:     $P_i$ shares $x_i$ into $[x_i] \leftarrow \mathit{ShamirShare}(x_i)$
4: end for
5: $[r] \leftarrow \sum_{i=1}^{n} [x_i]$
```

Complexity Analysis. Protocol *Rand* requires one round and one invocation.

3.2.2 Shared Random Element in Range

To generate a shared random element $[r]$ with bounded bit size, where $r \in \{0, ..., 2^m - 1\}$, Protocol 3.4 can be used. Let $\tau : \mathbb{Z}_n \times [1..n] \mapsto \{0, 1\}$ be a public function, s.t. $\forall x \in \mathbb{Z}_n : \sum_{i=1}^{n} \tau(x, i) = x$.

Protocol 3.4 Shared random element in range $[r] \leftarrow \mathit{Rand2m}(\mathbb{Z}_p, m)$

```
Input: $\mathbb{Z}_p$, $m$
Output: $[r]$, where $r \in [0, ..., 2^m - 1]$, $r \in \mathbb{Z}_p$
1: foreach $i \in \{1, 2, ..., n\}$ in parallel do
2:     $a_i \leftarrow \lfloor (2^m - 1)/n \rfloor + \tau(2^m - 1 \mod n, i)$
3:     $P_i$ picks a random $x_i$ from $\{0, ..., a_i\}$
4:     $P_i$ shares $x_i$ into $[x_i] \leftarrow \mathit{ShamirShare}(x_i)$
5: end for
6: $[r] \leftarrow \sum_{i=1}^{n} [x_i]$
```

The basic idea of protocol *Rand2m* is the same as Protocol *Rand*, namely to get the sum of n random uniform integers. While in Protocol *Rand2m*, the random integer x_i is chosen from a bounded range, which is achieved by the public function τ.

Complexity Analysis. Protocol *Rand2m* has the same complexity with Protocol *Rand*: one round and one invocation.

3.2.3 Shared Random Bit

Protocol 3.5 generates a shared random bit $[b]$, where $b \in \{0,1\} \subset \mathbb{Z}_p$. This protocol requires a prime modulus p with the property that $p \equiv 3\ (mod\ 4)$. Protocol *Rand* is used to generate a uniformly random value $[r]$ which is then squared and opened. If the product is zero, the protocol fails and retries, otherwise the parties compute the $v = u^{(p+1)/4} = r^{(p+1)/2} = r^{(p-1)/2} \cdot r$. Note that $r^{(p-1)/2}$ is the Legendre symbol $(\frac{r}{p}) = \pm 1$ for $r \neq 0$, then $b = 2^{-1} \cdot (v^{-1} \cdot r + 1) = 2^{-1} \cdot (\pm 1 \cdot r^{-1} \cdot r + 1) = 2^{-1} \cdot (\pm 1 + 1)$. Thus $b = 1$ if $(\frac{r}{p}) = 1$, and $b = 0$ if $(\frac{r}{p}) = -1$.

Protocol 3.5 Shared random bit $[b] \leftarrow$ *RandBit*$(\mathbb{Z}_p)$

```
Input: $\mathbb{Z}_p$, where $p \equiv 3 \mod 4$
Output: $[b]$, where $b \in \{0,1\} \subset \mathbb{Z}_p$
1: $[r] \leftarrow Rand(\mathbb{Z}_p)$
2: $[u] \leftarrow Mul([r],[r])$
3: $u \leftarrow OpenShare([u])$
4: if $u = 0$ then
5:     abort and retry
6: else
7:     $v \leftarrow u^{(\frac{p+1}{4})}$
8:     $[b] \leftarrow 2^{-1} \cdot (v^{-1} \cdot [r] + 1)$
9: end if
```

The value of Legendre symbol $(\frac{r}{p})$ depends on whether r is quadratic residue or not, and the probability that $(\frac{r}{p}) = 1$ is $1/2$ for $r \in \mathbb{Z}_p^*$, thus the bit b is uniformly random. The protocol fails with probability $1/p$, which is negligible for a large p that $p > 2^k$, where k is a security parameter.

Complexity Analysis. Protocol *RandBit* requires three rounds and three invocations.

3.2.4 Shared Random Invertible Element

Protocol 3.6 generates a non-zero shared random field element $[r]$, and optionally its inverse $[r^{-1}]$. The parties use the protocol *Rand* to generate two shared random secrets $[x]$ and $[y]$, and then reveal their product $[x \cdot y]$. If the product is 0, the protocol fails and retries, while if not, a non-zero shared random field element $[r]$ and its inverse $[r^{-1}]$ can be obtained in a trivial way.

We use *Rand** to denote the protocol that only generates a non-zero shared random field element $[r]$ without its inverse $[r^{-1}]$. Note that the protocol fails with probability $2/p$, when x or y is zero.

Protocol 3.6 Shared random invertible element $[r], [r^{-1}] \leftarrow \mathit{RandInv}(\mathbb{Z}_p)$

```
Input: $\mathbb{Z}_p$
Output: $[r], [r^{-1}]$, where $r \in \mathbb{Z}_p$
 1: $[x] \leftarrow \mathit{Rand}(\mathbb{Z}_p)$
 2: $[y] \leftarrow \mathit{Rand}(\mathbb{Z}_p)$
 3: $[c] \leftarrow \mathit{Mul}([x], [y])$
 4: $c \leftarrow \mathit{OpenShare}([c])$
 5: if $c = 0$ then
 6:     abort and retry
 7: else
 8:     $[r] \leftarrow [x]$
 9:     $[r^{-1}] \leftarrow c^{-1} \cdot [y]$
10: end if
```

Complexity Analysis. The random secrets $[x]$ and $[y]$ can be generated in parallel, thus the overall complexity is three rounds and four invocations.

3.3 Non-Interactive Generation of Random Secret

Random secret generation can be also achieved in a non-interactive way [CDI05], where the parties can generate fresh random shares by local computation only, using a set of previously distributed pseudo-random functions (PRFs). This technique is known as Pseudorandom Secret Sharing (PRSS), which is implemented in VIFF as PRSS module. This section presents a number of protocols for non-interactive generation of random secret from [CDI05], we refer readers to [CDI05, Hoo12] for the correctness proof and security analysis of these protocols.

PRSS requires a set of PRFs, and these PRFs depends on a set of PRF keys, which are previously distributed as replicated shares. This can be done by a trusted party, which is the case in VIFF, or using an interactive protocol to jointly share them, as shown in Protocol 3.7.

Notation. Assume there are n parties $P = \{P_1, ..., P_n\}$, and a threshold monotone access structure Γ with threshold $t < n$. Let $\mathcal{T} = \{T_1, T_2, ..., T_w\}$ be the set of all maximal forbidden set, i.e., $\mathcal{T} = \{T \subset P || T| = t\}, w = \binom{n}{t}$.

Protocol 3.7 Setup for Pseudorandom Secret Sharing $[k]^R \leftarrow \mathit{SetupPRSS}(\mathbb{Z}_p)$

```
1: foreach $i \in \{1, 2, ..., n\}$ do
2:     $P_i$ randomly picks $k_i$ from $\mathbb{Z}_p$
3:     $[k_i]^R \leftarrow \mathit{RShare}(\mathbb{Z}_p, t, n, k_i)$
4:     $P_i$ locally computes $[k]^R_j \leftarrow \sum_{i=1}^{n} [k_i]^R_j$, where $j$ such that $P_i \notin T_j$
5: end for
6: return $[k]^R$
```

The parties randomly picks $k_i \in \mathbb{Z}_p$ for $1 \leqslant i \leqslant n$, and distribute them using replicated secret sharing scheme. Then each party P_i locally computes a consistent replicated sharing of $k = \sum_{i=1}^{n} k_i$. The replicated shares of k are used as the PRF keys.

3.3.1 Shared Random Element

Let $\mathcal{H} : \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Z}$ be the pseudo-random function (PRF), $ctr \in \mathbb{N}$ is a static counter. Protocol 3.8 shows how to locally generate the RSS shares of a random secret [CDI05].

Protocol 3.8 Generate RSS Shares of a random element $[r]^R \leftarrow PRandRSS(\mathbb{Z}_p)$

```
1: static $ctr \leftarrow 0$
2: static $[k]^R \leftarrow SetupPRSS(\mathbb{Z}_p)$
3: foreach party $P_i$, $i \in \{1, 2, ..., n\}$ do
4:     $[r]^R_j \leftarrow \mathcal{H}([k]^R_j, ctr)$, where $j$ such that $P_i \notin T_j$
5:     $ctr \leftarrow ctr + 1$
6: end for
7: return $[r]^R$
```

In Protocol *PRandRSS*, each party P_i computes $[r]^R_j \leftarrow \mathcal{H}([k]^R_j, ctr)$, for all j such that $P_i \notin T_j$. It follows that $[r]^R$ is a consistent replicated sharing, where $r = \sum_{j=1}^{w} \mathcal{H}([k]^R_j, ctr)$, thus r is uniformly random.

As shown in Section 2.2.3, the replicated share can be locally converted to Shamir's share. Hence the non-interactive generation of random field element can be achieved as follows.

Protocol 3.9 Generation of shared random element $[r] \leftarrow PRand(\mathbb{Z}_p)$

```
1: $[r]^R \leftarrow PRandRSS(\mathbb{Z}_p)$
2: $[r] \leftarrow RSS2Shamir([r]^R)$
3: return $[r]$
```

3.3.2 Shared Random Element in Range

Similarly to Protocol *PRandRSS*, Protocol 3.10 shows how to noninteractively generate replicated shares of a random element with bounded bit size [CDI05]. Instead of using $\mathcal{H}$, we define a different PRF $\mathcal{H}^\alpha : \mathbb{Z} \times \mathbb{N} \rightarrow \{0, 1\}^\alpha$, which outputs a uniformly randomly chosen integer of fixed α bit size. And suppose that the parties have agreed upon a static counter $ctr \in \mathbb{N}$.

Each party P_i computes $[r]^R_j \leftarrow \mathcal{H}^\alpha([k]^R_j, ctr)$, for all j such that $P_i \notin T_j$. It follows that $[r]^R$ is a consistent replicated sharing, where $r = \sum_{j=1}^{w} \mathcal{H}^\alpha([k]^R_j, ctr)$, thus r has bit size $\alpha + \log w$ and its distribution is equal to the sum of w uniformly random numbers.

Protocol 3.10 Generate RSS Shares of a random element in range $[r]^R \leftarrow PRand2mRSS(\mathbb{Z}_p, \alpha)$

```
1: static $ctr \leftarrow 0$
2: static $[k]^R \leftarrow SetupPRSS(\mathbb{Z}_p)$
3: foreach party $P_i$, $i \in \{1, 2, ..., n\}$ do
4:     $[r]^R_j \leftarrow \mathcal{H}^\alpha([k]^R_j, ctr)$, where $j$ such that $P_i \notin T_j$
5:     $ctr \leftarrow ctr + 1$
6: end for
7: return $[r]^R$
```

By using *RSS2Shamir*, we can get the Shamir shares of $[r]^R$, as shown in Protocol 3.11.

Protocol 3.11 Generation of shared random element in range $[r] \leftarrow PRand2m(\mathbb{Z}_p, \alpha)$

```
1: $[r]^R \leftarrow PRandRSS(\mathbb{Z}_p, \alpha)$
2: $[r] \leftarrow RSS2Shamir([r]^R)$
3: return $[r]$
```

3.3.3 Pseudo Random Zero Sharing

Protocol 3.12 generates Shamir's random shares of zero [0] using a random polynomial of degree $2t < n$. The polynomial is defined as

$$z(x) = \sum_{i=1, P_j \notin T_i}^{w} p_i(j)([r_1]_j^R j + [r_2]_j^R j^2 + ... + [r_t]_j^R j^t)$$

where $p_i(x) = \prod_{P_j \in T_i} \frac{j-x}{j}$, $[r_i]^R$ is a random replicated share (for $1 \leqslant i \leqslant t$), and $z(0) = 0$.

Protocol 3.12 Generation of shared random element in range $[z] \leftarrow PRandZero(\mathbb{Z}_p)$

```
1: foreach $i \in \{1, 2, ..., t\}$ do
2:     $[r_i]^R \leftarrow PRandRSS(\mathbb{Z}_p)$
3: end for
4: foreach $j \in \{1, 2, ..., n\}$ do
5:     $[z]_j \leftarrow \sum_{i=1, P_j \notin T_i}^{w} p_i(j)([r_1]_j^R j + [r_2]_j^R j^2 + ... + [r_t]_j^R j^t)$
6: end for
7: return $[z]$
```

The correctness proof and security analysis of Protocol *PRandZero* can be found in [CDI05, Hoo12].

3.4 Basic Constructions for Round Efficiency

In secure multiparty computation protocols, round complexity is a dominant factor in the execution time. To achieve better performance in practise, SMC protocols should be designed with less rounds and invocations. This section presents some basic constructions that can be used to reduce the round complexity. The Pseudorandom Secret Sharing (PRSS) introduce in previous section is efficient, as it does not require any interactive communication (except for the setup protocol), thus we will use this technique wherever possible in this thesis.

3.4.1 Multiplication with Public Output

Suppose the parties wish to reveal the product of two secrets, the obvious way is to securely compute their product first by $[c] \leftarrow Mul([a], [b])$, and then open the product by $c \leftarrow OpenShare([c])$. In this way, the complexity is 2 rounds and 2 invocations. Protocol 3.13 exploits the benefits of Protocol *PRandZero*, and achieve this goal with only one round and one invocation. $[z]$ is generated to ensure that any set of 2t values of mi is uniformly random and does not depend on a and b. The protocol is proven to be secure in [CDI05].

Protocol 3.13 Multiplication with public output $c \leftarrow MulPub([a], [b])$

```
1: $[z] \leftarrow PRandZero(\mathbb{Z}_p)$
2: foreach $i \in \{1, 2, ..., 2t+1\}$ in parallel do
3:     $P_i$ computes $m_i \leftarrow [a]_i \cdot [b]_i + [z]_i$
4:     $P_i$ sends $m_i$ to all parties
5: end for
6: all parties compute $c \leftarrow \sum_{i=1}^{2t+1} m_i \prod_{j=1,j\neq i}^{2t+1} \frac{-j}{i-j}$
7: return $c$
```

Complexity Analysis. Protocol *MulPub* has the same complexity as Protocol *Mul*, namely one round and one invocation.

3.4.2 Generation of Shared Random Bit using PRSS

Protocol 3.14 shows how to generate a shared random bit using the PRSS technique. The basic idea is the same as Protocol *RandBit*, while Protocol *PRandBit* exploits the benefits of protocol *PRand* and protocol *MulPub*.

Protocol 3.14 Generate Shared Random Bit using PRSS $[b] \leftarrow PRandBit(\mathbb{Z}_p)$

```
Input: $\mathbb{Z}_p$, where $p \equiv 3 \mod 4$
Output: $[b]$, where $b \in \{0, 1\} \subset \mathbb{Z}_p$
1: $[r] \leftarrow PRand(\mathbb{Z}_p)$
2: $u \leftarrow MulPub([r], [r])$
3: if $u = 0$ then
4:     abort and retry
5: else
6:     $v \leftarrow u^{-(\frac{p+1}{4})}$
7:     $[b] \leftarrow 2^{-1} \cdot (v \cdot [r] + 1)$
8: end if
```

Complexity Analysis. Protocol *PRandBit* requires one round and one invocation, which is more efficient compared to three rounds and two invocations required by Protocol *RandBit*.

3.4.3 Generation of Shared Random Invertible Element using PRSS

Protocol 3.15 shows how to generate a non-zero shared random field element $[r]$, using PRSS technique. The idea is basically the same with Protocol *RandInv*, namely to generate two shared random secrets $[x]$ and $[y]$, and then reveal their product. Protocol *MulPub* is used here to save one round and one invocation.

Note that the protocol fails with probability $2/p$, when x or y is zero. In practice, a large p is chosen such that probability $2/p$ negligible. We use *PRand** to denote the protocol that only generates a non-zero shared random field element $[r]$ without its inverse $[r^{-1}]$.

Protocol 3.15 Generate Shared Random Invertible Element $[r], [r^{-1}] \leftarrow \mathit{PRandInv}(\mathbb{Z}_p)$

