SecurePL : A Compiler and Toolbox for Practical and Easy Secure Multiparty Computation

by Ismail Fatih YILDIRIM

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

Sabanci University

Spring, 2008

SecurePL: A Compiler and Toolbox for
Practical and Easy Secure Multiparty Computation

Approved by:

Assoc.Prof.Dr. Erkay Savaş (Dissertation Supervisor)	.....
Assist.Prof.Dr. Thomas Brochmann Pedersen (Dissertation Co-Supervisor)	.....
Prof.Dr. Aytül Erçil	.....
Assist.Prof.Dr. Selim Balcısoy	.....
Assist.Prof.Dr. Yücel Saygın	.....

Date of Approval: .....

SecurePL: A Compiler and Toolbox for Practical and Easy Secure Multiparty Computation

Ismail Fatih YILDIRIM

CS, Master's Thesis, 2008

Thesis Supervisor: Erkay Savas

Abstract

Secure multiparty computation is basically about techniques that allow multiple parties to jointly carry out computations that are based on data from each of the players while the data held by each player remains private to that player. Since the beginning of the notion of secure multiparty computation, many algorithms and methods were introduced on how to achieve this goal. This thesis first introduces different methods to do secure multiparty computation and later focusing on Secret sharing based multiparty computation it explains how efficient and secure multiparty operations can be done. Also while introducing secret sharing based secure multiparty computation we introduce a novel technique which allows to do secure multiparty computation using the Asmuth Bloom secret sharing scheme, which is not possible in the original scheme. The aim of this thesis is the design and implementation of a programming language and libraries for secure multiparty computation, SecurePL. We show that our tool's ease of use and security allows even a person who has absolutely no knowledge about security or cryptography to write applications that can do secure multiparty computation.

SecurePL: A Compiler and Toolbox for Practical and Easy Secure Multiparty Computation

Ismail Fatih YILDIRIM

CS, Master Tezi, 2008

Thesis Supervisor: Erkay Savas

Özet

Güvenli çok partili hesaplama basitçe, birden fazla partinin her partiden gelen bilgileri kullanarak ortaklaşa hesap yapmasına izin veren metodlar ile ilgilidir. Öyle ki bu hesaplamalar sonucunda her partinin bilgisi kendisinde saklı kalmaktadır. Güvenli çok partili hesaplama kavramının ortaya atıldığı günden beri, bu amaca ulaşmak için bir çok algoritma ve metod geliştirildi. Bu tez öncelikle güvenli çok partili hesaplamaya izin veren farklı metodları tanıtıp, daha sonra sır paylasımı bazlı güvenli çok partili hesaplama yontemleri uzerine odaklaşıp, bu hesaplamaların nasıl verimli yapılabileceğini açıklamaktadır. Ayrıca sır paylaşımı bazlı güvenli çok partili hesaplama konusu anlatılırken, Asmuth Bloom sır paylaşım yonteminin orjinal şeklinde mümkün olmayan guvenli çok partili hesaplama'ya izin veren yeni bir teknik tanıtacağiz. Bu tezin amacı güvenli cok partili hesaplama yapmak icin kullanılabilecek bir programlama dili ve kütüphane tasarlayıp uygulamaktır. Bizim aracımızın kullanım kolaylığı ve güvenliği sayesinde, güvenlik veya kriptoloji hakkında hiçbir bilgisi olmayan bir insanin bile güvenli çok partili hesaplama yapabileceğini gostereceğiz.

Acknowledgements

I wish to express my gratitude to,

- Thomas B. Pedersen and Erkay Savaş for their patience and support
- Thesis comittee for their participation
- K. Tuncay Tekle for his help while implementing SecurePL compiler
- last, but not the least, to my family, for being there when I needed them to be.

Contents

1 Introduction **1**

1.1 Secure Multiparty Computation 1

1.1.1 Yao's multiparty Computation Protocol . . 2

1.1.2 Secret Sharing Based multiparty Computation Protocol 5

1.1.3 Homomorphic Encryption 6

1.2 Compiler Design 7

1.2.1 Lexical Analyzer 8

1.2.2 Parser 9

1.2.3 Parse Tree and Abstract Syntax Tree . . . 9

1.2.4 Syntax Directed Translation 10

1.3 Previous Secure Multiparty Computation Projects and Compiler 10

1.3.1 VIFF, the Virtual Ideal Functionality Framework 10

1.3.2 Secure Multiparty Computation Language . 11

2 Secret Sharing **12**

2.1 Additive Secret Sharing 12

2.2 Shamir Secret Sharing 13
2.2.1 Decomposition and Recovery in Shamir Secret Sharing Scheme 15
2.2.2 Secure Multiparty Computation in Shamir Secret Sharing Scheme 16
2.3 Asmuth Bloom Secret Sharing 19
2.3.1 Decomposition and Recovery in Asmuth Bloom Secret Sharing Scheme 19
2.3.2 Secure Multiparty Computation in Asmuth Bloom Secret Sharing Scheme 21
2.4 Replicated Secret Sharing Scheme 24
2.5 Pseudo Random Secret Sharing 27
2.5.1 Conversion from Replicated Shares to Shamir Shares 28
2.5.2 Pseudo Random Secret Sharing 28
2.6 Bounds on non-interactive multiplication 29
2.6.1 Shamir Secret Sharing Scheme 29
2.6.2 Asmuth Bloom Secret Sharing Scheme . . . 30

3 Comparison **33**

3.1 Toft Comparison 34

3.1.1 Comparison 34
3.1.2 Initial Problem Transformation 34
3.1.3 Comparing $[r]$ and δ 36

4 SecurePL **37**

4.1 Introduction 37
4.2 Base Protocols, Libraries and Tool kits 39
4.2.1 Flex 39
4.2.2 Bison 40
4.2.3 Message Passing Interface (MPI) 40
4.2.4 NTL 41
4.3 SecurePL Compiler 41
4.3.1 Input Language 42
4.4 Implemented SecurePL Libraries and Functions . . 43
4.4.1 Configuration 43
4.4.2 sint 44
4.4.3 ShamirSharer 45
4.4.4 ShamirLib 46
4.5 User Implemented Libraries 48

5 Conclusion **50**

List of Figures

1 Multiplication of Secret Shared Values Using Shamir Secret Sharing Scheme 19

2 SecurePL Architecture 38

3 Configuration Class 44

4 sint Class 45

5 ShamirSharer Class 46

6 Sample SecurePL Code 51

1 Introduction

In this section we introduce the concept of secure multiparty computation. After the introduction to secure multiparty computation we will focus on different methods to do secure multiparty computation. After secure multiparty computation, the concepts of compiler design will be introduced to give some insight about the toolbox that will later be introduced in chapter 4

1.1 Secure Multiparty Computation

Let f be a function that takes n input elements and after computation gives n output elements . Secure multiparty computation is focused on methods such that, participating parties $P_1, P_2, ..., P_n$ where each party P_i knows the input x_i can compute the function

$$f(x_1, x_2, x_3, ..., x_n) = (y_1, y_2, y_3, ..., y_n)$$

together such that each party P_i learns only y_i of the result and has absolutely no information regarding the other components of the result. The public function f can be anything i.e. arithmetic operations, logical operations, comparison etc. For calculating any arbitrary function there are different methods that could be used like circuit evaluation or secret sharing protocols. in this section different approaches to secure multiparty computation will be introduced.