```
Input: $\mathbb{Z}_p$
Output: $[r], [r^{-1}]$, where $r \in \mathbb{Z}_p$
1: $[x] \leftarrow \mathit{PRand}(\mathbb{Z}_p)$
2: $[y] \leftarrow \mathit{PRand}(\mathbb{Z}_p)$
3: $c \leftarrow \mathit{MulPub}([x], [y])$
4: if $c = 0$ then
5:     abort and retry
6: else
7:     $[r] \leftarrow [x]$
8:     $[r^{-1}] \leftarrow c^{-1} \cdot [y]$
9: end if
```

Complexity Analysis. Protocol *PRandInv* requires one round and one invocation.

3.4.4 Inverse of Field Element

Let $[a]$ be a shared integer in $\mathbb{Z}_p$, Protocol 3.16 shows how to securely compute the inverse of a, which is denoted by $[a^{-1}]$ $(a \neq 0)$. The idea is to compute and reveal $b = a \cdot r$, where r is a random secret. If $b \neq 0$, the parties can locally compute $[a] \cdot b^{-1}$, which is the inverse of $[a]$.

Protocol 3.16 Invert of shared secret $[a^{-1}] \leftarrow \mathit{Inv}([a])$

```
Input: $[a]$, where $a \neq 0$, $a \in \mathbb{Z}_p$
Output: $[a^{-1}]$, where $a^{-1} \cdot a = 1$, $a^{-1} \in \mathbb{Z}_p$
1: $[r] \leftarrow \mathit{PRand}(\mathbb{Z}_p)$
2: $b \leftarrow \mathit{MulPub}([a], [r])$
3: if $b = 0$ then
4:     abort and retry
5: else
6:     $[a^{-1}] \leftarrow b^{-1} \cdot [r]$
7: end if
```

Complexity Analysis. Protocol *Inv* requires one round and one invocation.

3.4.5 Generation of Bitwise Shared Random Secrets

Let $[a]$ be a shared integer in $\mathbb{Z}_p$, we use $[a]_B = ([a_{\ell-1}], ..., [a_0])$ to denote the secret shared bits of a, where $\ell = \lceil \log_2(p) \rceil, a_0, ..., a_{\ell-1} \in \{0, 1\} \subset \mathbb{Z}_p$ and $a = \sum_{i=0}^{\ell-1} a_i 2^i$. Protocol 3.17 shows how to compute a shared bounded element $[r]$ and its bits $[r]_B$, where $r \in [0, 2^m - 1]$.

Protocol 3.17 Bitwise shared random secret $[r], [r]_B \leftarrow \mathit{PRandBitwise}(\mathbb{Z}_p, m)$

```
Input: $m$, where $p > 2^m$
Output: $[r], [r]_B = ([r_{m-1}], ..., [r_0])$, where $r \in \mathbb{Z}_p$, $r_i \in \{0, 1\} \subset \mathbb{Z}_p$ for $0 \leqslant i < m$
1: foreach $i \in \{0, ..., m-1\}$ in parallel do
2:     $[r_i] \leftarrow \mathit{PRandBit}(\mathbb{Z}_p)$        // generate $m$ random shared bits $[r]_B$
3: end for
4: $[r] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$            // local combination to obtain $[r]$
5: $[r]_B \leftarrow ([r_{m-1}], ..., [r_0])$
6: return $[r], [r]_B$
```

The basic idea is that the parties first generate shared random bits using *PRandBit* and then locally compute $[r] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$. To void integer overflow in field $\mathbb{Z}_p$, the filed size p must be larger than 2^m.

Complexity Analysis. Since the complexity for protocol *PRandBit* is one round and one invocation, and in step 2 the m random secret shared bits $[r]_B = ([r_{m-1}], ..., [r_0])$ can be generated in parallel, thus the overall complexity for this protocol is one rounds and m invocations.

3.4.6 k-ary Operation

A k-ary operation computes $y = a_1 \odot a_2 \odot ... \odot a_k = \odot_{i=1}^{k} a_i$, where $\odot$ denotes an associative binary operator. We present a well known protocol to perform k-ary operation with logarithmic rounds, as shown in Protocol 3.18.

Protocol 3.18 $\log(k)$ rounds operation $[c] \leftarrow$ *LogkOp*$(\odot, \boldsymbol{a})$

```
Input: $\odot, \boldsymbol{a} = (a_1, a_2, ..., a_k)$
Output: $c = \odot_{i=1}^{k} a_i$
1: foreach $i \in \{1, 2, ..., \lfloor k/2 \rfloor\}$ do
2:     $b_i \leftarrow a_{2i} \odot a_{2i-1}$
3: end for
4: if $k$ is even then
5:     $c \leftarrow LogkOp(\odot, (b_1, b_2, ..., b_{\lfloor k/2 \rfloor}))$
6: else
7:     $c \leftarrow LogkOp(\odot, (b_1, b_2, ..., b_{\lfloor k/2 \rfloor}, a_k))$
8: end if
```

Complexity Analysis. Assume the operation $\odot$ requires α rounds and β invocations, then the complexity of this protocol is $\alpha\lceil\log(k)\rceil$ rounds and $\beta(k-1)$ invocations. For convenience, we use *MulLogk* to denote Protocol 3.18 when the operation $\odot$ is multiplication. In this case, the overall complexity would be $\lceil\log(k)\rceil$ rounds and $(k-1)$ invocations.

3.4.7 Unbounded Fan-in Multiplication

Bar-Ilan and Beaver proposed a constant rounds solution for computing the prefix products $[c_j] = \prod_{i=1}^{j} [a_i]$ for $1 \leqslant j \leqslant k$ in [BIB89], where $a_i \neq 0$ for $1 \leqslant i \leqslant k$. The idea is to generate non-zero random values $[r_1], ..., [r_k]$ and compute their inverses $[r_1^{-1}], [r_2^{-1}], ..., [r_k^{-1}]$, and then compute and open $m_1 = a_1 r_1$, $m_i = r_{i-1}^{-1} a_i r_i$ for $i = 2, ..., k$. Hence $[c_j] = \prod_{i=1}^{j} [a_i] = [r_j^{-1}] \cdot \prod_{i=1}^{j} m_i$.

Complexity Analysis. The multiplication in step 4 and step 6, can be performed in parallel with the multiplication in protocol *PRandInv*, thus the overall complexity is 2 rounds and $3k - 1$ invocations.

The price paid for obtaining a constant rounds solution in protocol *PreMul* is a substantial increase of the communication and computation complexity. The prefix products can be also achieved with $\log(k)$ rounds and $k/2 \log(k)$ invocations, using a generic protocol for prefix operation [Hoo12], which requires less communication cost when $\log k \leqslant 6$, i.e., $k \leqslant 64$.

Protocol 3.19 Unbounded fan-in multiplication $[\boldsymbol{c}] \leftarrow PreMul([\boldsymbol{a}])$

```
Input: $[\boldsymbol{a}] = ([a_1], [a_2], ..., [a_k])$, where $a_i \in \mathbb{Z}_p, a_i \neq 0$ for $1 \leqslant i \leqslant k$
Output: $[\boldsymbol{c}] = ([c_1], [c_2], ..., [c_k])$, where $[c_j] = \prod_{i=1}^{j} [a_i]$ for $1 \leqslant j \leqslant k$
 1: foreach $i \in \{1, 2, ..., k\}$ do
 2:     $[r_i], [r_i^{-1}] \leftarrow PRandInv(\mathbb{Z}_p)$
 3: end for
 4: $m_1 \leftarrow MulPub([a_1], [r_1])$
 5: foreach $i \in \{2, ..., k\}$ do
 6:     $[m_i] \leftarrow Mul([r_i], [a_i])$
 7:     $m_i \leftarrow MulPub([m_i], [r_{i-1}^{-1}])$
 8: end for
 9: foreach $j \in \{1, ..., k\}$ do
10:     $[c_j] \leftarrow [r_j^{-1}] \cdot \prod_{i=1}^{j} m_i$
11: end for
```

3.5 Integer Comparison Protocols

This section shows how to construct protocols for comparing two secret inputs $[x]$ and $[y]$, where $x, y \in \mathbb{Z}_p$, using previously discussed protocols as building blocks. We start with a public equality test protocol in Section 3.5.1, which output public result, and then two equality test protocols with secret output are given in Section 3.5.2 and Section 3.5.3, and the greater equal than test protocol is presented in Section 3.1.

3.5.1 Equality Test with Public Result

Protocol 3.20 computes $s = (x = y)$, given two secrets $[x]$ and $[y]$. The idea is from [FH94], namely to generate a shared random non-zero element $[r]$, compute and reveal $c = (x - y) \cdot r$. Note that $x = y$ if and only if $(x - y)r = 0$, thus $s = (x = y) = (c = 0)$. Since r is uniformly random in $\mathbb{Z}_p^*$, then $(x - y)r$ is uniformly random in $\mathbb{Z}_p$, which hides $x - y$ perfectly if it is nonzero.

Protocol 3.20 Public Equality Test $s \leftarrow EqualPub([x], [y])$

```
Input: $[x], [y]$, where $x, y \in \mathbb{Z}_p$
Output: $s$, where $s = (x = y)$, $s \in \{0, 1\} \subset \mathbb{Z}_p$
 1: $[r] \leftarrow PRand^*(\mathbb{Z}_p)$
 2: $c \leftarrow MulPub([x] - [y], [r])$
 3: $s \leftarrow (c = 0)$
```

Complexity Analysis. The protocol requires 2 rounds and 2 invocations.

While Protocol *EqualPub* can give the equality test result without revealing the value of x, y, it leaks one bit information about x, y, namely whether they are equal or not. In some cases, we also want to keep the equality test result secret, i.e., given two secret inputs $[x], [y]$, to compute a secret bit value $[s]$, where $s = (x = y)$ without revealing any information about the inputs. This goal can be achieved by Protocol 3.21 in Section 3.5.2 and Protocol 3.22 in Section 3.5.3.

3.5.2 Probabilistic Equality Test

Nishide and Ohta proposed a probabilistic equality test protocol with a very small round complexity in [NO07]. The basic idea is based on the property of quadratic residues as follows: if a is zero, then $(\frac{c}{p}) = (\frac{r}{p})$ holds, where $c = a + r$, r is a random secret. If a is not zero, then $(\frac{c}{p}) \neq (\frac{r}{p})$ holds with non-negligible probability.

The protocol requires $p = 3 \mathit{mod}\, 4$, which implies that Legendre symbol $(\frac{-1}{p}) = -1$. The parties compute $c_j = ar_j + b_j {r'_j}^2$ for $1 \leqslant j \leqslant k$, where b_j is shared random bit, r_j, r'_j are shared random elements, and k is a security parameter such that $(\frac{1}{2})^k$ is negligible.

Since $b_j {r'_j}^2$ is uniformly random, $[c_j]$ can be safely open without leaking any information about a. Assuming c_j is not a zero (If c_j is a zero, the parties discard the c_j and retry), if $a = 0$, it is obvious that $(\frac{c_j}{p}) = (\frac{b_j {r'_j}^2}{p}) = b_j$. While in the case of $a \neq 0$, $(\frac{c_j}{p}) = b_j$ holds with probability $\frac{1}{2}$, as ar_j is uniformly random. By doing the test k times, we can have $s = 1$ for sure when $a = 0$, and $s = 0$ with a high probability $1 - (\frac{1}{2})^k$ for sufficiently large k when $a \neq 0$.

Protocol 3.21 Probabilistic Equality Test $[s] \leftarrow \mathit{ProbEqual}([x], [y])$

Input: $[x], [y]$, where $x, y \in \mathbb{Z}_p$, $p = 3 \mathit{mod}\, 4$, a security parameter k
Output: $[s]$, where $s = (x = y)$, $s \in \{0, 1\} \subset \mathbb{Z}_p$

```
1: $[a] \leftarrow [x] - [y]$
2: foreach $j \in \{1, 2, ..., k\}$ in parallel do
3:     $[b_j] \leftarrow 2 \cdot PRandBit(\mathbb{Z}_p) - 1$
4:     $[r_j] \leftarrow PRand(\mathbb{Z}_p)$
5:     $[r'_j] \leftarrow PRand(\mathbb{Z}_p)$
6:     $[c_j] \leftarrow Mul([a], [r_j]) + MulLogk([b_j], [r'_j], [r'_j])$
7:     $c_j \leftarrow OpenShare([c_j])$
8:     if $c_j = 0$ then
9:         retry: go to Line 3
10:    else
11:        $J_j \leftarrow c_j^{(q-1)/2}$            // Legendre symbol
12:        $[z_j] \leftarrow J_j \cdot 2^{-1} \cdot ([b_j] + J_j)$
13:    end if
14: end for
15: $[s] \leftarrow MulLogk([z_1], [z_2], ..., [z_k])$
```

Complexity Analysis. The complexity of computing each component is as follows: one round and k invocations for generating $[b_j]$, 3 rounds and $4k$ invocations for computing c_j, and $\log(k)$ rounds and $k - 1$ invocations for the last step. The total complexity is $4 + \log(k)$ rounds and $6k - 1$ invocations.

Remark. In protocol *ProbEqual*, the random secret $[r_j]$ is generated to ensure that ar_j is uniformly random, such that probability that $(\frac{c_j}{p}) = b_j$ is exactly $\frac{1}{2}$ in the case of $a \neq 0$. Since the absence of r_j does not harm the privacy of a, we can skip the generation of $[r_j]$, which will save us k invocations, and then $c_j = a + b_j {r'_j}^2$.

It has been shown that $y = a + r$ is a quadratic residue with probability in the range $\frac{1}{2} \pm (3 + \sqrt{p})/p$, where $a \in \mathbb{Z}_p$, r is uniformly random in $\mathbb{Z}_p$ [Per92]. The deviation part $(3 + \sqrt{p})/p$ is negligible when p is a large prime number, which is the typical setting in our

SMC protocols. Thus in our case, we can roughly say that $(\frac{c_j}{p}) = (\frac{a+b_j {r'_j}^2}{p}) = b_j$ holds with probability $\frac{1}{2}$ in the case of $a \neq 0$, and the overall error probability of protocol *ProbEqual* is still negligible when r_j is missing.

3.5.3 Bounded Equality Test

Protocol 3.22 shows how to perform equality test for two bounded secret $[x], [y]$, where $x, y \in [0, 2^m - 1] \subset \mathbb{Z}_p$. The core idea is from the equality test protocol in [NO07]. The parties compute and reveal $c = x - y + r$, where r is a random value. We note that $c = r$ if and only if $x = y$. Therefore, the parties compute whether all bits of c are the same as the bits of r.

In our protocol, since x, y are bounded in range $[0, 2^m - 1]$, we need to add 2^m to avoid modulo reduction in $\mathbb{Z}_p$ when $x < y$. Thus the parties compute and reveal $c = x - y + r + 2^m$, and then test whether all bits of c are the same as $[r']_B$.

Protocol 3.22 Bounded Equality Test $[s] \leftarrow$ *BoundEqual*$([x], [y])$

Input: $[x], [y]$ where $x, y \in [0, 2^m - 1] \subset \mathbb{Z}_p$, $p > 2^{m+k}$, where k is a security parameter
Output: $[s]$, where $s = (x = y)$, $s \in \{0, 1\} \subset \mathbb{Z}_p$