1.1.1 Yao's multiparty Computation Protocol

The protocol explained in this section is based on Lindell and Pinkas' paper: A proof of Yao's protocol for secure two party computation. [10]

Garbled Truth Tables and Garbled Circuits

Let f be a polynomial time function. Then the first step for Yao's protocol is to view the function f as a boolean circuit C. For evaluating a boolean circuit C with inputs x and y from two players a gate by gate evaluation needs to be done from the input wires to the output wires. Once the incoming wires to a gate g have obtained the values $\alpha\beta \in \{0,1\}$, it is possible to give the outgoing wires of the gate the value $g(\alpha,\beta)$. finally the output of circuit would be given based on the outputs on the output wires of the circuit. So computing in a circuit is allocating the appropriate zero-one values to the wires of the circuit.

A high level definition of Yao's protocol is a construction that is actually a compiler that takes any polynomial time function f, or the circuit C that is actually calculating the function f , and constructs a protocol that calculates the function f securely in the presence of semi-honest adversaries. The basic idea behind Yao's protocol is to build a circuit that only input and output wires are visible to the players. To achieve this two random values are specified to each wire so that one value represents 0 and the other one represents 1. For instance let ω be the label of a wire and for that wire two values k_ω^1 and k_ω^0 are chosen where the first represents the bit 1 while the second represents the bit 0. Since the values are random, even if one players sees the value k_ω^ρ the player cannot find out if ρ is 0 or 1. But also since the

values on a wire are just random values, in a setup where a gate g has two input wires ω_1 and ω_2 and one output wire ω_3, it is not possible to evaluate the gate with inputs $k^{\rho}_{\omega_1}$ and $k^{\sigma}_{\omega_2}$ because there is no information in the input about ρ and σ values. To overcome this problem we need to use "garbled truth tables" which take random values as inputs and give random values as output. But while receiving one output on the output wire the player must have no information about the other output that could have been produced by the other wire. To do this the four different values that can be an input for a gate $k^1_{\omega_1}, k^0_{\omega_1}, k^1_{\omega_2}, k^0_{\omega_2}$ have to be used as encryption keys. After a chain of decryptions the player would get either $k^0_{\omega_3}$ or $k^1_{\omega_3}$. Based on this setup an example of a truth table of an OR gate would be like:

Input wire ω_1	Input wire ω_2	Output wire ω_3	Garbled Encryption Table
k^0_1	k^0_2	k^0_3	$E_{k^0_1}(E_{k^0_2}(k^0_3))$
k^0_2	k^1_2	k^1_3	$E_{k^0_1}(E_{k^1_2}(k^1_3))$
k^1_2	k^0_2	k^1_3	$E_{k^1_1}(E_{k^0_2}(k^1_3))$
k^1_2	k^1_2	k^1_3	$E_{k^1_1}(E_{k^1_2}(k^1_3))$

Table 1: Garbled OR gate truth table

Since all the outputs are encrypted when evaluating the gate for input incoming from input wires and getting an output the player would have no information whatsoever about the other output .

Till now we described only how to create one garbled gate. a garbled circuit is made up of garbled gates and also has a output decryption table. the aim of this table is to map the result from the output wires of the circuit back to real values. So after an evaluation a result of k^{γ}_r can be converted to a 0 or 1 values depending on the output table .

Oblivious Transfer and 1 out-of 2 Oblivious Transfer

In cryptography an oblivious transfer protocol is a protocol by which a sender sends some information to the receiver, but is oblivious as to which information is received by the receiver. In a 1 out of 2 oblivious transfer protocol the sender has two messages m_0 and m_1 and the receiver has a bit β. The receiver then wants to receive the message m_β without getting any information about $m_{1-\beta}$ and without the sender learning β.

The protocol for 1 out of 2 oblivious transfer by Even, Goldreich and Lempel is a generic protocol but can be instantiated using RSA as follows:

1. The sender generates RSA parameters including the modulus N , the public exponent e, private exponent d and two public messages α_0 and α_1 and sends N, e, α_0, α_1 to the receiver.

2. The receiver picks a random value r, encrypts r, adds α_β, where β is either 0 or 1 depending on which message the receiver wants to get, to the encryption of r and sends the result q back to the sender.

3. The sender computes k_0 to be the decryption of $q - \alpha_0$ and k_1 to be the decryption of $q - \alpha_1$ and sends $m_0 + k_0$ and $m_1 + k_1$ back to the receiver.

4. The receiver knows k_β and subtracts this value from the corresponding part of the message received to get m_β

The Protocol

Based on the protocols on generating a garbled circuit and 1 out of 2 oblivious transfer we can now make the formal definition of Yao's protocol. In this protocol the sender generates a garbled circuit and sends it to the other party, henceforth the receiver. The sender and receiver then interact so that the receiver obtains the input wire keys that are associated with the input. The interaction between the sender and receiver is done by a 1 out of 2 oblivious transfer protocol so that the sender has absolutely no information about the input that the receiver is using for the circuit. After receiving the input values the receiver evaluates the circuit while always using 1 out-of 2 oblivious transfer protocol for additional input requested from the player that constructed the circuit[10, 1, 15].

1.1.2 Secret Sharing Based multiparty Computation Protocol

Secure multiparty computation based on secret sharing is a method which allows more than two parties to do information theoretically secure computation. the main difference between secret sharing method and Yao's method is that for any polynomial arithmetic function f, in secret sharing based secure multiparty computation there is no need to do any conversion of the function f to a circuit C. Secret sharing based secure multiparty computation is based on the principle of sharing an input secret α among n players and evaluate the polynomial arithmetic function f based on these secret shares, and when the secret shares are combined back the player gets $f(\alpha)$ instead of α. Secret sharing based secure multiparty computation will be explained later in more detail in section 2 including how to do secure multiparty computation for

each secret sharing scheme since the current implemented secure multiparty computation libraries in SecurePL are all secret sharing based.