```
1: $[r'], [r']_B \leftarrow PRandBitwise(\mathbb{Z}_p, m)$
2: parse $[r']_B$ as $([r'_{m-1}], ..., [r'_0])$
3: $[r_d] \leftarrow PRand2m(\mathbb{Z}_p, k)$
4: $[r] \leftarrow 2^m \cdot [r_d] + [r']$
5: $c \leftarrow OpenShare([x] - [y] + [r] + 2^m)$
6: foreach $i \in \{0, 1, ..., m - 1\}$ do
7:     $[c'_i] \leftarrow 1 - c_i + (2c_i - 1) \cdot [r'_i]$
8: end for
9: $[s] \leftarrow MulLogk([c'_0], [c'_1], ..., [c'_{m-1}])$
```

Lemma 3.1 *Protocol 3.22 outputs $[s = (x = y)]$ for given $[x], [y]$, provided that $0 \leqslant x < 2^m$ and $0 \leqslant y < 2^m$, $x, y \in \mathbb{Z}_p$.*

Proof We distinguish two cases for Protocol 3.22, namely $x = y$, and $x \neq y$.
If $x = y$, then $c = r + 2^m$, hence $c \equiv r' \mod 2^m$, then $c_i = r'_i$ for $i \in \{0, 1, ..., m - 1\}$.
Hence $c'_i = 2c_i(c_i - 1) + 1 = 1$ for $i \in \{0, 1, ..., m - 1\}$, therefore $s = 1$.
If $x \neq y$, then there exist a bit position t such that $c_i \neq r'_i$, where $t \in \{0, 1, ..., m - 1\}$.
Hence if $c_t = 0$, then $r'_t = 1$, and $c'_t = 1 - c_t + (2c_t - 1)r'_t = 0$, therefore $s = 0$
While if $c_t = 1$, then $r'_t = 0$, and $c'_t = 1 - c_t + (2c_t - 1)r'_t = 0$, therefore $s = 0$
Therefore, Protocol 3.22 outputs $[s = (x = y)]$.

Complexity Analysis. The complexity of computing each component is as follows: one round and m invocations for generating $[r']_B$, one round and one invocation for *OpenShare* and $\log(m)$ rounds and $m - 1$ invocations for computing $[s]$ using *MulLogk*. Thus the overall complexity is $2 + \log(m)$ rounds and $2m$ invocations.

Remark. While Protocol *ProbEqual* has a constant round complexity and communication complexity, which only depends on the security parameter k, the complexity of Protocol *BoundEqual* mainly determined by the bit length m of inputs. Protocol *BoundEqual* is designed for efficiency in the case of small inputs. It performs better than Protocol *ProbEqual* for a small m $(m < k)$.

In the VIFF framework, the security parameter k is usually set to about 30 (to get a negligible fail probability or to guarantee statistical security). Under this setting, Protocol *BoundEqual* is preferred when the bit length of inputs is small than 30, and Protocol *ProbEqual* has advantage for large inputs.

3.5.4 Greater Equal Than Test

Given two bounded secret inputs $[x], [y]$, where $x, y \in [0, 2^m - 1] \subset \mathbb{Z}_p$, Protocol 3.23 computes $[s = (x \geqslant y)]$. This protocol is from [EFG+09], and implemented in VIFF by Toft.

Protocol 3.23 Greater Equal Than Test $[s] \leftarrow GEqualThan([x], [y])$

```
Input: $[x], [y]$, where $x, y \in [0, 2^m - 1] \subset \mathbb{Z}_p$, a security parameter $k$, $p > 2^{m+k} + 2^m$
Output: $[s]$, where $s = (x \geqslant y)$, $s \in \{0, 1\} \subset \mathbb{Z}_p$
 1: $[r'], [r']_B \leftarrow PRandBitwise(\mathbb{Z}_p, m)$
 2: parse $[r']_B$ as $([r'_{m-1}], ..., [r'_0])$
 3: $[r_d] \leftarrow PRand2m(\mathbb{Z}_p, k)$
 4: $[z] \leftarrow [x] - [y] + [r'] + 2^m$
 5: $[d] \leftarrow 2^m \cdot [r_d] + [z]$
 6: $d \leftarrow OpenShare([d])$
 7: get the least $m$ significant bits of $d$, let it be $(d_{m-1}, ..., d_0)$
 8: $[b] \leftarrow PRandBit(\mathbb{Z}_p)$
 9: $[c] \leftarrow 1 - 2 \cdot [b]$
10: $[mask] \leftarrow PRand^*(\mathbb{Z}_p)$
11: $[a_m] \leftarrow 0$
12: foreach $j \in \{m-1, ..., 0\}$ do
13:     $[e_j] \leftarrow [c] + [r'_j] - d_j + 3 \cdot [a_{j+1}]$
14:     $[a_j] \leftarrow [a_{j+1}] + [r'_j] + d_j - 2 \cdot d_j \cdot [r'_j]$
15: end for
16: $[e_m] \leftarrow [c] - 1 + 3 \cdot [a_0]$
17: $[e] \leftarrow MulLogk([mask], [e_0], [e_1], ..., [e_m])$
18: $e \leftarrow OpenShare([e])$
19: $e' \leftarrow (e \neq 0)$
20: $[u] \leftarrow e' + [b] - 2 \cdot e' \cdot [b]$
21: $[s] \leftarrow ([z] - d \mod 2^m - 2^m \cdot [u])/2^m$
```

The overall idea is to masking $x - y$, using a large random number r, and then comparing the bits of x, y. Consider $d' = x - y + 2^m$, it follows that $d'_m = 0 \Leftrightarrow x < y$, where d'_m is the most significant bit of d'. As $d'_m = (d' - (d' \mod 2^m))/2^m$, the problem of comparing x, y becomes to compute $d' \mod 2^m$.

Let $d = d' + r$, and $r = r' + 2^m \cdot r_d$, then

$$d' \mod 2^m = ((d \mod 2^m) - (r \mod 2^m)) \mod 2^m$$

Note that if $(d \mod 2^m) \geqslant (r \mod 2^m)$, then $d' \mod 2^m = (d \mod 2^m) - (r \mod 2^m)$, else $d' \mod 2^m = (d \mod 2^m) - (r \mod 2^m) + 2^m$. We use an underflow bit u to indicate whether $(d \mod 2^m) < (r \mod 2^m)$. Hence

$$\begin{aligned} d'_m &= (d' - (d' \mod 2^m))/2^m \\ &= (x - y + 2^m - (d \mod 2^m) + (r \mod 2^m) - u \cdot 2^m)/2^m \\ &= (x - y + 2^m - (d \mod 2^m) + r' - u \cdot 2^m)/2^m \\ &= (z - (d \mod 2^m) - u \cdot 2^m)/2^m \end{aligned}$$

We now show that step 7 to step 20 in Protocol 3.23 exactly computes $u = (d \mod 2^m) < (r \mod 2^m)$.

Proof If $(d \mod 2^m) > (r \mod 2^m)$, then there exist a bit position t such that $d_i = r'_i$ for $m-1 \geqslant i \geqslant t$, and $d_t = 1, r'_t = 0$;
Hence $a_i = 0$ for $m-1 \geqslant i \geqslant t+1$, $a_t = 3$, $a_i > 0$ for $k-1 \geqslant i \geqslant 0$;
Then $e_i = c$ for $m-1 \geqslant i \geqslant t+1$, $e_t = c-1$, $e_{t-1} = c+3 > 0$ for $t-2 \geqslant i \geqslant 0$, $e_m = c-1+3 \cdot a_0$;
Hence if $b = 0$, then $c = 1$, hence $e_t = 0$, and $e = 0$, then $e' = 0$, thus $u = 0$;
While if $b = 1$, then $c = -1$, hence $e_t = -2$, and $e = 1$, then $e' = 1$, thus $u = 0$.

If $(d \mod 2^m) = (r \mod 2^m)$, then $d_i = r'_i$ for $m-1 \geqslant i \geqslant 0$;
Hence $a_i = 0$ for $m-1 \geqslant i \geqslant 0$, $e_i = c$ for $m-1 \geqslant i \geqslant 0$, $e_m = c-1$;
Hence if $b = 0$, then $c = 1$, hence $e_m = 0$, and $e = 0$, then $e' = 0$, thus $u = 0$;
While if $b = 1$, then $c = -1$, hence $e_m = -2$, and $e \neq 0$, then $e' = 1$, thus $u = 0$.

If $(d \mod 2^m) < (r \mod 2^m)$, then there exist a bit position t such that $d_i = r'_i$ for $m-1 \geqslant i \geqslant t+1$, and $d_t = 0, r_t = 1$;
Hence $a_i = 0$ for $m-1 \geqslant i \geqslant t+1$, $a_t = 3$, $a_i > 0$ for $k-1 \geqslant i \geqslant 0$;
Then $e_i = c$ for $m-1 \geqslant i \geqslant t+1$, $e_t = c+1$, $e_{t-1} = c+3 > 0$ for $t-2 \geqslant i \geqslant 0$, $e_m = c-1+3 \cdot a_0$;
Hence if $b = 0$, then $c = 1$, hence $e_k = 2$, and $e \neq 0$, then $e' = 0$, thus $u = 1$;
While if $b = 1$, then $c = -1$, hence $e_k = 0$, and $e = 0$, then $e' = 1$, thus $u = 1$.

Complexity Analysis. The complexity of computing each component is as follows: one round and m invocations for generating $[r']_B$, one round and one invocation for *OpenShare* and $\log(m+2)$ rounds and $m+1$ invocations for computing $[e]$ using *MulLogk*. Thus the overall complexity is $4 + \log(m+2)$ rounds and $2m+4$ invocations.

3.6 Symmetric Boolean Function

A symmetric Boolean function is a Boolean function whose value does not depend on the permutation of its input bits, i.e., it depends only on the number of ones in the input. This section presents two different solutions for computing symmetric Boolean functions, given k shared inputs $[a_1], [a_2], ..., [a_k]$, where $a_i \in \{0, 1\} \subset \mathbb{Z}_p$ for $1 \leqslant i \leqslant k$, and $k < p - 1$. Section 3.6.1 presents the first solution, which is based on unbounded fan-in multiplication [DFK+06], and the second solution using bounded equality test protocol is given in Section 3.6.2.

3.6.1 Unbounded Fan-in Symmetric Boolean Functions

A symmetric boolean function can be written as $f(x_1, ..., x_k) = \phi(1 + \sum_{i=1}^{k} [x_k])$ for some function $\phi : \{1, 2, ..., k+1\} \to \{0, 1\}$. By using Lagrange polynomial interpolation, we can construct a polynomial of degree k with coefficients $\alpha_0, ..., \alpha_k$ such that $\phi(x) = \sum_{i=0}^{k} \alpha_i x^i$ for all $x \in \{1, 2, ..., k+1\}$. For example, in the case of the boolean function AND, the polynomial satisfies the conditions $\phi(k+1) = 1$ and $\phi(x) = 0$ for $1 \leqslant x \leqslant k$.

Based on the above observations, we can have a generic, efficient solution for secure computation in constant rounds of any symmetric boolean function, as shown in Protocol 3.24 [DFK$^+$06].

Protocol 3.24 Unbounded fan-in symmetric boolean function $[s] \leftarrow SymBool([a_1], [a_2], ..., [a_k])$

Input: $[a_1], [a_2], ..., [a_k]$, where $a_i \in \{0, 1\} \subset \mathbb{Z}_p$ for $1 \leqslant i \leqslant k$
Output: $[s]$ where $s = f(a_1, a_2, ..., a_k)$

```
1: $[b] \leftarrow 1 + \sum_{i=1}^{k} [a_k]$
2: $([b], [b^2], ..., [b^k]) \leftarrow PreMul([b], [b], ..., [b])$
3: $[s] \leftarrow \alpha_0 + \sum_{i=1}^{k} \alpha_i [b^i]$
```

Complexity Analysis. Only step 2 requires interaction, so complexity is the same as protocol *PreMul*: 2 rounds and $3k - 1$ invocations.

3.6.2 Symmetric AND/OR Operation Using Equality Test

Since the result of a symmetric Boolean function only depends only on the number of ones in the input, the symmetric AND/OR operation can be performed using equality test. The idea is that $[s] = AND([a_1], [a_2], ..., [a_k]) = [\sum_{i=1}^{k} [a_i] = k]$, and $[s] = OR([a_1], [a_2], ..., [a_k]) = [1 - (\sum_{i=1}^{k} [a_i] = 0)]$. We propose Protocol 3.25 and Protocol 3.26 for computing symmetric AND and symmetric OR using protocol *BoundEqual* respectively.

Protocol 3.25 symmetric AND operation $[s] \leftarrow AND([a_1], [a_2], ..., [a_k])$

Input: $[a_1], [a_2], ..., [a_k]$, where $a_i \in \{0, 1\} \subset \mathbb{Z}_p$ for $1 \leqslant i \leqslant k$
Output: $[s]$ where $s = \prod_{i=1}^{k} a_i$

```
1: $[s] \leftarrow BoundEqual(\sum_{i=1}^{k} [a_i], k)$
```

Protocol 3.26 symmetric OR operation $[s] \leftarrow OR([a_1], [a_2], ..., [a_k])$

Input: $[a_1], [a_2], ..., [a_k]$, where $a_i \in \{0, 1\} \subset \mathbb{Z}_p$ for $1 \leqslant i \leqslant k$
Output: $[s]$ where $s = \sum_{i=1}^{k} a_i$