1.1.3 Homomorphic Encryption

Homomorphic encryption is a form of encryption where a player can do a form of algebraic operation on plain text by doing algebraic operations on cipher text. Homomorphic encryption systems are all malleable by design, which means it is possible for an adversary to transform a ciphertext into another ciphertext which decrypts to a related plaintext. So homomorphic encryption systems might be deemed as not secure for data transfers.Even though this seems to be an unsuited property for an encryption system, the property of being malleable allows different players to do algebraic operations on secret inputs without revealing the data thus doing secure multiparty computation. Some example encryption systems that allow secure multiparty operations are

- RSA cryptosystem
- El Gamal Cryptosystem
- Goldwasser Micali Cryptosystem
- Benaloh Cryptosystem
- Paillier Cryptosystem

Examples:

In these examples $E(\alpha)$ denotes the cipher text of the plain text α

Unpadded RSA Cryptosystem: For public keys n and e the encryption of a plain text would be $E(\alpha) = \alpha^e \bmod n$. And the homomorphic property would be:

$$E(\alpha_1) \times E(\alpha_2) = \alpha_1^e \times \alpha_2^e \bmod n = (\alpha_1 \times \alpha_2)^e \bmod n = E(\alpha_1 \times \alpha_2 \bmod n)$$

Goldwasser Micali Cryptosystem: For public modulus n and quadratic residue x the encryption of a bit β would be $E(\beta) = r^2 x^\beta \bmod n$.And the homomorphic property would be:

$$E(\beta_1) \times E(\beta_2) = r_1^2 x^{\beta_1} \times r_2^2 x^{\beta_2} \bmod n = (r_1 \times r_2)^2 x^{\beta_1+\beta_2} \bmod n = E(\beta_1 \oplus \beta_2)$$

Paillier Cryptosystem: For public key modulus n and and base g the encryption of a message α would be $E(\alpha) = g^\alpha r^n \bmod n^2$. and the homomorphic property would be:

$$E(\alpha_1) \times E(\alpha_2) = g^{\alpha_1} r_1^n \times g^{\alpha_2} r_2^n \bmod n^2 = g^{\alpha_1+\alpha_2}(r_1 r_2)^n \bmod n^2 = E(\alpha_1 + \alpha_2 \bmod n)$$

Even though there are homomorphic cryptosystems to do addition or multiplication using only encrypted values as inputs, there is still no encryption system that allows both addition and multiplication using only encrypted values as input.

1.2 Compiler Design

Simply explained a compiler is a program that reads a program written in one language called the source language and translates this to an equivalent

program in another language called destination language. There are thousands of source languages, ranging from traditional programming languages such as FORTRAN to specialized languages. Destination languages are also varied; a target language can be a machine language of any computer between a microprocessor to a supercomputer or can be something else than machine code at all. Since machine code is just another programming language compilers can be also called translators between the input language to the destination language. A compiling operation consists of two steps: the first one is the analysis step where the compiler "understands" the input language and the second step is the synthesis step where the equivalent of the program is being produced in the destination language. We will explain some concepts about compiler design we heavily used for the implementation of SecurePL.

1.2.1 Lexical Analyzer

The lexical analyzer is the first phase of the compiler. It's main task is to read the input of characters from the source program and generate a list of tokens (sequence of characters that have a collective meaning) as output that will be used for syntax analysis. The lexical analyzer and parser are in close interaction with each other. Upon receiving a "get next token" command from the parser the lexical analyzer starts reading from the input stream until it can identify the next token. For the lexical analysis part of the project we used the UNIX tool flex as a scanner generator for lexical analyzing.

1.2.2 Parser

The parser obtains a list of tokens from the lexical analyzer and verifies that the string can be generated by the grammar of the source language. Thus a parser is a component in a compiler that takes the input of list of tokens and converts it to a data structure that is more suitable for further processing and checks for syntax error at the same time. The output the parser generates can be in the form of a parse tree, abstract syntax tree or any other hierarchical structure that can be traversed and used for further processing of the input source code.

1.2.3 Parse Tree and Abstract Syntax Tree

A parse tree is an ordered rooted tree that represents the syntactic structure of a string according to some formal grammar. In a parse tree the interior nodes are labeled by non-terminals of the grammar while the leaf nodes are labeled by terminals of the grammar.Even though they are related concepts the parse trees and abstract syntax trees are different constructs both used in compilers. Whereas a parse tree is the tree representation of the structure of the string an abstract syntax tree is the representation of the syntax of the input code. In an abstract syntax tree each node of the tree represents a construct occurring in the source code. It is "abstract" in the sense that there may be some constructs that do are not represented in the tree bur are present in the original code.

1.2.4 Syntax Directed Translation

Syntax-directed translation is a method of translating a string into a sequence of actions by attaching one such action to each rule of a grammar. Thus, parsing a string of the grammar produces a sequence of rule applications. And Syntax-directed translation provides a simple way to attach semantics to any such syntax. In syntax directed translation grammar symbols are associated with attributes to associate information with the programming language constructs that they represent and the values of these attributes are then calculated by the semantic rules associated with the production rules. There is no limit as what data an attribute may hold, it can hold any type on information. By the evaluation of these semantic rules the compiler may generate intermediate codes. Not only limited to generation of intermediate code the evaluation process can be used for evaluating anything.

1.3 Previous Secure Multiparty Computation Projects and Compiler

Besides our implementation of a secure multiparty computation programming language there are two recent projects that also focus on secure multiparty computation.

1.3.1 VIFF, the Virtual Ideal Functionality Framework

VIFF is a framework which allows you to specify secure multi-party computations in a clean and easy way. VIFF allows you to do secure multi-party computations, in which a number of parties (three or more at the moment)

execute a cryptographic protocol to do some joint computation. Built using python the VIFF project is mainly a set of python classes that enables a user to create SMC applications without the need to implement the actual protocols, like secret sharing or encryption, that are used in SMC operations.[11]

1.3.2 Secure Multiparty Computation Language

The Secure Multiparty Computation Language is a domain specific programming language for secure multiparty computation. It is a high-level, domain-specific language, which allows programmers to express concepts such as clients, server, and operations on secret values directly using a special syntax and control structures tailored to the domain of SMC. [12]

2 Secret Sharing

In cryptography, secret sharing refers to any method for distributing a secret among participants each of which get a share of secret. In secret sharing schemes there is a secret α which is shared among n participants. The secret can be recovered only if a certain condition is met, so that among n participants any group of $t+1$ have to come together to be able to recover the secret. This method is called *t-private* secret sharing .

In this chapter we first introduce some additive and polynomial secret sharing methods. Based on these methods we also show how secure multiparty computation can be done . After the introduction the basic additive and polynomial secret sharing method we will introduce a new secret sharing called *Pseudo Random Secret Sharing* which will be used later on for the secure multiparty comparison of secret shared values. After PRSS we will show a new novel modified secret sharing method which allows us to do secure multiparty computation in Asmuth Bloom Secret Sharing Scheme. And finally we will talk about bounds on how many multiplication and addition can be done without and with user interaction in mainly Shamir and Asmuth Bloom secret sharing scheme.

2.1 Additive Secret Sharing

As will become more clearer later while introducing *pseudo random secret sharing* we will also work with additive secret sharing schemes. In additive secret sharing scheme the sum of the shared values gives the secret. By definition this secret sharing scheme is a *t=n-1* private secret sharing scheme

where for less than n players it is impossible to recover any data about the secret. Additive secret sharing can be over any ring.

Additive secret sharing of a secret $\alpha \in \mathbb{F}$ simply consists of n random shares $\alpha_1, \alpha_2, ..., \alpha_n$ such that

$$\alpha = \sum_{i=1}^{n} \alpha_i$$

Here since the secret shares all are random elements chosen uniformly from the field $\mathbb{F}$ there is no way that *n-1* players can get any information about the secret using their shares.

2.2 Shamir Secret Sharing

Polynomial Interpolation

In the mathematical sub field of numerical analysis, polynomial interpolation is the interpolation of a given data set by a polynomial. In other words, given some data points (such as obtained by sampling), the aim is to find a polynomial which goes exactly through these points. Given a set of $n+1$ data points (x_i, y_i) where no two x_i are the same, one is looking for a polynomial p of degree at most n with the property[3]

$$p(x_i) = y_i \, , \, i = 0, 1, 2, 3, ..., n$$

Lagrange Interpolation Polynomial

In numerical analysis, a Lagrange polynomial, is the interpolation polynomial for a given set of data points in the Lagrange form. Given a set of $k+1$ data

points $(x_0, y_0), ..., (x_k, y_k)$ where no two x_i are the same, the interpolation polynomial in the Lagrange form is a linear combination

$$L(x) = \sum_{j=0}^{k} y_j l_j(x)$$

of Lagrange basis polynomials

$$l_j(x) = \prod_{i=j,\, i \neq j}^{k} \frac{x - x_i}{x_i - x_j}$$

Van der Monde matrix

In linear algebra, a Van der Monde matrix, is a matrix with a geometric progression in each row, i.e., an $m \times n$ matrix:

$$V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \cdots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \cdots & \alpha_m^{n-1} \end{bmatrix}$$

or

$$V_{i,j} = \alpha_i^{j-1}$$

The Van der Monde matrix is used for the dimension reduction and randomization steps of the secure multiplication protocol of Shamir Secret Sharing Scheme.

2.2.1 Decomposition and Recovery in Shamir Secret Sharing Scheme

For some given data α the goal is to divide it into n parts $\alpha_1, ..., \alpha_n$ such that:

1. knowledge of any $t+1$ or more pieces makes α computable
2. knowledge of any t or fewer α_i pieces leaves α completely undetermined (in the sense that all its possible values are equally likely).

The original shamir secret sharing scheme is based on linear interpolation. Given $t+1$ points in the 2-dimensional plane $(x_1, y_1), ..., (x_k, y_k)$. With distinct x_i 's , there is one and only one polynomial $q(x)$ of degree t such that $q(x) = y_i$ for all i [13].

Decomposition Process in Shamir Secret Sharing Scheme

To divide a secret α into pieces α_i , pick a random t degree polynomial $q(x) = a_0 + a_1 x + ...a_t x^t$ in which $a_0 = \alpha$, and evaluate:

$$\begin{aligned} \alpha_1 &= q(1) \\ \alpha_1 &= q(2) \\ \vdots &\ \vdots \ \vdots \\ \alpha_n &= q(n) \end{aligned}$$

Recovery Process in Shamir Secret Sharing Scheme

Given any subset of k of these α_i values (together with their identifying indices), it is possible to find the coefficients of $q(x)$ by interpolation, and

then evaluate $\alpha = q(0)$.

2.2.2 Secure Multiparty Computation in Shamir Secret Sharing Scheme

Given two secrets α and β shared by polynomials $f_\alpha(x)$ and $f_\beta(x)$ respectively of degree k the it is possible for the players to compute $c \times \alpha$,where c is a constant integer, $\alpha + \beta$ and $\alpha\beta$ using their local shares of α and β only.

The two linear operations are simple and for their evaluation we do not need any communication between the players. This is because if $f_\alpha(x)$ and $f_\beta(x)$ encode α and β , then the polynomials $h_\gamma(x) = c \times f_\alpha(x)$ and $k_\gamma(x) = f_\alpha(x) + f_\beta(x)$ encode $c \times \alpha$, $\alpha + \beta$ respectively. Thus to compute for example $\alpha + \beta$ each player P_i holding $f_\alpha(x_i)$ and $f_\beta(x_i)$ can locally compute $k_\gamma(x_i) = f_\alpha(x_i) + f_\beta(x_i)$. Likewise since c is a known constant P_i can compute $h_\gamma(x_i) = c \times f_\alpha(x_i)$. Furthermore $h_\gamma(x)$ is a random polynomial only if $f_\alpha(x)$ is random, and $k_\gamma(x)$ is a random polynomial if only one of $f_\alpha(x)$ or $f_\beta(x)$ was random. The only constraint for these operations is that $t \leq n - 1$.

The multiplication operation for two secrets is a bit harder. Assuming that $n \geq 2t + 1$ the free coefficient of the polynomial $h_\gamma(x) = f_\alpha(x) \times f_\beta(x)$ is $\alpha\beta$ but there are mainly two problems with encoding $\alpha\beta$ using the polynomial $h_\gamma(x)$:

1. The first and the most obvious one is that the degree of $h_\gamma(x)$ is now $2t$ instead of t . While this poses no immediate problems since $n \geq 2t + 1$ it would pose a problem for later multiplications since more multiplications would increase the degree of the multiplication even further.

2. The second problem is more subtle. $h_\gamma(x)$ is not a random polynomial of degree $2t$. For example $h_\gamma(x)$ as a product of two polynomials cannot be irreducible[4].