```
1: $[s] \leftarrow 1 - BoundEqual(\sum_{i=1}^{k} [a_i], 0)$
```

Complexity Analysis. The above protocols only require a equality test for a bounded value, where protocol *BoundEqual* is used, thus the complexity is $2 + \log(\log(k))$ rounds and $2\log(k)$ invocations. Compared to the 2 rounds and $3k - 1$ invocations required by protocol *SymBool*, our protocol requires much less communication cost, and the round complexity depends on $\log(\log(k))$, which grows slowly when k increases. Overall, our protocols are more efficient than protocol *SymBool*.

3.7 Summary

This chapter provides an overview of basic protocols for Shamir's secret sharing scheme. Most of the protocols are implemented in VIFF by VIFF development team, mainly located in the PassiveRuntime Class in VIFF. The PassiveRuntime Class realizes the basic arithmetic operations and non-interactive random secret generation using PRSS. For integer comparison, Protocol *ProbEqual* and protocol *GEqualThan* are already existed in VIFF. Protocol *BoundEqual* are *AND*, *OR* are proposed and added to VIFF by the author of this thesis.

Table 3.1 gives an overview of the protocol introduced in this chapter.

Protocol	Round	Invocation	Security	Method in VIFF
Mul	1	1	perfect	Passive.mul
MulPub	1	1	perfect	Passive.mul_public
Inner	1	1	perfect	Passive.in_prod
Inv	1	1	perfect	Passive.invert
Rand	1	1	perfect	Not Defined
Rand2m	1	1	statistical	Not Defined
RandInv	3	4	perfect	Not Defined
RandBit	3	3	perfect	Not Defined
PRand	0	0	perfect	Passive.prss_share_random
PRandZero	0	0	perfect	Passive.prss_share_zero
PRand2m	0	0	statistical	Passive.prss_share_random_max
PRandInv	1	1	perfect	Not Defined
PRandBit	1	1	perfect	Passive.prss_share_random
PRandBitwise	1	m	perfect	Not Defined
EqualPub	2	2	perfect	Passive.equal_public
ProbEqual	$4 + \log(k)$	$6k - 1$	statistical	ProbabilisticEqualityMixin
BoundEqual	$2 + \log(m)$	$2m$	statistical	Not Defined
GEqualThan	$4 + \log(m + 2)$	$2m + 4$	statistical	ComparisonToft07Mixin
SymBool	2	$3k - 1$	perfect	Not Defined
AND, *OR*	$2 + \log(\log(k))$	$2\log(k)$	statistical	Not Defined

Table 3.1: Overview of basic protocols in Chapter 3

Chapter 4

Secure Frequent Itemset Mining

The goal of frequent itemset mining is to discover sets of items that frequently co-occur in the transactional database. The original motivation for mining frequent itemset came from market basket analysis, where the basic idea is to examine the customer behavior in terms of the purchased products, and find out which items are frequently purchased together.

Frequent itemsets mining is a key component of many data mining tasks that rely on finding frequent patterns such as association rules learning, sequence mining. This chapter gives two secure frequent itemset mining solutions based on secure multiparty computation. We start with the introduction of basic concepts of frequent itemset and Apriori algorithm in Section 4.1. Section 4.2 presents a secure Apriori protocol which outputs public frequent itemsets. In Section 4.3, a fully secure Apriori protocol which generate secret outputs is given.

4.1 Frequent Itemset Mining

The problem of frequent itemset mining is defined as follows [AIS93]. Given a set of items $I = \{I_1, I_2, ..., I_n\}$ and a transactional database $\boldsymbol{T} = \{T_1, T_2, ..., T_m\}$ where each transaction T_i consists of items from I. An itemset is a set which consists of items from I. The support $\mathit{supp}(X)$ of an itemset X is defined as the number of transactions in T that contain the itemset X. An itemset is called frequent if its support is larger than a specified minimum support $\mathit{MinSupp}$. The goal of frequent itemset mining is to find all frequent itemsets. Table 4.1 shows an example transactional database and its frequent itemset [Lov12].

Transaction	S1	S2	S3	S4
T1	1	0	1	0
T2	0	1	0	0
T3	0	0	0	1
T4	0	1	1	1
T5	0	1	1	0
T6	0	1	1	0
T7	1	1	1	1
T8	1	0	1	0
T9	1	1	1	0
T10	1	1	1	0

Frequent Itemset	Support
(S1)	5
(S2)	7
(S3)	8
(S4)	3
(S1, S2)	3
(S1, S3)	5
(S2, S3)	6
(S2, S4)	2
(S3, S4)	3
(S1, S2, S3)	3
(S2, S3, S4)	2

Table 4.1: An example transactional database and its frequent itemsets

In the above example, the set of items I is $\{S1, S2, S3, S4\}$, and the minimum support *MinSupp* is set to 2. Note that a frequent itemset has the property that any subset of a frequent itemset is also frequent. In other words, if an itemset is not frequent, none of its supersets are frequent. For example, all subsets of frequent itemset $(S1, S2, S3)$ are also frequent.

There are many algorithms proposed in literature for frequent itemset mining, of which two well known algorithms are Apriori and Eclat. It has been shown in [Jag10] that Apriori is faster and better parallelizable than Eclat, thus we choose Apriori algorithm as the base to design protocols for secure frequent itemset mining.

4.1.1 Apriori Algorithm

Apriori is probably the best-known algorithm to mine frequent itemsets, which uses a breadth-first search strategy to count the support of itemsets [AS94]. Let L_k be set of frequent k-itemsets, and C_k be set of candidate k-itemsets, where k-itemsets is a itemset having k items. Each member in L_k and C_k has two two fields: itemset and support.

The basic idea of Apriori algorithm is that it first generates the set of frequent 1-itemsets, and then runs the following cycle. Given the set of frequent $(k-1)$-itemsets L_{k-1}, a candidate generation function is used to generate a candidate set of potentially frequent k-itemsets C_k, and then verifies which of those candidates are really frequent. Algorithm 4.1 presents the pseudocode of Apriori, and the candidate generation function is presented in Algorithm 4.2.

Algorithm 4.1 Apriori algorithm for frequent itemsets mining

```
1: $L_1$ = frequent 1-itemsets, $k \leftarrow 2$
2: while $L_{k-1} \neq \emptyset$ do
3:   $C_k \leftarrow AprioriGen(L_{k-1})$ // Generate new candidates
4:   foreach transaction $t \in T$ do
5:     $C_t \leftarrow subset(C_k, t)$      // Candidates contained in $t$
6:     for all candidates $c \in C_t$, $c.support \leftarrow c.support + 1$
7:   end for
8:   $L_k \leftarrow \{c \in C_k | c.support \geqslant MinSupp\}$
9:   $k \leftarrow k + 1$
10: end while
11: return $\bigcup_k L_k$
```

Algorithm 4.2 Apriori candidate generation algorithm $C_k \leftarrow AprioriGen(L_{k-1})$

```
1: insert into $C_k$, select $p.item_1, ..., p.item_{k-1}, q.item_{k-1}$ from $p, q \in L_{k-1}$
   where $p.item_1 = q.item_1, ..., p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$
2: foreach itemset $c \in C_k$ do
3:   foreach (k-1)-subset $s \subset c$ do
4:     if $s \notin L_{k-1}$, delete $c$ from $C_k$
5:   end for
6: end for
7: return $C_k$
```

4.2 Secure Apriori Protocol

This section present a secure Apriori protocol for secure frequent itemset mining, based on the *Apriori* algorithm. The input is a shared transactional database, as shown in Table 4.2, and the outputs are public frequent itemsets.

Transactions	Item I_1	Item I_2	$\cdots$	Item I_n
$[T_1]$	$[t_{11}]$	$[t_{12}]$	$\cdots$	$[t_{1n}]$
$[T_2]$	$[t_{21}]$	$[t_{22}]$	$\cdots$	$[t_{2n}]$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$[T_m]$	$[t_{m1}]$	$[t_{m2}]$	$\cdots$	$[t_{mn}]$

Table 4.2: A secret shared transactional database

Let $I = \{I_1, I_2, ..., I_n\}$ be the given set of items, and $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$ be the shared transactional database, which consists of transactions from all parties. The parties use Shamir's Secret Sharing Scheme as described in Section 2.2.2 to securely share their own transaction data. Each shared transaction $[T_i] = \{[t_{i1}], [t_{i2}], ..., [t_{in}]\}$ is a n-bit-vector, where t_{ij} is a boolean value (0 or 1) which represents the presence or absence of item I_j in transaction T_i. For convenience, we define item vector $[T_{:i}]$ as the i-th column of the $[\boldsymbol{T}]$, i.e., $[T_{:i}] = \{[t_{1i}], [t_{2i}], ..., [t_{mi}]\}$.

Let C_k be the set of candidate frequent k-itemset, L_k be the set of frequent k-itemset, and $[S_k]$ be the set of support values for the itemsets in L_k, for $1 \leqslant k \leqslant n$.

4.2.1 Calculate the Support of an Itemset

In the original Apriori algorithm, the support of an itemset is obtained by traversing the whole database T, which is inefficient in the case of secure frequent itemset mining, as the database $[T]$ is not public. To efficiently compute the support of an itemset, Protocol 4.3 can be used. The basic idea is that the support of an itemset is exactly the inner product of item vectors for each item in the itemset. For example, the support of itemset (S1, S2) in Table 4.1 is 3, which can be calculated as $(1, 0, 0, 0, 0, 0, 1, 1, 1, 1) \cdot (0, 1, 0, 1, 1, 1, 1, 0, 1, 1) = 3$.

Protocol 4.3 Calculate the Support of an Itemset $[supp] \leftarrow Supp([\boldsymbol{T}], itemset)$

Input: $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$, $I = \{I_1, I_2, ..., I_n\}$, $itemset = \{item_1, ..., item_l\}$
Output: $[supp]$

```
 1: get the first item in $itemset$, let it be $I_x$. compute $[v] \leftarrow [T_{:x}]$
 2: if $l = 1$ then
 3:     $[supp] \leftarrow \sum[v]$
 4: else
 5:     for $2 \leqslant j \leqslant l - 1$ do
 6:         get the $j$-th item in $itemset$, let it be $I_y$
 7:         for $1 \leqslant i \leqslant m$ do
 8:             $[v_i] \leftarrow Mul([v_i], [t_{iy}])$
 9:         end for
10:     end for
11:     get the last item in $itemset$, let it be $I_z$. compute $[supp] \leftarrow Inner([v], [T_{:z}])$
12: end if
```

Complexity Analysis. Protocol 4.3 requires $m(l - 2) + 1$ invocations for $l \geqslant 1$.

4.2.2 Check Minimum Support Requirement

Protocol *CkSupp* is used to select frequent k-itemsets from a set of candidate k-itemsets, by checking whether the support value is larger than the minimum support *MinSupp*. It outputs the set of frequent k-itemsets L_k, and the set of support values $[S_k]$ for the itemsets in L_k

Protocol 4.4 Check Minimum Support Requirement $L_k, [S_k] \leftarrow CkSupp(C_k, MinSupp)$

```
Input: $C_k$, $MinSupp$
Output: $L_k, [S_k]$
1: $L_k \leftarrow \emptyset, [S_k] \leftarrow \emptyset$
2: foreach $itemset \in C_k$ do
3:     $[supp] \leftarrow Supp([\boldsymbol{T}], itemset)$
4:     if $OpenShare(GEqualThan([supp], MinSupp))$ then
5:         $L_k \leftarrow L_k \cup itemset$
6:         $[S_k] \leftarrow [S_k] \cup [supp]$
7:     end if
8: end for
9: return $L_k, [S_k]$
```

4.2.3 Secure Apriori Protocol

The secure Apriori protocol is presented as Protocol 4.5, which outputs all the frequent itemsets $\boldsymbol{L}$, given the shared transactional database $[\boldsymbol{T}]$, set of items I, and minimum support *MinSupp*. The support values of frequent itemsets are stored in $[\boldsymbol{S}]$, which can be reused for other mining tasks, for example, the association rule learning, which is shown in 4.2.4.

In the protocol, *AprioriGen* algorithm is reused to generate the candidates set. This is possible because the itemsets are public, only the support values are secret.

Protocol 4.5 Secure Apriori protocol $\boldsymbol{L}, [\boldsymbol{S}] \leftarrow SApriori([\boldsymbol{T}], I, MinSupp)$

```
Input: $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$, $I = \{I_1, I_2, ..., I_n\}$, $MinSupp$
Output: $\boldsymbol{L} = \{L_1, L_2, ...\}, [\boldsymbol{S}] = \{[S_1], [S_2], ...\}$
1: $C_1 \leftarrow \{\{I_1\}, \{I_2\}, ..., \{I_n\}\}$
2: $L_1, [S_1] \leftarrow CkSupp(C_1, MinSupp)$
3: $k \leftarrow 2$
4: while the size of $L_{k-1}$ is greater equal than $k$ do
5:     $C_k \leftarrow AprioriGen(L_{k-1})$
6:     $L_k, [S_k] \leftarrow CkSupp(C_k, MinSupp)$
7:     $k \leftarrow k + 1$
8: end while
9: return $\boldsymbol{L} = \{L_1, L_2, ...\}, [\boldsymbol{S}] = \{[S_1], [S_2], ...\}$
```

Security Analysis. Protocol 4.5 is secure in the sense that the input transactions are secret, no information is leaked during the protocol running, except that the frequent itemsets are public. An adversary can also see the order of frequent itemsets, namely the finding process of each frequent itemset, but this does not harm privacy, as the information can also be deduced from the output.

4.2.4 Association Rule Learning

The result of secure Apriori protocol can be directly used for mining association rules. An association rule is defined as an implication of the form $X \Rightarrow Y$, where X, Y are the subsets of a frequent itemset Z, and $X, Y \neq \emptyset$, $X \cup Y = Z$. The confidence of a rule is defined as $conf(X \Rightarrow Y) = supp(X \cup Y)/supp(X)$. We are interested in the association rules whose confidence is larger than a specified confidence threshold $MinConf$.

Protocol 4.6 shows how to check the confidence requirement for a rule, assuming Z is a frequent itemset, X is a subset of Z. Since the itemsets are public, the support of a frequent itemset can be obtained by looking up its index in the set $\boldsymbol{L}$, and get the element from $[\boldsymbol{S}]$ with the same index.

Protocol 4.6 Check the confidence threshold $R \leftarrow CkConf(X, Z, MinConf, \boldsymbol{L}, [\boldsymbol{S}])$

```
1: get the support of $X$, let it be $[supp_X]$
2: get the support of $Z$, let it be $[supp_Z]$
3: if $OpenShare(GEqualThan([supp_Z], MinConf \cdot [supp_X]))$ then
4:     return $R \leftarrow (X, Z \backslash X)$
5: else
6:     return $R \leftarrow NULL$
7: end if
```

To find all the association rules from a secret shared transactional database $[\boldsymbol{T}]$, Protocol 4.7 is used. It first computes all frequent itemsets with the secure Apriori protocol, and then generate and check the association rules for each frequent itemset.

Protocol 4.7 Association rule learning $[\boldsymbol{R}] \leftarrow Associ([\boldsymbol{T}], I, MinSupp, MinConf)$

```
Input: $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$, $I = \{I_1, I_2, ..., I_n\}$, $MinSupp$, $MinConf$
Output: $\boldsymbol{R} = \{R_1, R_2, ...\}$, where $R_i = (X, Y)$
1: $\boldsymbol{L}, [\boldsymbol{S}] \leftarrow SApriori([\boldsymbol{T}], I, MinSupp)$
2: $k \leftarrow 2$, $\boldsymbol{R} \leftarrow \emptyset$
3: while $L_k \neq \emptyset$ do
4:     foreach $Z \in L_k$ do
5:         foreach $j \in \{1, 2, ..., k-1\}$ do
6:             get all the subset of $I$, with size $j$, let it be $Sub_j$
7:             for each $X \in Sub_j$, $\boldsymbol{R} \leftarrow \boldsymbol{R} \cup CkConf(X, Z, MinConf, \boldsymbol{L}, [\boldsymbol{S}])$
8:         end for
9:     end for
10:    $k \leftarrow k + 1$
11: end while
12: return $\boldsymbol{R}$
```

4.3 Fully Secure Frequent Itemset Ming

This section presents a fully secure frequent itemset mining solution in which the itemsets are kept secret instead of public. To hide the content of an itemset, we use a shared n-bit-vector to the itemset, where n is the total number of items that can appear in the database. The shared n-bit-vector consists of boolean values (0 or 1) which represent the presence or absence of items in the itemset. For example, the itemset $(S1, S2)$ in Table 4.1 is represented by a 4-bit-vector $(1, 1, 0, 0)$.

Notation. Let $I = \{I_1, I_2, ..., I_n\}$ be the given set of items, and $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$ be the shared transactional database, as explained in Section 4.2. As the itemsets become secret, we use $[C_k]$ to denote the set of candidate k-itemset, and $[L_k]$ to denote the set of frequent k-itemset, for $1 \leqslant k \leqslant n$. The support values of frequent k-itemsets are stored in $[S_k]$.

4.3.1 Secure Candidates Generation

Since the itemsets are secret, the *AprioriGen* function in original Apriori algorithm cannot be reused now. To securely generate candidate itemsets, Protocol 4.9 is used. The basic idea follows from the property of frequent itemset that any subset of a frequent itemset is also frequent. For a potential frequent k-itemset, it can be represented by the union of $(k-1)$ frequent $(k-1)$-itemsets.

Let the set of frequent $(k-1)$-itemset be $[L_{k-1}]$ that having a itemsets. To generate the set of candidate k-itemset, we first compute $\binom{a}{k-1}$ itemsets, where each itemset is the union of $k-1$ frequent $(k-1)$-itemset. A candidate k-itemset should have the size of k, i.e., the sum of the shared n-bit-vector which represent the itemset should equal to k. Thus we verify the size for each obtained itemset, and remove those itemsets whose size are not k. The union of sets can be obtained by Protocol 4.8.

Protocol 4.8 Union of sets $[y] \leftarrow \mathit{SetUnion}([A])$

```
Input: $[A] = \{[A_1], [A_2], ...[A_a]\}$, where $[A_i] = \{[A_{i1}], [A_{i2}], ..., [A_{in}]\}$ is a shared $n$-bit-vector
Output: $[y] = \{[y_1], [y_2], ..., [y_n]\}$
 1: foreach $j \in \{1, 2, ..., n\}$ do
 2:     $[y_j] \leftarrow OR(\{[A_{1j}], [A_{2j}], ..., [A_{aj}]\})$
 3: end for
```

Protocol 4.9 Secure candidate generation $[C_k] \leftarrow \mathit{SGen}([L_{k-1}])$

```
Input: $[L_{k-1}] = \{[Z_1], [Z_2], ...[Z_a]\}$, where $[Z_i]$ is itemset represented by a shared $n$-bit-vector
Output: $[C_k] = \{[c_1], [c_2], ...\}$
 1: $[C_k] \leftarrow \emptyset$
 2: initialize a set $s = (1, 2, ..., a)$, let $B = \{B_1, .., B_t\}$ be all $t = \binom{a}{k}$ combinations from $s$
    where $B_i = \{B_{i1}, B_{i2}, ..., B_{ik}\}$ is a set of $k$ elements from $s$
 3: foreach $i \in \{1, 2, ..., t\}$ do
 4:     $[A] \leftarrow \{[Z_{B_{i1}}], [Z_{B_{i2}}], ..., [Z_{B_{ik}}]\}$
 5:     $[\mathit{itemset}] \leftarrow \mathit{SetUnion}([A])$
 6:     $\mathit{sum} \leftarrow \mathit{OpenShare}(\sum[\mathit{itemset}])$
 7:     if $\mathit{sum} = k$ then
 8:         $[C_k] \leftarrow [C_k] \cup [\mathit{itemset}]$
 9:     end if
10: end for
```

4.3.2 Calculate the Support of an Itemset

To calculate the support of an itemset, we traverse the whole database to check whether the itemset is a subset of a transaction. Protocol 4.10 shows how to compute $[s = (A \subset B)]$, assuming A, B are two shared n-bit-vector. The basic idea is that if $A \subset B$, then the inner product of A, B equal to the sum of A.

Protocol 4.10 Check Subset Relationship $([s]) \leftarrow \mathit{SubSet}([A], [B])$

```
Input: $[A] = \{[a_1], [a_2], .., [a_n]\}$, $[B] = \{[b_1], [b_2], .., [b_n]\}$
Output: $[s]$
1: $[c] \leftarrow \mathit{Inner}([A], [B])$
2: $[s] \leftarrow \mathit{BoundEqual}([c], \sum[A])$
```

Using protocol *SubSet*, the the support of an itemset can be obtained as shown in Protocol 4.11.

Protocol 4.11 Calculate support of itemset $([\mathit{supp}]) \leftarrow \mathit{SSupp}([\boldsymbol{T}], [\mathit{itemset}])$

```
Input: $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$, $[\mathit{itemset}]$
Output: $[\mathit{supp}]$
1: $[\mathit{supp}] \leftarrow 0$
2: foreach $i \in \{1, ..., m\}$ do
3:     $[\mathit{supp}] \leftarrow [\mathit{supp}] + \mathit{SubSet}([\mathit{itemset}], [T_i])$
4: end for
```

4.3.3 Check Minimum Support Requirement

To check minimum support requirement, Protocol 4.12 is used. The idea is the same as protocol *CkSupp*, except that the frequent itemset are kept secret, and the support of an itemset is calculated by using protocol *SSupp*. It outputs the set of frequent k-itemsets $[L_k]$, and the set of support values $[S_k]$ for the itemsets in $[L_k]$

Protocol 4.12 Check minimum support $[L_k], [S_k] \leftarrow \mathit{SCkSupp}([\boldsymbol{T}], [C_k], \mathit{MinSupp})$

```
1: $[L_k] \leftarrow \emptyset, [S_k] \leftarrow \emptyset$
2: foreach $[\mathit{itemset}] \in [C_k]$ do
3:     $[\mathit{supp}] \leftarrow \mathit{SSupp}([\boldsymbol{T}], [\mathit{itemset}])$
4:     if $\mathit{OpenShare}(\mathit{GEqualThan}([\mathit{supp}], \mathit{MinSupp}))$ then
5:         $[L_k] \leftarrow [L_k] \cup [\mathit{itemset}]$
6:         $[S_k] \leftarrow [S_k] \cup [\mathit{supp}]$
7:     end if
8: end for
9: return $[L_k], [S_k]$
```

4.3.4 Fully Secure Apriori Protocol

The fully secure Apriori protocol is presented as Protocol 4.13, which outputs all the frequent itemsets $[\boldsymbol{L}]$, and their support values $[\boldsymbol{S}]$. We first randomly add the single-item candidate itemset $\{\{[I_1]\}, \{[I_2]\}, ..., \{[I_n]\}\}$ to the set of size one candidate itemset $[C_1]$. And then go through the Apriori algorithm, but in a secure fashion. The output of fully secure Apriori protocol is secret shared frequent itemset, and their secret shared support.

Protocol 4.13 Fully secure Apriori protocol $[\boldsymbol{L}] \leftarrow SSApriori([\boldsymbol{T}], I, MinSupp)$

```
Input: $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_m]\}$, $I = \{I_1, I_2, ..., I_n\}$, $MinSupp$
Output: $[\boldsymbol{L}] = \{[L_1], [L_2], ...\}, [\boldsymbol{S}] = \{[S_1], [S_2], ...\}$
1: $[C_1] \leftarrow \{\{[I_1]\}, \{[I_2]\}, ..., \{[I_n]\}\}$ // the order of itemset in $C_1$ is randomized

2: $[L_1], [S_1] \leftarrow SCkSupp([\boldsymbol{T}], [C_1], MinSupp)$
3: $k \leftarrow 2$
4: while the size of $[L_{k-1}]$ is greater equal than $k$ do
5:     $[C_k] \leftarrow SGen(L_{k-1})$
6:     $[L_k], [S_k] \leftarrow SCkSupp([\boldsymbol{T}], [C_k], MinSupp)$
7:     $k \leftarrow k + 1$
8: end while
9: return $[\boldsymbol{L}] = \{[L_1], [L_2], ...\}, [\boldsymbol{S}] = \{[S_1], [S_2], ...\}$
```

Security Analysis. Protocol 4.13 is fully secure in the sense that both the input and output are secret. An adversary can only know the number of frequent itemset, no information about the itemset itself and its support is leaked through the computation. The secret output can be used as secret input for other data mining task, such as association rules learning, or opened public to a set of specified parties.

4.4 Performance Result

To test the performance of our protocols, we implemented them in VIFF, and conducted a benchmarking experiment with the 'SPECT' datset and 'KRKPA7' datset from UCI Machine Learning Repository [FA10]. We measure the overall execution time of our programs under the experiment settings introduced in Section 2.3. The experiment result is shown in Table 4.3.

The 'SPECT' datset consists of 267 transaction, with 23 attributes, and the 'KRKPA7' datset consists of 3196 transaction, with 37 attributes (for detail description of the dataset, see Section 5.3). For our experiment, we did some preprocessing to make each attribute binary, such that we can treat each attribute as an item. And we choose 1000 transactions fromp the 'KRKPA7' datset to form a subsets 'KRKPA7-1000', as the 'KRKPA7' datset is too big for our experiment.

Data Set	MinSupp	*SApriori*	*Associ*	*SSApriori*
SPECT-267	50%	1.94s	1.94s	42.05s
SPECT-267	40%	4.40s	4.58s	69.07s
SPECT-267	30%	19.25s	23.98s	218.91s
SPECT-267	25%	41.06s	54.90s	378.64s
KRKPA7-1000	50%	11.16s	13.32s	494.85s
KRKPA7-1000	40%	13.92s	17.50s	560.47s
KRKPA7-1000	30%	28.79s	41.83s	934.48s

Table 4.3: Performance Result of Secure Frequent Itemset Mining

We set different minimum support value (as the percentage of the size of data set) for each data set. As expected, the running time of our protocols increases when the minimum support value goes down. This is because the smaller the minimum support value is, the more frequent itemsets needed to be computed. There is a natural tradeoff between privacy and performance, the price paid for obtaining a fully secure frequent itemset mining solution in Protocol $SSApriori$ is significantly increased execution time.

Chapter 5

Secure Decision Tree Learning

Decision tree learning is a technique commonly used in data mining to create a predictive model which maps observations about an item to conclusions about the item's target value. Decision trees used in data mining are of two main types, regression tree and classification tree. For regression tree analysis, the predicted outcome can be a real number (numerical value), while for classification tree analysis, the predicted outcome is a class (categorical value). This thesis focuses on classification tree analysis, or classification by decision tree learning.

In this chapter, we first introduce the ID3 algorithm and splitting measures for building a decision tree in Section 5.1, and then present our secure ID3 protocol for privacy preserving decision tree learning in Section 5.2. Section 5.3 shows the benchmarking results of our protocols.

5.1 Decision Tree Learning

In decision tree learning, the input is a transactional database, that is, a database consist of a set of transactions. Each transaction is described in terms of a set of non-class attributes and a class attribute, where each attribute has a set of categorical attribute values that can appear in each transaction. Table 5.1 shows an example transactional database containing 14 transactions [Qui86, WFH11]. There are 4 non-class attributes: 'Outlook', 'Temperature', 'Humidity', 'Windy', and a class attribute 'Play Tennis'.

Day	Outlook	Temperature	Humidity	Windy	Play Tennis
1	sunny	hot	high	false	no
2	sunny	hot	high	true	no
3	overcast	hot	high	false	yes
4	rainy	mild	high	false	yes
5	rainy	cool	normal	false	yes
6	rainy	cool	normal	true	no
7	overcast	cool	normal	true	yes
8	sunny	mild	high	false	no
9	sunny	cool	normal	false	no
10	rainy	mild	normal	false	yes
11	sunny	mild	normal	true	yes
12	overcast	mild	high	true	yes
13	overcast	hot	normal	false	yes
14	rainy	mild	high	true	no

Table 5.1: The Weather Problem

Given a transactional database, the aim of decision tree learning is to build a decision tree that can be used to classify new transactions (predict the class attribute value for new transactions by viewing only the non-class attributes values). A decision tree consists of nodes and edges that connect nodes. Each non-leaf node is a test node and corresponds to a non-class attribute, and the edges correspond to the possible values taken on by that attribute. Figure 5.1 shows the decision tree built from the database in Table 5.1.

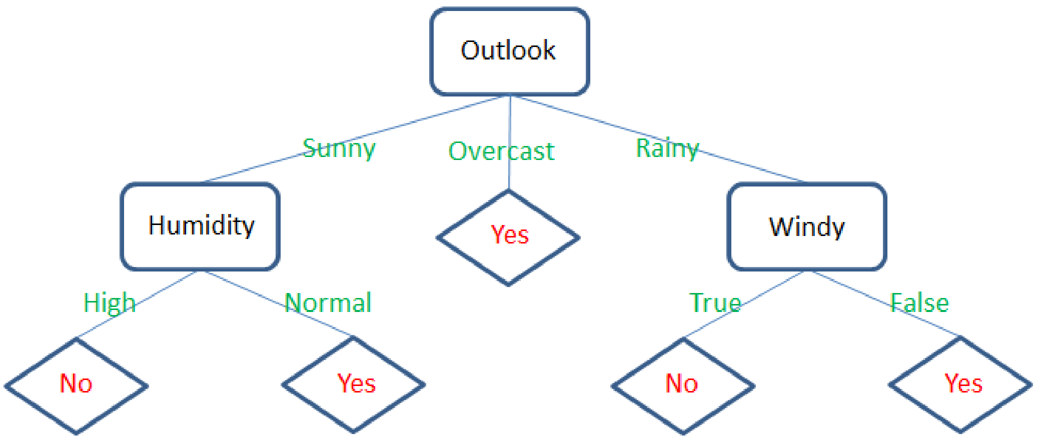

Figure 5.1 The Decision Tree for Weather Problem

To make a decision, one starts at the root node, at each non-leaf node in the tree, test the item's corresponding attribute value to determine which edge to follow, until one reaches a leaf node where a prediction can be made. A leaf node corresponds to the expected value of the class attribute for a decision path from the root node to that leaf node. An example decision path from Figure 5.1 would be:

If ('Outlook' = Sunny) **and** ('Humidity' = Normal) **then** 'Play Tennis' = Yes

5.1.1 ID3 Algorithm

ID3 (Iterative Dichotomiser 3) is a popular algorithm for decision tree learning, proposed by Ross Quinlan [Qui86]. The overall approach of ID3 algorithm is to choose the attribute that best classifies the transactions into their classes and then partition the transactions set according to the values of that attribute. This process is recursively applied to each of the subsets produced until "pure" nodes are found - a pure node contains elements of only one class - or until there are no attributes left to consider.

Let T be the transactions set, R be the non-class attribute set, C be the class attribute. The ID3 algorithm is shown as Algorithm 5.1.

Algorithm 5.1 ID3 Algorithm: ID3(R, C, T)