To overcome these two problems Ben-Or et. al.[4] proposes a method for randomization of the coefficients and a degree reduction method in their seminal paper. This method is later improved by Gennaro et. al.[7] Who do both dimension reduction and randomization in one step. The only drawback of both methods is that it requires interaction between participants to do multiplication of two polynomially secret shared data. In this thesis we used the method proposed by Gennaro et. al. for the multiplication and re randomization step for secure multiparty multiplication of shamir secret shared values.

Multiplication with Dimension Reduction

For each player P_i denote $f_\alpha(i)$ and $f_\beta(i)$ the shares that the player holds for the secret $f_\alpha(x)$ and $f_\beta(x)$ respectively. Then the product of $f_\alpha(x)$ and $f_\beta(x)$ is:

$$f_\alpha(x)f_\beta(x) = \gamma_{2t}x^{2t} + ... + \gamma_1 x + \alpha\beta = f_{\alpha\beta}(x)$$

For $1 \leq i \leq 2t+1$, $f_{\alpha\beta}(i) = f_\alpha(i) \times f_\beta(i)$ so we can write:

$$A \begin{bmatrix} \alpha\beta \\ \gamma_1 \\ \vdots \\ \gamma_{2t} \end{bmatrix} = \begin{bmatrix} f_{\alpha\beta}(1) \\ f_{\alpha\beta}(2) \\ \vdots \\ f_{\alpha\beta}(2t+1) \end{bmatrix}$$

Where the matrix denoted by $A = (a_{ij})$ is the $(2t + 1)$ by $(2t + 1)$ Van der Monde matrix defined as $a_{ij} = i^{j-1}$. Since A is non-singular and has an inverse. the first row of A^{-1} , donated by $(\lambda_1, ..., \lambda_{2t+1})$, are known constants. Then the previous equation implies that

$$\alpha\beta = \lambda_1 f_{\alpha\beta}(1) + ... + \lambda_{2t+1} f_{\alpha\beta(2t+1)}$$

Given polynomials $h_1(x), ..., h_{2t+1}(x)$ all of degree t which would satisfy $h_i(0) = f_{\alpha\beta}(i)$ for $1 \leq i \leq 2t + 1$ define $H(x) = \sum_{i=1}^{2t+1} \lambda_i h_i(x)$. Note that $H(0)$ is exactly $\lambda_1 f_{\alpha\beta}(1) + ... + \lambda_{2t+1} f_{\alpha\beta}(2t + 1)$ and hence $\alpha\beta$. Furthermore also notice that

$$H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j)$$

So based on these preliminaries, after multiplying their shares, each player P_i secret shares the multiplication of their shares with a polynomial $h(x)$ with the properties above then the polynomial $H(x)$, used for the sharing of $\alpha\beta$, is exactly of degree t It also is a random polynomial because the values $\lambda_1, ..., \lambda_{2t+1}$ are all non-zero values.

Input for player P_i: The secret shared values for α and β , $f_\alpha(i)$ and $f_\beta(i)$

1. Player P_i shares the multiplication $f_\alpha(i)f_\beta(i)$ by choosing a random polynomial $h_i(x)$ of degree t such that $h(0) = f_\alpha(i)f_\beta(i)$.
2. Player P_i gives each player P_j the secret shared value $h_i(j)$ for $1 \leq j \leq 2t+1$
3. Each player P_j computes his share of $\alpha\beta$ via a random polynomial H, i.e. the value $H(j)$, by locally computing the linear combination $H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j)$

Figure 1: Multiplication of Secret Shared Values Using Shamir Secret Sharing Scheme

2.3 Asmuth Bloom Secret Sharing

The Chinese Remainder Theorem

The Asmuth Bloom secret sharing scheme is based on the Chinese remainder theorem, which states that, given n congruences $x \equiv x_1 \bmod m_1$, $x \equiv x_2 \bmod m_2$, ... , $x \equiv x_n \bmod m_n$, where $m_1, ..., m_n$ are co-prime there exist exactly one solution in $[0, M-1]$ where $M = m_1 \times m_2 \times ... \times m_n$.[2, 9]

2.3.1 Decomposition and Recovery in Asmuth Bloom Secret Sharing Scheme

In the Asmuth Bloom secret sharing scheme, the secrest are integers in the interval $[0, p-1]$ where p is a prime number. Secrets are shared among n players. the secret sharing scheme has n public moduli: $p < m_1 < m_2 < ... < m_n$ which are chosen subject to the following:

1. $gcd(m_i, m_j) = 1$ for $i \neq j$

2. $gcd(p, m_j) = 1$ for all i

3. $\prod_{i=1}^{t+1} m_i > p \prod_{i=1}^{t} m_{n-i+1}$

Here, as before n denotes the number of participants, and $t+1$ is the number of different participant shares required to reconstruct the secret. Finally let $M_r = \prod_{i=1}^{t+1} m_i$.

Decomposition Process in Asmuth-Bloom Secret Sharing Scheme

The decomposition process begins with a secret α . Assuming that $0 \leq \alpha < p$, $y = \alpha + Ap$ where A is a blinding factor chosen subject to he condition:

$$0 \leq y < M_r$$

For $i = 1, \ldots, n$ the secret shares for each participant are then computed as

$$y_i = y \bmod m_i$$

Recovery Process in Asmuth-Bloom Secret Sharing Scheme

To recover x , it suffices to to find y . If y_{i_1}, y_{i_2}, ..., y_{i_r} are known then by, Chinese remainder theorem y is known modulo $N = \prod_{j=1}^{t+1} m_{i_j}$. Since $N \geq M_r$ this uniquely determines y thus α [2].

The Asmuth-Bloom secret sharing scheme is a perfect secret sharing scheme. Assuming a coalition with $t-1$ members, no information about the secret can be obtained by this coalition[2]. Using the perfect secret sharing scheme of Asmuth Bloom, to be able to do secure multiparty computation

some distance has to be created between the secure zone, which is the security zone where no information is revealed, and recoverable zone, which is the zone where the players can recover the secret. To create the necessary distance for secure multiparty computation, we made some modifications on the original Asmuth Bloom secret sharing scheme.

2.3.2 Secure Multiparty Computation in Asmuth Bloom Secret Sharing Scheme

The original asmuth bloom Secret sharing scheme does not allow neither additive nor multiplicative secure multiparty operations without losing the ability to recover the secret. But in this thesis we introduce a new modified Asmuth bloom Secret Sharing scheme which allows us to do addition and multiplication without losing the ability to recover the secret. The limits on how much addition and multiplication can be done based on secret sharing parameters will be discussed later in this chapter.