```
1:  if $R$ is empty then
2:    return a leaf node tree with the class value assigned to most transactions in $T$
3:  else if $T$ consists of transactions which all have the same class value $c$ then
4:    return a leaf node with the value $c$
5:  else
6:    Determine the best attribute $A$ that classifies the transactions in $T$
7:    Create a non-leaf node for attribute $A$, let $A = \{a_1, ..., a_m\}$
8:    foreach $a_i \in A$ do
9:      let $T(a_i)$ be the subset of $T$ that have the value $a_i$ for attribute $A$
10:     add edge $a_i$ to the node $A$, and go to ID3$(R - A, C, T(a_i))$
11:   end for
12: end if
```

The main task in ID3 algorithm is to determine the best attribute A that classifies the transactions in T, at each node in the development of a decision tree. This relies on a measure for goodness of split, that is, how well an attribute discriminates classes. To calculate an attribute's goodness of split at a particular node in the tree, the transactions set T can be set out to a contingency table for the attribute [Min89]. Let $A = \{a_1, ..., a_m\}$, $C = \{c_1, ..., c_n\}$, the contingency table is illustrated in Table 5.2. The element x_{ij} in the table corresponds to the number of transactions in T that have attribute value a_i for attribute A and class value c_j for class C. $x_{i:}$ denotes the sum of i-th row elements, and $x_{:j}$ is the sum of j-th column elements. The total number of transactions in T is N.

	c_1	c_2	...	c_n	Total
a_1	x_{11}	x_{12}	...	x_{1n}	$x_{1:}$
a_2	x_{21}	x_{22}	...	x_{2n}	$x_{2:}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
a_m	x_{m1}	x_{m2}	...	x_{mn}	$x_{m:}$
Total	$x_{:1}$	$x_{:2}$	...	$x_{:n}$	N

Table 5.2: The Contingency Table

Given a contingency table, we can have an overview of the attribute's goodness of split. A perfect attribute A would have each attribute value a_i associated with only one class value c_j, i.e., $x_{ij} = x_{i:}$, namely that all rows in the contingency table would have only one non-zero element. And a useless attribute would have the same elements in each row, i.e., $x_{i2} = x_{i2} = ... = x_{in}$, for $1 \leqslant i \leqslant m$, therefore the attribute can hardly split the transactions set.

5.1.2 Splitting Measures

This section reviews various splitting measures for calculating an attribute's goodness of split. All these measures can be calculated from the contingency table [Min89]. We start with Quinlan's information measure, which is the default and popular splitting measure used in ID3 algorithm [Qui86].

Information Measure

Information Measure computes the information gain based on entropy formula from information theory: $Entropy(X) = -\sum_{i=1}^{n} P_r(X = x_i) \log P_r(X = x_i)$ for a discrete random variable X

with possible values $\{x_1, ..., x_n\}$. First the entropy of the total transactions set T is calculated, and then the entropy for each attribute value a_i is calculated, which is added proportionally to get total entropy for the split. Information Measure (IM) is then defined as the gain in information, or decrease in entropy, brought by knowledge of the attribute:

$$\begin{aligned} IM &= Entropy(T) - \sum_{i=1}^{m} \frac{|T(a_i)|}{|T|} Entropy(T(a_i)) \\ &= \frac{1}{N}(\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}\,\mathrm{log}x_{ij} - \sum_{i=1}^{m} x_{i:}\,\mathrm{log}x_{i:} - \sum_{j=1}^{n} x_{:j}\,\mathrm{log}x_{:j} + N\log N) \end{aligned} \tag{5.1}$$

The attribute that yields the largest IM is selected as the best attribute for splitting. The disadvantage of IM is that the transactions may be over-classified, since it has no concept of statistical significance, therefore it attempts to produce a decision tree able to classify every single transaction [Min89].

Since Quinlan's original work, there have been a number of alternative splitting measures, which can be used to replace IM in ID3 algorithm. Some of them are variant of IM, which are also based on entropy function, for example, the G statistic, Gain-ratio measure [Min89]. Since it is difficult to implement a secure logarithm function, we will not use the entropy based measure for our secure ID3 protocol. Instead of these, there are two simple and efficient ones, Chi-squapre statistic χ^2 and GINI index.

Chi-square (χ^2) statistic

It has been shown in [Min87] that χ^2 can be used as a splitting measure to select the best attribute. The Chi-square (χ^2) statistic calculates the degree of association between the non-class attribute and class attribute. A larger χ^2 indicates greater association, namely the attribute discriminate the transactions well. Using the notations in Table 5.2, the formula for this function is defined as follows:

$$\chi^2 = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{(x_{ij} - E_{ij})^2}{E_{ij}} = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{(Nx_{ij} - x_{i:}x_{:j})^2}{Nx_{i:}x_{:j}} \tag{5.2}$$

where $E_{ij} = x_{i:}x_{:j}/N$, i.e., the expected value for each element in the contingency table. Comparing to Information Measure, the Chi-square (χ^2) statistic favors two-valued attributes, which leading to very narrow and large trees with many levels [Min87]. However, it is much easier to compute, since the function only involves addition and multiplication operation, which can be effectively implemented in secure multi-party computation.

Gini Index

Breiman et al. employed another measure called Gini Index to determine the best attribute in [BFOS84]. They defined a Gini function to measures the 'impurity' of an attribute with respect to the classes. Given the probability distribution $P_i = P_r(X = x_i)$ of a discrete random variable X with possible values $\{x_1, ..., x_n\}$, the Gini function is $Gini(X) = 1 - \sum_{i=1}^{n} P_i^2$. Using the notations in Table 5.2, the Gini Index (GI) for the given attribute is defined as the $Gini$ value of total transactions set T minus the weighted $Gini$ value of each subset $T(a_i)$:

$$GI = Gini(T) - \sum_{i=1}^{m} \frac{|T(a_i)|}{|T|} Gini(T(a_i)) = \frac{1}{N}(\sum_{i=1}^{m}\sum_{j=1}^{n} \frac{{x_{ij}}^2}{x_{i:}} - \sum_{j=1}^{n} \frac{{x_{:j}}^2}{N}) \tag{5.3}$$

Summary
Note that the row total $x_{i:} = \sum_{j=1}^{n} x_{ij}$ and column totals $x_{:j} = \sum_{i=1}^{m} x_{ij}$ could be zero, i.e., all elements in the i-th row or j-th column of the contingency table are zero. In this case, the Entropy function and Gini function have no definition, as the variable X is no longer a discrete random variable. And for the Chi-square (χ^2) statistic, $\sum\limits_{j=1}^{n} \frac{(x_{ij}-E_{ij})^2}{E_{ij}}$ is indeterminate, as E_{ij} becomes zero. Thus for Equation 5.1, 5.2, 5.3, the i-th row elements x_{ij} are excluded from calculation, if $x_{i:} = 0$, and the j-th row elements x_{ij} are excluded from calculation, if $x_{:j} = 0$

It has been shown that GI and IM are very similar, the difference between these two is that the GI tries to create pure nodes with larger subset, while IM normally tries to create a balanced tree [SM08]. The Gini Index is also simpler compared to Information Measure, and is favorable to be used for a secure ID3 algorithm. We will implement both Chi-square χ^2 statistic and Gini Index in our Secure ID3 Protocol, which is presented in the next section.

For the given example transaction set in Table 5.1, the contingency tables and splitting measure values for attribute 'Outlook', 'Temperature' , 'Humidity' and 'Windy' is shown in Table 5.3. It is clear that all measures prefer 'Outlook' as the best attribute for splitting.

Outlook	Yes	No	Total	Temperature	Yes	No	Total
Sunny	2	3	5	Hot	2	2	4
Overcast	4	0	4	Mild	4	2	6
Rain	3	2	5	Cool	3	1	4
Total	9	5	14	Total	9	5	14
Measures	$IM = 0.25, \chi^2 = 3.55, GI = 0.12$			Measures	$IM = 0.03, \chi^2 = 0.57, GI = 0.02$		
Humidity	Yes	No	Total	Windy	Yes	No	Total
High	3	4	7	True	6	2	8
Normal	6	1	7	False	3	3	6
Total	9	5	14	Total	9	5	14
Measures	$IM = 0.15, \chi^2 = 2.8, GI = 0.09$			Measures	$IM = 0.05, \chi^2 = 0.93, GI = 0.03$		

Table 5.3: Example Contingency Tables and Splitting Measures

5.2 Secure ID3 Protocol

This section present a secure ID3 protocol for privacy preserving decision tree learning, using the Chi-square statistic χ^2 and GINI index as splitting measures.

5.2.1 Data Representation

In our privacy preserving decision tree learning solution, we assume the parties have agreed on a fixed set of attributes, and each attribute has a fixed set of attribute values. The input is a secret shared transactional database, which is contributed by all parties by sharing their own transactions using Shamir's Secret Sharing Scheme. The transactional database is represented as a binary matrix, in which a bit represent the presence of an attribute value in a transaction. Table 5.4 shows such a representation for the example in Table 5.1.

Let $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$ be the set of attributes, where $A_i = \{A_{i1}, A_{i2}, ..., A_{im}\}$ for $1 \leqslant i \leqslant c$, assuming there are m attribute values for attribute A_i, and the last attribute A_c is the class

Day	Outlook			Temperature			Humidity		Windy		Play	
	sunny	overcast	rainy	hot	mild	cool	high	normal	true	false	yes	no
1	1	0	0	1	0	0	1	0	0	1	0	1
2	1	0	0	1	0	0	1	0	1	0	0	1
3	0	1	0	1	0	0	1	0	0	1	1	0
4	0	0	1	0	1	0	1	0	0	1	1	0
5	0	0	1	0	0	1	0	1	0	1	1	0
6	0	0	1	0	0	1	0	1	1	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
14	0	0	1	0	1	0	1	0	1	0	0	1

Table 5.4: The Weather Problem

attribute. The secreted shared database is denoted by $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_l]\}$, assuming there are l transactions in total. We use $[T_{ij}]$ to denote the bit in i-th row and j-th column of $[\boldsymbol{T}]$.

Since the attributes $\boldsymbol{A}$ is public, and the structure of transactional database $[\boldsymbol{T}]$ is known to all parties, we can easily transform the database into attribute value vectors $[\boldsymbol{V}]$. An attribute value vector is shared l-bit-vector, which corresponds to the column labeled with the attribute value in $[\boldsymbol{T}]$. For example, the attribute value vector for 'sunny' in Table 5.4 is $\{1, 1, 0, 0, 0, 0, ..., 0\}$.

Let $[\boldsymbol{V}] = \{[V_1], [V_2], ..., [V_c]\}$, where $[V_i] = \{[V_{i1}], [V_{i2}], ..., [V_{im}]\}$ for $1 \leqslant i \leqslant c$. We use $[V_{ij}]$ to denote the attribute value vector for attribute value A_{ij} in A_i. The attribute value vectors $[\boldsymbol{V}]$ is obtained using Protocol 5.2.

Protocol 5.2 Create Attribute Value Vectors: $[\boldsymbol{V}] \leftarrow \mathit{AttrVect}([\boldsymbol{T}], \boldsymbol{A})$

Input: $[\boldsymbol{T}]$, $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$, where $A_i = \{A_{i1}, A_{i2}, ..., A_{im}\}$ for $1 \leqslant i \leqslant c$
Output: $[\boldsymbol{V}] = \{[V_1], [V_2], ..., [V_c]\}$, where $[V_i] = \{[V_{i1}], [V_{i2}], ..., [V_{im}]\}$ for $1 \leqslant i \leqslant c$

```
1: foreach $i \in \{1, ..., c\}$ do
2:     foreach $j \in \{1, ..., m\}$ do
3:         get the column position of attribute value $A_{ij}$, let it be $x$
4:         $[V_{ij}] \leftarrow \{[T_{1x}], [T_{2x}], ..., [T_{lx}]\}$
5:     end for
6:     $[V_i] \leftarrow \{[V_{i1}], [V_{i2}], ..., [V_{im}]\}$
7: end for
8: $[\boldsymbol{V}] = \{[V_1], [V_2], ..., [V_c]\}$
```

The decision tree $\boldsymbol{T'} = \{P_1, P_2, ...\}$ is represented by a set of public decision paths, and a decision path P_i is a list of tree nodes, where a tree node is an attribute value. The last node is a leaf node, which is a class attribute value. For example, the decision path from Figure 5.1 (**If** ('Outlook' = Sunny) **and** ('Humidity' = Normal) **then** 'Play Tennis' = Yes) is represented by $(sunny, normal, yes)$, and the decision tree for the given example would be:
$\{(sunny, normal, yes), (sunny, high, no), (overcast, yes), (rainy, true, no), (rainy, false, yes)\}$

Remark. If a natural representation of transactional database like the example in Table 5.1 is used, a lot of equality tests would be needed to obtain the attribute value vectors $[\boldsymbol{V}]$, as shown in Protocol 5.3. The bit matrix representation of transactional database is much more efficient than the natural representation.

Protocol 5.3 Create Attribute Value Vectors: $[\boldsymbol{V}] \leftarrow CrAttrVect([\boldsymbol{T}], \boldsymbol{A})$

```
Input: $[\boldsymbol{T}]$, $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$, where $A_i = \{A_{i1}, A_{i2}, ..., A_{im}\}$ for $1 \leqslant i \leqslant c$
Output: $[\boldsymbol{V}] = \{[V_1], [V_2], ..., [V_c]\}$, where $[V_i] = \{[V_{i1}], [V_{i2}], ..., [V_{im}]\}$ for $1 \leqslant i \leqslant c$
1: foreach $i \in \{1, ..., c\}$ do
2:     foreach $j \in \{1, ..., m\}$ do
3:         $[V_{ij}] \leftarrow \{ProbEqual(A_{ij}, [T_{1i}]), ProbEqual(A_{ij}, [T_{2i}]), ..., ProbEqual(A_{ij}, [T_{li}])\}$
                                        // or use Protocol $BoundEqual$
4:     end for
5: end for
```

5.2.2 Create Contingency Table

As shown in Section 5.1, the contingency table is the basis for all splitting measures. In the original ID3 algorithm, the table can be easily obtained by traversing the given database. This is not efficient in the case of secure ID3 protocol, as the database becomes secret shared, hence traversing database would requires a lot of secure equality tests. To efficiently compute contingency table, we introduced the concept of path vector.

A path vector is a shared l-bit-vector, which is defined as the Hadamard product of all attribute value vectors in the decision path. Given the path vector $[p_v]$ of a decision path p, Protocol 5.4 is used to create the candidate path vectors for attribute A_a, which is a candidate attribute for the next tree node to be added in p.

Protocol 5.4 Generate Candidate Path Vectors: $[P'_v] \leftarrow PathVect([\boldsymbol{V}], A_a, [p_v])$

```
Input: $[\boldsymbol{V}]$, $A_a = \{A_{a1}, A_{a2}, ..., A_{am}\}$, $[p_v]$
Output: $[P'_v]$, where $[P'_v] = \{[P'_{v1}], [P'_{v2}], ..., [P'_{vm}]\}$
1: foreach $i \in \{1, ..., m\}$ in parallel do
2:     foreach $j \in \{1, ..., l\}$ in parallel do
3:         $[v_j] \leftarrow Mul([p_{vj}], [V_{a,i,j}])$
4:     end for
5:     $[P'_{vi}] \leftarrow \{[v_1], [v_2], ..., [v_l]\}$
6: end for
7: $[P'_v] \leftarrow \{[P'_{v1}], [P'_{v2}], ..., [P'_{vm}]\}$
```

Complexity Analysis. All the computation can be done in parallel, thus the round complexity is one. This protocol requires $m \cdot l$ invocations, where m is the number of attribute values in A_a. Generally, m is a small constant, hence the communication complexity is $\mathcal{O}(l)$, which is linear to the size of transactional database.

To create the contingency table for attribute A_a, we first use Protocol 5.4 to generate candidate path vectors $[P'_v]$, and then the element in contingency table is calculated as the inner product of the candidate path vector and class attribute value vector, as shown in Table 5.5 and Protocol 5.5.

Attribute Value	A_{c1}	A_{c2}	$\cdots$	A_{cn}
A_{a1}	$Inner([P'_{v1}],[V_{c1}])$	$Inner([P'_{v1}],[V_{c2}])$	$\cdots$	$Inner([P'_{v1}],[V_{cn}])$
A_{a2}	$Inner([P'_{v2}],[V_{c1}])$	$Inner([P'_{v2}],[V_{c2}])$	$\cdots$	$Inner([P'_{v2}],[V_{cn}])$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
A_{am}	$Inner([P'_{vm}],[V_{c1}])$	$Inner([P'_{vm}],[V_{c2}])$	$\cdots$	$Inner([P'_{vm}],[V_{cn}])$

Table 5.5: Secure Computing Contingency Table

Protocol 5.5 Create Contingency Table: $[\boldsymbol{X}] \leftarrow CrTab([\boldsymbol{V}], [P'_v], A_c)$

Input: $[\boldsymbol{V}]$, $[P'_v] = \{[P'_{v1}], [P'_{v2}], ..., [P'_{vm}]\}$, $A_c = \{A_{c1}, A_{c2}, ..., A_{cn}\}$
Output: $[\boldsymbol{X}] = \{[X_1], [X_2]...\}$, where $[X_i] = \{[x_{i1}], ..., [x_{in}]\}$

```
1: $a \leftarrow 1$
2: foreach $i \in \{1, ..., m\}$ do
3:     foreach $j \in \{1, ..., n\}$ do
4:         $[x'_{ij}] \leftarrow Inner([P'_{vi}], [V_{cj}])$
5:     end for
6: end for
```

Complexity Analysis. Protocol 5.5 requires $m \cdot n$ invocations.

5.2.3 Calculate χ^2 Statistic and Gini Index

Protocol 5.7 and Protocol 5.8 illustrate how to calculate the χ^2 statistic and Gini Index respectively. They accept a contingency table $[\boldsymbol{X}]$ as input, and output the splitting measure value. The output is represented by a fraction, namely two secret shared values, numerator $[f_1]$ and denominator $[f_2]$, which are calculated based on the following equations:

$$\chi^2 = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{(Nx_{ij} - x_{i:}x_{:j})^2}{Nx_{i:}x_{:j}} = \frac{\sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n} (Nx_{ij} - x_{i:}x_{:j})^2 (\prod\limits_{k=1,k\neq i}^{m} x_{k:} \prod\limits_{k=1,k\neq j}^{n} x_{:k})}{N \prod\limits_{k=1}^{m} x_{k:} \prod\limits_{k=1}^{n} x_{:k}} \tag{5.4}$$

$$GI = \frac{1}{N}(\sum_{i=1}^{m}\sum_{j=1}^{n} \frac{{x_{ij}}^2}{x_{i:}} - \sum_{j=1}^{n} \frac{{x_{:j}}^2}{N}) = \frac{N \sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n} ({x_{ij}}^2 \prod\limits_{k=1,k\neq i}^{m} x_{k:}) - (\sum\limits_{j=1}^{n} {x_{:j}}^2)(\prod\limits_{k=1}^{m} x_{k:})}{N^2 \prod\limits_{k=1}^{m} x_{k:}} \tag{5.5}$$

Note that the row totals $R = \{R_1, R_2, ..., R_m\}$ are all non-zero now, thus $\prod\limits_{k=1,k\neq j}^{n} x_{:k}$ can be computed as $\frac{1}{x_{:j}} \prod\limits_{k=1}^{n} x_{:k}$. This also applies to the case of column totals $C = \{C_1, C_2, ..., C_n\}$.

Since N is the total number of transactions, which is the same for each candidate attribute, and we calculate splitting measures only for comparison, thus the N in the denominator part of χ^2 and the N^2 in the denominator part of GI can be left out.

As explained in Section 5.1.2, the row total $x_{i:}$ and column total $x_{:j}$ could be zero, and it should be excluded from calculation to avoid indeterminate result. To address this issue, Protocol *Zero2One* is used to check the zeroness of the row total $x_{i:}$. If it is zero, the row total

$x_{i:}$ is changed from zero to one. This transformation does not effect the computation result of the above equations, as the protocol is only used for the calculation of product.

Protocol 5.6 Zero to One Test: $[y] \leftarrow \mathit{Zero2One}([x])$

```
Input: $[x] = \{[x_1], ..., [x_n]\}$
Output: $[y] = \{[y_1], ..., [y_n]\}$
 1: foreach $i \in \{1, ..., n\}$ do
 2:     $[y_i] \leftarrow [x_i] + \mathit{ProbEqual}([x_i], 0)$ // or use $\mathit{BoundEqual}$
 3: end for
```

Complexity Analysis. The communication complexity of Protocol 5.6 is $n \cdot (6k - 1)$ invocation when Protocol *ProbEqual* is used, and $n \cdot (2m)$ when Protocol *BoundEqual* is used.

Remark. If the parties can accept to leak the zeroness information about row total $x_{i:}$, then Protocol *EqualPub* can be used, which is more efficient, as the communication complexity reduces to $2n$ invocations.

Protocol 5.7 Calculate χ^2 statistic: $\{[f_1], [f_2]\} \leftarrow \mathit{X2}([\boldsymbol{X}])$

```
Input: $[\boldsymbol{X}] = \{[X_1], [X_2], ..., [X_m]\}$, where $[X_i] = \{[x_{i1}], [x_{i2}], ..., [x_{in}]\}$ for $1 \leqslant i \leqslant m$
Output: $\{[f_1], [f_2]\}$          // fraction: numerator $[f_1]$, denominator $[f_2]$
 1: $[R] \leftarrow \{\sum_{j=1}^{n} [x_{1j}], \sum_{j=1}^{n} [x_{2j}], ..., \sum_{j=1}^{n} [x_{mj}]\}$ // Row totals, $R = \{R_1, R_2, ..., R_m\}$
 2: $[C] \leftarrow \{\sum_{i=1}^{m} [x_{i1}], \sum_{i=1}^{m} [x_{i2}], ..., \sum_{i=1}^{m} [x_{in}]\}$ // Column totals, $C = \{C_1, C_2, ..., C_n\}$
 3: $[N] \leftarrow \sum [R]$, $[f_1] \leftarrow 0$
 4: $[R'] \leftarrow \mathit{Zero2One}([R])$        // $R' = \{R'_1, R'_2, ..., R'_m\}$
 5: $[J_r] \leftarrow \mathit{MulLogk}([R'])$
 6: $[C'] \leftarrow \mathit{Zero2One}([C])$        // $C' = \{C'_1, C'_2, ..., C'_m\}$
 7: $[J_c] \leftarrow \mathit{MulLogk}([C'])$
 8: $[J] \leftarrow \mathit{Mul}([J_r], [J_c])$
 9: for $i \in \{1, ..., m\}$, $[R_i^{-1}] \leftarrow \mathit{Inv}([R'_i])$
10: for $j \in \{1, ..., n\}$, $[C_j^{-1}] \leftarrow \mathit{Inv}([C'_j])$
11: foreach $i \in \{1, ..., m\}$ do
12:     $[A_i] \leftarrow \mathit{Mul}([J], [R_i^{-1}])$
13:     foreach $j \in \{1, ..., n\}$ do
14:         $[t] \leftarrow \mathit{Inner}(\{N, -[R_i]\}, \{[x_{ij}], [C_j]\})$
15:         $[B_j] \leftarrow \mathit{Mul}([t], [t])$
16:     end for
17:     $[D_i] \leftarrow \mathit{Inner}(\{[C_1^{-1}], ..., [C_n^{-1}]\}, \{[B_1], ..., [B_n]\})$
18: end for
19: $[A] \leftarrow \{[A_1], ..., [A_m]\}$, $[D] \leftarrow \{[D_1], ..., [D_m]\}$
20: $[f_1] \leftarrow \mathit{Inner}([A], [D])$
21: $[f_2] \leftarrow [J]$
```

Protocol 5.8 Calculate Gini Index: $\{[f_1], [f_2]\} \leftarrow GI([\boldsymbol{X}])$

```
Input: $[\boldsymbol{X}] = \{[X_1], [X_2], ..., [X_m]\}$, where $[X_i] = \{[x_{i1}], [x_{i2}], ..., [x_{in}]\}$ for $1 \leqslant i \leqslant m$
Output: $\{[f_1], [f_2]\}$                // fraction: numerator $[f_1]$, denominator $[f_2]$
 1: $[R] \leftarrow \{\sum_{j=1}^{n}[x_{1j}], \sum_{j=1}^{n}[x_{2j}], ..., \sum_{j=1}^{n}[x_{mj}]\}$ // Row totals, $R = \{R_1, R_2, ..., R_m\}$
 2: $[C] \leftarrow \{\sum_{i=1}^{m}[x_{i1}], \sum_{i=1}^{m}[x_{i2}], ..., \sum_{i=1}^{m}[x_{in}]\}$ // Column totals, $C = \{C_1, C_2, ..., C_n\}$
 3: $[N] \leftarrow \sum[R]$, $[f_1] \leftarrow 0$
 4: $[R'] \leftarrow Zero2One([R])$        // $R' = \{R'_1, R'_2, ..., R'_m\}$
 5: $[J_r] \leftarrow MulLogk([R'])$
 6: for $i \in \{1, ..., m\}$, $[R_i^{-1}] \leftarrow Inv([R'_i])$
 7: foreach $i \in \{1, ..., m\}$ do
 8:     $[A_i] \leftarrow Mul([J_r], [R_i^{-1}])$
 9:     $[B_i] \leftarrow Inner([X_i], [X_i])$
10: end for
11: $[A] \leftarrow \{[A_1], ..., [A_m]\}$, $[B] \leftarrow \{[B_1], ..., [B_m]\}$
12: $[f_1] \leftarrow Inner(\{[N], -[J_r]\}, \{Inner([A], [B]), Inner([C], [C])\})$
13: $[f_2] \leftarrow [J_r]$
```

Complexity Analysis. Protocol $X2$ requires $2mn + 4m + 2n - 1$ invocations, while Protocol GI only needs $4m + 2$ invocations, which is more efficient, compared to Protocol $X2$.

After the splitting measure value for each attribute is obtained, we need a protocol to securely compare these values. Protocol 5.9 compares a list of fraction numbers $[\boldsymbol{x}]$, and output the index position of the largest fraction. The basic idea of this protocol is that if $\frac{a}{b} \geqslant \frac{c}{d}$ then $a \cdot d \geqslant b \cdot c$.

Protocol 5.9 Compare Fractions: $I \leftarrow ArgMax([\boldsymbol{x}])$

```
Input: $[\boldsymbol{x}] = \{([f_{11}], [f_{12}]), ..., ([f_{n1}], [f_{n2}])\}$ // a list of $n$ fraction
Output: $i$                // the index position of the largest fraction
 1: $i \leftarrow 1$, $([f_1], [f_2]) \leftarrow ([f_{11}], [f_{12}])$
 2: foreach $j \in \{2, ..., n\}$ do
 3:     $[t] \leftarrow GEqualThan(Inner(([f_1], [f_2]), ([f_{j2}], -[f_{j1}])), 0)$
 4:     $[i] \leftarrow [i] + Mul([t], j - [i])$
 5:     $[f_1] \leftarrow [f_1] + Mul([t], [f_{j1}] - [f_1])$
 6:     $[f_2] \leftarrow [f_2] + Mul([t], [f_{j2}] - [f_2])$
 7: end for
 8: $i \leftarrow OpenShare([i])$
```

Complexity Analysis. The complexity is mainly determined by Protocol $GEqualThan$. Since n is the number of candidate attributes, which is usually a small constant, thus the communication complexity is $\mathcal{O}(m)$, where m is the bit length of inputs.

5.2.4 Complete Decision Path

Protocol 5.10 is used to complete a decision path, i.e., to add a leaf node to current path. It returns a finished path with the leaf node be the class attribute value that is associated with the most transactions.

Protocol 5.10 Complete Decision Path: $c \leftarrow PathEnd([\boldsymbol{V}], \boldsymbol{A}, p, [p_v])$

```
Input: $[\boldsymbol{V}]$, $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$, $p$, $[p_v]$, where $A_c = \{A_{c1}, ..., A_{cn}\}$
Output: $p'$
1: foreach $j \in \{1, ..., n\}$ do
2:     $[x_j] \leftarrow Inner([p_v], [V_{cj}])\}$
3: end for
4: $i \leftarrow ArgMax([x_1], ..., [x_n])$
5: $p' \leftarrow p \cup A_{ci}$
```

There are two situations in which we want to stop the growing of current path, and finish it early. One situation is that there is no attributes left to be selected to split the tree. And the second situation is that the number of transactions under current path is too small, and it is better to stop the growing early to avoid long decision paths. Protocol 5.11 is used to check the second situation, it returns a public value s to indicate whether current path p reach stopping threshold. The *threshold* is the minimum number of transactions that each decision path should have.

Protocol 5.11 Check Stopping Condition: $s \leftarrow CkStop(p, [p_v], threshold)$

```
Input: $[\boldsymbol{V}]$, $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$, $p$, $[p_v]$
Output: $s$
1: $[s] \leftarrow GEqualThan(\sum[p_v], threshold)$
2: $s \leftarrow OpenShare([s])$
```

Protocol 5.12 is to check whether all transactions under the given path p have the same class value. If so, the decision path should become complete, the protocol returns a public value $c = 1$, and a new path ending with a leaf node. In this case, let the last non-leaf node in p be A_{ij}, the row for A_{ij} in the contingency table of attribute A_i would have only one non-zero element, i.e., the row total should be equal to one of the element in the row. Note that Protocol *CkEnd* only opens the equality test result, it will not leak information about the transactions under current path p.

Protocol 5.12 Check Decision Path: $c, p' \leftarrow CkEnd([\boldsymbol{V}], \boldsymbol{A}, p, [p_v])$

```
Input: $[\boldsymbol{V}]$, $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$, $p$, $[p_v]$, where $A_c = \{A_{c1}, ..., A_{cn}\}$
Output: $c, p'$
1: $c \leftarrow 0$, $p' \leftarrow p$
2: foreach $j \in \{1, ..., n\}$ do
3:     $[x_j] \leftarrow Inner([p_v], [V_{cj}])\}$
4: end for
5: $[s] \leftarrow \sum_{j=1}^{n}[x_j]$
6: foreach $j \in \{1, ..., n\}$ do
7:     if $OpenShare(ProbEqual([x_j], [s]))$ then
8:         $c \leftarrow 1$, $p' \leftarrow p \cup A_{cj}$
9:     end if
10: end for
```

5.2.5 Double Field Setting

Generally, all sub-protocols in a secure multiparty computation protocol use the same finite field $\mathbb{Z}_p$, and the field is large enough to hold any possible integers occurred during the computation. Logically, the larger the field is, the more computation time is needed. In the secure ID3 protocol, the large integers occurs in Protocol GI and $X2$, where the Gini Index and χ^2 statistic are calculated as fractions. The optimistic estimate of the possible largest integer should be less than $(N^2)(N/c)^{2c}$ for Protocol GI and $(N^2)(N/c)^{4c}$ for Protocol $X2$, where N is the total number of transactions, and c is the number of class attribute values.

While such a large prime p is only needed for the calculation of splitting measures, for other sub-protocols, a much smaller prime p' is enough. If the whole protocol use the same finite field $\mathbb{Z}_p$, it would greatly increase the computation work of other sub-protocols. To address this issue, we proposed to use a double field setting, i.e., a large field for Protocol GI and $X2$, and a small filed for other tasks.

To convert share between different fields, Protocol 5.13 is used. Suppose $[s]$ is a secret in field $\mathbb{Z}_p$, we want to obtain a secret $[s']$ in another field $\mathbb{Z}_{p'}$, such that $s = s'$. The idea is to generate two random secrets $[r]$ and $[r']$, such that $r = r'$ and $r \in \mathbb{Z}_p$, $r' \in \mathbb{Z}_{p'}$. Then the parties open $r + s$, and compute $[s'] = r + s - [r']$.

Protocol 5.13 Convert share between different fields : $[s'] \leftarrow ConvertShare([s], k)$

```
Input: $[s], k$, where $s \in \mathbb{Z}_p$, $k$ is a security parameter
Output: $[s']$, where $s' \in \mathbb{Z}_{p'}$
1: generate $[r], [r']$ from $\mathbb{Z}_p$ and $\mathbb{Z}_{p'}$ respectively, such that $r = r'$
   $[r] \leftarrow PRand2m(\mathbb{Z}_p, k)$, $[r'] \leftarrow PRand2m(\mathbb{Z}_{p'}, k)$
2: $t \leftarrow OpenShare([r] + [s])$
3: $[s'] \leftarrow t - [r']$
```

5.2.6 Overview of Secure ID3 Protocol

The overall approach for secure ID3 protocol is to generate a set of unfinished decision paths after a best attribute is chosen, and then determine the next node for each unfinished decision path, until it reaches a leaf node, i.e., the current decision path becomes complete. The secure ID3 protocol is shown in Protocol 5.14. It first transform the database into attribute value vectors using $AttrVect$, and initialize the set of unfinished paths $\boldsymbol{U}$ and the set of path vectors $[\boldsymbol{P_v}]$. Then it deals with each unfinished path in $\boldsymbol{U}$, trying to complete the path or extend it by adding non-lead nodes. The finished paths are added to the decision tree $\boldsymbol{T'}$, and unfinished paths are added back to $\boldsymbol{U}$ after a splitting.

Protocol 5.14 Secure ID3 Protocol: $\boldsymbol{T'} \leftarrow SID3([\boldsymbol{T}], \boldsymbol{A})$

Input: $[\boldsymbol{T}] = \{[T_1], [T_2], ..., [T_l]\}$, $\boldsymbol{A} = \{A_1, A_2, ..., A_c\}$, *threshold*
Output: $\boldsymbol{T'}$