Modified Asmuth Bloom Secret Sharing Scheme

The parameters $p < m_1 < m_2 < ... < m_n$ are chosen according to the same conditions with the original Asmuth-Bloom secret sharing scheme. Finally let

$$
\begin{aligned}
M_r &= \prod_{i=1}^{t+1} m_i \\
M_s &= \prod_{i=1}^{s-1} m_{n-i+1} \\
M_n &= \prod_{i=1}^{n} m_i
\end{aligned}
$$

Here, as before n is the number of participants, $t+1$ the number of shares required for reconstruction of the secret and s the maximum number of participants who can gain absolutely no information (the minimum security parameter) about the secret when they combine their shares.

Decomposition Process in Modified Asmuth Bloom Secret Sharing Scheme

The decomposition process begins with a secret α . Assuming that $0 \leq \alpha < p$, $y = \alpha + Ap$ where A is a blinding factor chosen subject to the conditions:

1. $0 \leq y \leq M$
2. $p\,M_s \leq M \leq M_r \leq M_n$

Here M is the security parameter which defines the secure zone.For $i = 1, \ldots, n$ the secret share for each participant P_i is then computed as

$$y_i = y \bmod m_i$$

Recovery Process in Modified Asmuth Bloom Secret Sharing Scheme

This is the same as the original Asmuth Bloom secret sharing scheme.

Addition and Multiplication with Modified Asmuth Bloom Secret Sharing Scheme

With the modification we made to the construction phase of the Asmuth Bloom secret sharing scheme, we created a semi safe zone between $p\,M_s$ and M_r which allows us to do multiplication and addition until we reach the barrier of M_r . When two secrets are shared among n players with Asmuth Bloom secret sharing scheme, it is possible to do computation without revealing the secrets. Suppose that α and β have been shared as

$$\begin{aligned} \alpha_i &= \alpha + Ap \bmod m_i \\ \beta_i &= \beta + Bp \bmod m_i, \end{aligned}$$

for $i = 1, ..., n$ a possible secret share of $\alpha + \beta$ is

$$(\alpha + \beta) + (A + B)\,p = (\alpha + Ap) + (\beta + Bp) \equiv \alpha_i + \beta_i \bmod m_i,$$

which can be computed locally by each player. Likewise a secret share of $\alpha\beta$ is

$$(\alpha\beta) + (AB + \alpha B + \beta A)\,p = (\alpha + Ap)\,(\beta + Bp) \equiv \alpha_i\beta_i \bmod m_i.$$

The problem is that $(\alpha + \beta) + (A + B)\, p$ is potentially as large as $2M$, and $(\alpha\beta) + (AB + \alpha B + \beta A)\, q$ is potentially as large as M^2, when $\alpha + Ap$ and $\beta + Bp$ can be as large as M. If the result, $\gamma + Cp$ is larger than M_r e can no longer guarantee the unique recovery of the secret (In this case the sum or addition of two secrets).

In general after a additions and b multiplications the result $\gamma + Cp$ can be as large as $(a + 1)\, M^{(b+1)}$. To compute such an expression *and* be able to recover the result we need:

$$(a + 1)\, M^{(b+1)} \leq M_r \leq M_n$$

Notice that the secrecy threshold is not reduced since $qM_s \leq M \leq (a + 1)\, M^{(b+1)}$. For the calculation we use $M = pM_s$ because it is the smallest possible value, and it gives the largest number of possible additions and multiplications before an $overflow$ occurs.

2.4 Replicated Secret Sharing Scheme

The replicated secret sharing scheme is a bit different in term of setup from the previous secret sharing schemes introduced in this chapter. In this secret sharing scheme instead of building the secret sharing based on how many players out of n can reconstruct the secret, the main focus is on *which* players can reconstruct the secret by combining their secret shares. This secret sharing scheme was first introduced by Ito, Saito and Nishizeki in 1987[8] and further developed by Benaloh and Leichter in 1988[5].

In replicated secret sharing scheme the secret can be divided among a set of P trustees such that any "qualified subset" of P can reconstruct the secret and any unqualified subset cannot. The qualified subsets in this secret sharing scheme are called *access structures.* For any arbitrary access structure $\mathcal{A}$ a subset $\mathcal{S}$ of players is called a *qualified* set if $\mathcal{S} \in \mathcal{A}$ and an *unqualified set* if $\mathcal{S} \notin \mathcal{A}$. In this setup a maximal unqualified set means a set $\mathcal{S}$ of unqualified players where any unqualified set of $\mathcal{T}$ would hold $|\mathcal{T}| \leq |\mathcal{S}|$. So for a threshold structure t the number of elements in the maximal unqualified set would be $|\mathcal{S}| = t$.

Decomposition in Replicated Secret Sharing Scheme

In this secret sharing method a secret is shared among all the sets (up to $2^{|P|}$) of the access structure $\mathcal{A}$, divide the secret among each member of the set. So in the worst case, each of the n trustees have to hold on the order of 2^n shares.

Let A be all subsets of trustees and $\mathcal{T}$ be the set containing all maximal unqualified subsets so we can assume $\mathcal{T} \in A$. In a replicated secret sharing scheme a player secret shares a secret α by first creating additive secret shares over the number of elements in $\mathcal{T}$. I.e. by choosing random numbers α_T such that

$$\alpha = \sum_{T \in \mathcal{T}} \alpha_T$$

A secret share α_T is then distributed to all players P_i where $i \notin T$. Thus a replicated secret share of any player P_i will consist of shares α_T where $i \notin T$.

As an example consider the set of players $P = \{1, 2, 3, 4, 5\}$ for five players where each element is the unique id of each player. For a secret sharing with a threshold $t = 2$ where there is need for more than 2 players to recover the secret. the Set containing all maximal unqualified subsets would be

$$\mathcal{T} = \{\{1,2\}\,\{1,3\}\,\{1,4\}\,\{1,5\}\,\{2,3\}\,\{2,4\}\,\{2,5\}\,\{3,4\}\,\{3,5\}\,\{4,5\}\}$$

Since the number of elements in $\mathcal{T}$ is 10 we secret share α into 10 additive secret shares $\alpha_{\{1,2\}}, \alpha_{\{1,3\}}, \alpha_{\{1.4\}}, ..., \alpha_{\{4,5\}}$. Finally each player P_i get a secret share of α such that $T \in \mathcal{T}$ and $i \notin T$, thus the final distribution of secret shares would be:

Player ID	Secret shares In Possession
1	$\alpha_{\{2,3\}}\alpha_{\{2,4\}}\alpha_{\{2,5\}}\alpha_{\{3,4\}}\alpha_{\{3,5\}}\alpha_{\{4,5\}}$
2	$\alpha_{\{1,3\}}\alpha_{\{1,4\}}\alpha_{\{1,5\}}\alpha_{\{3,4\}}\alpha_{\{3,5\}}\alpha_{\{4,5\}}$
3	$\alpha_{\{1,2\}}\alpha_{\{1,4\}}\alpha_{\{1,5\}}\alpha_{\{2,4\}}\alpha_{\{2,5\}}\alpha_{\{4,5\}}$
4	$\alpha_{\{1,2\}}\alpha_{\{1,3\}}\alpha_{\{1,5\}}\alpha_{\{2,3\}}\alpha_{\{2,5\}}\alpha_{\{3,5\}}$
5	$\alpha_{\{1,2\}}\alpha_{\{1,3\}}\alpha_{\{1,4\}}\alpha_{\{2,3\}}\alpha_{\{2,4\}}\alpha_{\{3,4\}}$

In this secret sharing scheme a secret α is divided into $\binom{n}{t}$ secret shares and each player gets $\binom{n-1}{t}$ secret shares. So we see that replicated secret sharing scheme is a very inefficient secret sharing scheme.

Reconstruction in Replicated Secret Sharing Scheme

Since the secret is divided additively among the share holders, just as in an additive secret sharing schemes when more than $t+1$ players join their shares they will have all the shares required to additively reconstruct the secret α

2.5 Pseudo Random Secret Sharing

The last secret sharing method which will be used in secure multiparty comparison is pseudo random secret sharing. The method of pseudo random secret sharing is used to convert secret shares of a random number which is shared using a replicated secret sharing into a shamir secret shared random number. Thus it enables the users to create secret shared random numbers. This method is introduced by Cramer, Damgaard and Ishai, thus this section will be based on their paper: Share conversion, pseudo random secret sharing and its applications to secure comparison[6].

Even though in terms of reconstruction complexity, replicated secret sharing scheme is easier to compute (modular addition instead of Lagrange interpolation) its nature that the number of shares is $\binom{n}{t}$, and the amount of data each player has to hold as a share is $\binom{n-1}{t}$, makes the secret sharing scheme inefficient for larger numbers of players. Even though there is such a drawback to this secret sharing scheme it has a key property: the secret shares α_T created from a secret are totally *independent* from each other. So shares of a random secret $s \in K$ consists of replicated instances of random and independent values from K. This is not the case for any other secret sharing scheme like shamir secret sharing scheme because there the secret schares α_i are different point of a single random polynomial, thus all related to each other.Using this property of replicated secret scharing scheme it is possible to create shamir secret shared pseudo random values without any interaction among the players if the players have access to a previously shared randomness.

2.5.1 Conversion from Replicated Shares to Shamir Shares

Suppose that a secret α has been shared according to the t-private replicated scheme where A is an access structure and n the number of total players, thus

$$\alpha = \sum_{A \subseteq [n]:|A|=n-t} \alpha_A \tag{2.1}$$

where α_a has been given to all players in set A .

To convert these replicated secret shares to shares of α according to the t-private shamir secret sharing scheme, each player P_i is assigned the the point i on the shamir sharing polynomial. Now, for each set $A \subseteq [n]$ of cardinality $n - t$, let f_A be the unique t - degree polynomial such that:

1. $f_A(0) = 1$, and
2. $f_A(i) = 0$, for all $i \in [n] \setminus A$.

Each player P_j can then compute their share β_j of the shamir secret sharing by

$$\beta_j = \sum_{A \subseteq [n]:|A|=n-t, j \in A} \alpha_A f_A(j) \tag{2.2}$$

2.5.2 Pseudo Random Secret Sharing

In 2.5.1 we explained the conversion of a replicated secret share to shamir secret share. Based on this method the main observation would be if the secret α is some random value, all replicated shares α_A would be independent and and random. Hence the initially distributed α_A can be used as keys to a pseudo-random function $PRF\psi_{\cdot}(.)$, and as long as the participating players

agree on a common input a to the function, they can compute $\psi_{\alpha_A}(a)$ and replace this value with α_A in the formula (2.2) in 2.5.1. Thus each player P_j would compute its share β_j of the random secret α as:

$$\beta_j = \sum_{A \subseteq [n]:|A|=n-t, j \in A} \psi_{\alpha_A}(a) f_A(j)$$

In this method note that, if the field $\mathbb{F}$, where the elements are chosen, is of characteristic 2 this protocol can be modified so that the shared values are guaranteed to be 0 or 1 by choosing a pseudo random function (PRF) that outputs 0 or 1.

2.6 Bounds on non-interactive multiplication

2.6.1 Shamir Secret Sharing Scheme

Given two secrets α and β shared by polynomials $f_\alpha(i)$ and $f_\beta(i)$, respectively of degree t the aim is to calculate $\alpha\beta$. Recall that, when each participant multiplies their shares they get the multiplication of the secret in the constant term, we discussed earlier that this is not enough to do secure multiplication. Even under the assumption $n \geq 2t + 1$ which enables us to do at least one multiplication before the degree of the polynomial exceeds the number of participants and resulting an overflow we still have to do at least randomization of the coefficients of the resulting polynomial. To overcome this problem Gennaro et-al[7] introduced a one step degree reduction and randomization method. The drawback of this method is that it requires interaction between participants. So even under the assumption of $n \geq 2t + 1$ even after one multiplication, coefficient randomization and degree reduction has to be

applied to the secret shares so that even after multiplication the secret can be recovered using at most n participants. The method described in [7] is basically to randomize the coefficients of the new polynomial and re share the secret among n participants.

As a result we conclude that based on t and n where t is the degree of the polynomial of the secret, and n is the number of total participants it is not possible to do non-interactive multiplication. Even if $n \geq 2t+1$ because after each participant multiplies their share of secret the new polynomial is no more a random polynomial and re-randomization has to be done among participants to protect the security of the scheme.