```
 1: $[\boldsymbol{V}] \leftarrow AttrVect([\boldsymbol{T}], \boldsymbol{A})$
 2: Initialize $\boldsymbol{T'} = \emptyset$, $\boldsymbol{U} = \{p\}$, where $p = \emptyset$
 3: Initialize $[\boldsymbol{P_v}] = \{[p_v]\}$, where $[p_v] = \{[1], [1], ..., [1]\}$, an $l$-vector with all $[1]$
 4: while $\boldsymbol{U}$ is not empty do
 5:   get a path $p$ from $\boldsymbol{U}$, delete it from $\boldsymbol{U}$
 6:   get its path vector $[p_v]$ from $[\boldsymbol{P_v}]$, delete it form $[\boldsymbol{P_v}]$
 7:   $\boldsymbol{B} \leftarrow \boldsymbol{A} \backslash \{p, A_c\}$            // candidate attribute set
 8:   if $CkStop(p, [p_v], threshold)$ or $\boldsymbol{B}$ is empty then
 9:     $p' \leftarrow PathEnd([\boldsymbol{V}], \boldsymbol{A}, p, [p_v])$
10:     $\boldsymbol{T'} \leftarrow \boldsymbol{T'} \cup p'$               // add a complete path to the tree
11:   else
12:     $c, p' \leftarrow CkEnd([\boldsymbol{V}], \boldsymbol{A}, p, [p_v])$
13:     if $c$ then
14:       $\boldsymbol{T'} \leftarrow \boldsymbol{T'} \cup p'$             // add a complete path to the tree
15:     else
16:       let the number of attributes in $\boldsymbol{B}$ be $z$
17:       foreach $j \in \{1, ..., z\}$ do
18:         let the $j$-th attribute in $\boldsymbol{B}$ be $A_i$
19:         $[P'_{vj}] \leftarrow PathVect([\boldsymbol{V}], A_i, [p_v])$ // generate candidate path vectors
20:         $[\boldsymbol{X}] \leftarrow CrTab([\boldsymbol{V}], [P'_{vj}], A_c)$ // create contingency table
                            // Protocol ConvertShare is used here for double field setting
21:         $[M_j] \leftarrow GI([\boldsymbol{X}])$     // or use Protocol $X2$
22:       end for
23:       $i \leftarrow ArgMax(\{[M_1], ..., [M_z]\})$
24:       Let the $i$-th attribute in $B$ is $A_y$
25:       $[\boldsymbol{P_v}] \leftarrow [\boldsymbol{P_v}] \cup [P'_{vi}]$
26:       foreach $a \in A_y$ do
27:         $p' \leftarrow p \cup a$            // extend $p$ with new attribute value
28:         $\boldsymbol{U} \leftarrow \boldsymbol{U} \cup p'$           // add the new path $p'$ to path set $\boldsymbol{U}$
29:       end for
30:     end if
31:   end if
32: end while
```

For double field setting, Protocol ConvertShare is is used after the creation of contingency table. The elements in contingency table are converted from a small field to a big field.

Security Analysis. Protocol 5.14 is secure in the sense that the input transactions are secret, and no information about the transaction content is leaked. An adversary can see the process of building a decision tree, but this does not harm the privacy, as anyone has the output can deduce that information from the output.

Complexity Analysis. The complexity of secure ID3 protocol is mainly determined by two parts, the generation of candidate path vectors (Protocol *PathVect*) and selecting attributes to be added as tree nodes (Protocol *ArgMax*). Protocol *PathVect* requires $\mathcal{O}(l)$ invocations, and Protocol *ArgMax* requires $\mathcal{O}(m)$, where l is the size of transactional database, and m is the bit length of inputs.

5.3 Performance Result

This section presents the experiment results of secure ID3 protocol. We implemented the protocols in VIFF, and conducted a benchmarking experiment with the following data sets from UCI Machine Learning Repository [FA10]. We measure the overall execution time of our programs under the experiment settings introduced in Section 2.3. The threshold for early stopping is set to 5% of the size of input dataset.

- **SPECT.** Data on cardiac Single Proton Emission Computed Tomography (SPECT) images. It has 267 SPECT image sets (patients), classified into two categories: normal and abnormal, based on 23 binary attributes.
- **Scale.** Balance Scale Weight & Distance Database. Each transaction is classified as having the balance scale tip to the right, tip to the left, or be balanced. The non-class attributes are the left weight, the left distance, the right weight, and the right distance. Each attribute has five categorical values. This data set has 625 transactions in total.
- **Car.** Car Evaluation Database has the following six attributes: buying price (4 attribute values), price of the maintenance (4 attribute values), number of doors (4 attribute values), capacity in terms of persons to carry (3 attribute values), the size of luggage boot (3 attribute values), and estimated safety of the car (3 attribute values). This data set has 1728 transactions in total.
- **KRKPA7.** Chess (King+Rook versus King+Pawn) Data Set, which is used to describe a Chess End-Game, where the pawn is on a7, and it is the King+Rook's side (white) to move. It has 36 attributes and 3196 transactions in total, each transaction is classified into two classes: White can win and White can not win.

To compare the efficiency of two different field settings (Single Field, Double Field), and two equality test protocols (*BoundEqual*, *ProbEqual*), we tested the program for four possible combinations, as shown in Table 5.6. For the single field setting, the bit length is set to $\log_2(N^2)(N/c)^{4c}$, which is enough to cover the largest integer. A much smaller bit length 10 is used in the double field setting for the small field, and the big field use the same bit length as the single field setting.

Data	Size	Bit Len.	Single Field		Double Field	
			ProbEqual	*BoundEqual*	*ProbEqual*	*BoundEqual*
SPECT	267	45 bit	235s	198s	235s	173s
Scale	625	65 bit	43s	39s	41s	36s
Car	1728	92 bit	109s	108s	104s	95s

Table 5.6: Performance Result of Different Settings for Secure ID3 Protocol

As expected, using double field setting instead of single field setting achieves better performance. And Protocol *BoundEqual* performs better than Protocol *ProbEqual*, this is because the bit length 10 is much smaller than the security parameter in VIFF (see Section 3.5.3). It is clear that using Double Field + *BoundEqual* Protocol is the best combination for secure ID3 protocol.

The results in Table 5.6 are obtained by using Gini Index as splitting measure. To compare the performance of Gini Index and χ^2 statistic in secure ID3 protocol, we measures the execution time of our program under the Double Field + *BoundEqual* Protocol setting, as illustrated in Table 5.7.

Data	Size	Bit Len.	Gini Index	Bit Len.	χ^2 Statistic
SPECT	267	45 bit	173s	73 bit	205s
Scale	625	65 bit	36s	112 bit	43s
Car	1728	92 bit	95s	162 bit	119s
KRKPA7	3196	66 bit	1079s	110 bit	1112s

Table 5.7: Performance Result of Gini Index and χ^2 Statistic in Secure ID3 Protocol

Though Protocol *GI* has less complexity than Protocol *X2*, their performance in secure ID3 protocol is more or less the same, Protocol *GI* only has slight advantages. This is because the execution time is mainly determined by the generation of candidate path vectors (Protocol *PathVect*) and selecting attributes to be added as tree nodes (Protocol *ArgMax*). Table 5.8 shows the execution time of Protocol *PathVect* and Protocol *ArgMax* in secure ID3 protocol.

Data	Size	Bit Len.	*PathVect*	*ArgMax*	*GI*	*SID3*
SPECT	267	45 bit	106.2s	36.4s	10.5s	173s
Scale	625	65 bit	22.1s	5.5s	1.1s	36s
Car	1728	92 bit	73.4s	9.4s	0.8s	95s
KRKPA7	3196	66 bit	990.1s	43.8s	5.5s	1079s

Table 5.8: Performance of Sub-protocols in Secure ID3 Protocol

Chapter 6

Conclusions

In this thesis, we considered using secure multiparty computation for privacy preserving data mining. More specifically, the problem of secure frequent itemset mining and secure decision tree learning are addressed.

With regard to secure multiparty computation, our protocols are based on threshold secret sharing schemes. The protocols are implemented in the Virtual Ideal Functionality Framework and tested with data set from UCI Machine Learning Repository.

We started with reviewing existed basic protocols that are used as building blocks for complex protocols. For efficient computation, we proposed a new equality test protocol for bounded secrets, and protocols for symmetric AND/OR operation based on our equality test protocol.

In Chapter 4, we show how to build protocols for secure frequent itemset mining, based on the Apriori algorithm. Two secure frequent itemset mining protocols are given, one outputs public frequent itemsets and the other one generate secret outputs. There is a natural tradeoff between privacy and performance, the price paid for obtaining a fully secure frequent itemset mining solution is significantly increased execution time.

Chapter 5 presents a secure ID3 protocol for privacy preserving decision tree learning, based on the ID3 algorithm. To avoid the secure computation of logarithm function, we considered using χ^2 statistic and Gini Index instead of information measure used in original ID3 algorithm. The experiment result shows χ^2 statistic is more efficient than Gini Index. In addition, we considered using a double setting for secure ID3 protocol to improve performance.

6.1 Further Research

Based on our work, the following topics may be interesting for further research.

- **Fully secure ID3 protocol.** The secure ID3 protocol in Chapter 5 outputs public decision tree. In some cases, the parties may want to keep the result secret as well, hence it is interesting to have a fully secure ID3 protocol.

- **Efficient implementation.** Our protocols are implemented using VIFF, which is based on Python programming language. As an interpreted language, Python is slower than compiled language such as C/C++. In addition, VIFF uses Twisted library for network communication, which may introduce extra cost. An ad-hoc application can be designed for improving efficiency of our protocols.

Bibliography

[AIS93] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 207–216, New York, NY, USA, 1993. ACM.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[AY08] C. C. Aggarwal and P. S. Yu. A general survey of privacy-preserving data mining models and algorithms. In C. C. Aggarwal and P. S. Yu, editors, *Privacy-Preserving Data Mining: Models and Algorithms*, pages 11–52. Springer, 2008.

[BCD$^+$09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer-Verlag.

[Bea92] Donald Beaver. Foundations of secure interactive computing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 377–391, London, UK, UK, 1992. Springer-Verlag.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. Stone. *Classification and regression trees.* Belmont, CA: Wadsworth International Group, 1st edition, 1984.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th Symposium on Theory of Computing (STOC '88)*, pages 1–10, New York, 1988. A.C.M.

[BIB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 201–209, New York, NY, USA, 1989. ACM.

[Bla79] G.R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference 1979*, volume 48 of *AFIPS Conference Proceedings*, pages 313–317, 1979.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. In *Proceedings of the Sixteenth International Conference on Financial Cryptography and Data Security*, FC '12, 2012.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS '01)*. IEEE Computer Society, 2001.

[CAS12] Cassandra project. http://www.cassandra-project.eu/, 2012.

[CDI05] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proceedings of the Second international conference on Theory of Cryptography*, TCC'05, pages 342–362, Berlin, Heidelberg, 2005. Springer-Verlag.

[CDN01] R. Cramer, I. Damgård, and J.B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology—EUROCRYPT '01*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–300, Berlin, 2001. Springer-Verlag. Full version `eprint.iacr.org/2000/055`, October 27, 2000.

[CGMA85] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 383–395, Washington, DC, USA, 1985. IEEE Computer Society.

[DFK$^+$06] I. Damgård, M. Fitzi, E. Kiltz, J.B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. 3rd Theory of Cryptography Conference (TCC 2006)*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304, Berlin, 2006. Springer-Verlag.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[DK10] Ivan Damgård and Marcel Keller. Secure multiparty aes. In Radu Sion, editor, *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 367–374. Springer Berlin Heidelberg, 2010.

[DN03] I. Damgård and J.B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology—CRYPTO '03*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264, Berlin, 2003. Springer-Verlag.

[EFG$^+$09] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Proceedings of the 9th International Symposium on Privacy Enhancing Technologies*, PETS '09, pages 235–253, Berlin, Heidelberg, 2009. Springer-Verlag.

[FA10] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[FH94] Matthew K. Franklin and Stuart Haber. Joint encryption and message-efficient secure computation. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 266–277, London, UK, UK, 1994. Springer-Verlag.

[Gei10] Martin Geisler. *Cryptographic Protocols: Theory and Implementation.* PhD thesis, Aarhus University, Denmark, February 2010.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game - or - a completeness theorem for protocols with honest majority. In *Proc. 19th Symposium on Theory of Computing (STOC '87)*, pages 218–229, New York, 1987. A.C.M.

[Hoo12] Sebastiaan de Hoogh. *Design of Large Scale Applications of Secure Multiparty Computation: Secure Linear Programming.* PhD thesis, Eindhoven University of Technology, the Netherlands, July 2012.

[IKS+07] Ali İnan, Selim V. Kaya, Yücel Saygın, Erkay Savaş, Ayça A. Hintoğlu, and Albert Levi. Privacy preserving clustering on horizontally partitioned data. *Data Knowl. Eng.*, 63(3):646–666, December 2007.

[Jag10] Roman Jagomägis. Secrec: a privacy-aware programming language with applications in data mining. Master's thesis, UNIVERSITY OF TARTU, Estonia, July 2010.

[JW05] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 593–599, New York, NY, USA, 2005. ACM.

[KC04] Murat Kantarcıoğlu and Chris Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1026–1037, September 2004.

[Kel10] Marcel Keller. viff boost extension. http://lists.viff.dk/pipermail/viff-devel-viff.dk/2010-August/000847.html, 2010.

[Kil88] J. Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 20–31, New York, NY, USA, 1988. ACM.

[Lov12] Radu Lovin. Data mining 101: Part 3 - mining frequent patterns (maximal and closed frequent itemsets). http://www.dataminingarticles.com/closed-maximal-itemsets.html, 2012.

[LP00] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '00, pages 36–54, London, UK, UK, 2000. Springer-Verlag.

[LP09] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *The Journal of Privacy and Confidentiality*, 1(1):59–98, 2009.

[Mau09] A. Mauland. Realizing distributed rsa using secure multiparty computations. Master's thesis, Norwegian University of Science and Technology, Norwegian, July 2009.

[Min87] J. Mingers. Expert systems - rule induction with statistical data. *Journal of the Operational Research Society*, 38(1):39–47, January 1987.

[Min89] J. Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4):319–342, March 1989.

[MR92] Silvio Micali and Phillip Rogaway. Secure computation (abstract). In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 392–404, London, UK, UK, 1992. Springer-Verlag.

[NO07] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Proceedings of the 10th international conference on Practice and theory in public-key cryptography*, PKC'07, pages 343–360, Berlin, Heidelberg, 2007. Springer-Verlag.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology—EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Berlin, 1999. Springer-Verlag.

[Per92] René Peralta. On the distribution of quadratic residues and nonresidues modulo a prime number. *Mathematics of Computation*, 58(197):pp. 433–440, January 1992.

[Qui86] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[RB89] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proc. 21st Symposium on Theory of Computing (STOC '89)*, pages 73–85, New York, 1989. A.C.M.

[Sch11] Berry Schoenmakers. Multiparty computation. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 812–815. Springer, 2nd edition, 2011.

[Sch12] Berry Schoenmakers. Cryptographic protocols. Lecture Notes, 2012.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[SM08] Saeed Samet and Ali Miri. Privacy preserving id3 using gini index over horizontally partitioned data. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '08, pages 645–651, Washington, DC, USA, 2008. IEEE Computer Society.

[Tof07] Tomas Toft. *Primitives and Applications for Multi-party Computation.* PhD thesis, Aarhus University, Denmark, March 2007.

[VCKP08] Jaideep Vaidya, Chris Clifton, Murat Kantarcıoğlu, and A. Scott Patterson. Privacy-preserving decision trees over vertically partitioned data. *ACM Trans. Knowl. Discov. Data*, 2(3):14:1–14:27, October 2008.

[VKC08] Jaideep Vaidya, Murat Kantarcıoğlu, and Chris Clifton. Privacy-preserving naïve bayes classification. *The VLDB Journal*, 17(4):879–898, July 2008.

[WFH11] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* Elsevier, 3rd edition, 2011.

[Yao82] A. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164. IEEE Computer Society, 1982.