2.6.2 Asmuth Bloom Secret Sharing Scheme

Here we will analyze the modified Asmuth-Bloom secret sharing scheme to see how many multiplications can be done before an *overflow* occurs and the secret turns out to be unrecoverable. Earlier in section 2.3 we mentioned that based on the chosen moduli for the Asmuth-Bloom secret sharing scheme we can define three values as:

$$\begin{aligned} M_r &= \prod_{i=1}^{t+1} m_i \\ M_s &= \prod_{i=1}^{s-1} m_{n-i+1} \\ M_n &= \prod_{i=1}^{n} m_i \end{aligned}$$

Later on, while explaining the decomposition protocol of the modified

Asmuth-Bloom secret sharing scheme, it is mentioned that to decompose a secret α among n participants, assuming that $0 \leq \alpha < p$, $y = \alpha + Ap$ where A is a blinding factor chosen subject to the conditions:

1. $0 \leq y \leq M$

2. $p\,M_s \leq M \leq M_r \leq M_n$

Theorem 1 *The number of non-interactive multiplications is related to the security parameters n and s*

To create the largest semi-safe zone between M and M_r we assume that M is the smallest possible value thus; $M = pM_s$ and M_r is the largest possible value thus M_n. Since we are using close relative prime moduli $m_1, m_2, ..., m_n$ and prime p we can assume that all of them are in the form of $2^b + x$. Here x is a variable different for each moduli/prime and b the base bit length of these number. Since all of the moduli are close integers after discarding the differences x. Then the values M_s and M_n can be rewritten as:

$$\begin{aligned} M_s &= \prod_{i=1}^{s-1} 2^b, \\ M_n &= \prod_{i=1}^{n} 2^b. \end{aligned}$$

and the M value would be

$$M = 2^b \times M_s.$$

After evaluating the multiplications we get

$$
\begin{aligned}
M_s &= 2^{(s-1)b}, \\
M_n &= 2^{nb},
\end{aligned}
$$

therefore

$$M = 2^{sb}.$$

Based on these calculations after $a-1$ multiplications for the secret still to be recoverable $M^a \leq M_n$ inequality has to hold.

$$
\begin{aligned}
M^a &\leq M_n \\
2^{(sb)\times a} &\leq 2^{nb} \\
log_2(2^{(sb)\times a}) &\leq log_2(2^{nb}) \\
sba &\leq nb \\
a &\leq \frac{n}{s}
\end{aligned}
$$

So based on n and our security parameter s, which is the number of participants which cannot gain absolutely no information about the secret, the number of multiplications that can be done can be written in terms of the ratio of the number of participants to the security parameter. So finally we can say that as long as the upper bound in the number of multiplications holds, the participants can do *non-interactive* multiplications using only local shares.

3 Comparison

In chapter 2 we introduced different secret sharing methods and how to do secure multiparty computation with them. In this chapter we will explain a method to do secure multiparty *comparison*.

In cryptography secure multiparty computation was first suggested as a problem in a paper by Andrew C. Yao[1]. The problem was later known as the millionaire's problem. The problem introduced by Yao was that two millionaires wanted to compare their wealth but neither of them wanted the other party to know his/her wealth. Yao presented a solution to this problem which we explained in section 1.1.1. But the solution to do comparison was not practical and was too computationally heavy to be used in real life applications. With the introduction of secret sharing based secure multiparty computation methods, algebraic operations could be done without any problem in terms of computational power. Even though algebraic secure multiparty computation was practically possible there still was no efficient or practical comparison protocol. The comparison protocol we implemented in SecurePL to do comparison of secret shared values is based on the protocol that was presented by Tomas Toft in Workshop on "Practical Applications of New Research in Cryptography"[1] and implemented for the VIFF project [14].

[1] http://people.sabanciuniv.edu/pedersen/panrc08/

3.1 Toft Comparison

3.1.1 Comparison

Let $[\alpha]$ and $[\alpha']$ be two secret shared values; the aim of the comparison protocol is to compute a secret sharing of which one of these two values is greater than the other one. Thus trying to calculate

$$[\beta] = [\alpha] \overset{?}{\leq} [\alpha'] \in \{0, 1\}$$

To do this comparison we assume that both α and α' are of bounded size (l bits) and that $\alpha \neq \alpha'$. Also all computations have to be modulo $p > 2^{l+k} + 2^k$ where l is the number of bits the inputs are and k is the security parameter. And finally to be able to do comparison the last assumption is that there is a source of secret randomness. For real life implementation the first two assumptions can be achieved without any problem. To get a shared source of randomness the pseudo random secret sharing method which was explained in section 2.5.2 will be used.

Since there is no algebraic method to do comparison the best method to do efficient multiplication without being unable to use algebraic methods is to convert the problem to another problem that can be solved efficiently.

3.1.2 Initial Problem Transformation

The problem of comparing two secret shared values is converted to a simpler problem which can be solved using algebraic methods. First the value α' is

subtracted from α in two's complement

$$[\gamma] = 2^l + [\alpha] - [\alpha'].$$

From this equation we just have to extract the $l + 1^{th}$ bit of γ to get which one of these two secret shares is greater. If the values of $l + 1^{th}$ bit is 1 then α is greater than α' and if it is 0 then α' is greater than α. So basically

$$[\gamma] - \left[\gamma \bmod 2^l\right]$$

is almost the result. Since it is impossible to open the secret $[\gamma]$ without revealing the difference between the secret values thus revealing information that can make one player learn the secret of another player we add a $l+k$ bit random integer which is shared as bits before reconstructing . For random bit values $[r_1], [r_2], ..., [r_{l+k}]$ generated using pseudo random secret sharing we first calculate

$$[r] = \sum_{i=1}^{l+k} 2^{i-1} [r_i]$$

to get a random number that is secret shared among all players. Then we add this random $l + k$ bit values to $[\gamma]$ and open the value

$$\delta = [\gamma] + [r]$$

thus we get

$$\left[\gamma \bmod 2^l\right] = \left(\delta \bmod 2^l - \left[r \bmod 2^l\right]\right) \bmod 2^l$$

the final $\bmod 2^l$ in this equation can be achieved by comparing $\left[r \bmod 2^l\right]$ and $\delta \bmod 2^l$

3.1.3 Comparing $[r]$ and δ

To do the comparison between $\left[r \bmod 2^l\right]$ and $\delta \bmod 2^l$ each party has to calculate:

$$[\eta_i] = [\sigma] + \delta_i - [r_i] + 3 \sum_{j>i} \delta_j \oplus [r_j] \,,$$

where the product of $[\eta_i]$'s will be 0 if $[r] > \delta$. After revealing $\lambda = [v] \prod [\eta_i]$ the result would be $(\lambda == 0) \oplus (\sigma == -1)$ where $\sigma \in \{1, -1\}$is generated randomly before calculation by each player. So finally the comparison boils down to $l + 1$ multiplications of the $[\eta_i]$ values.

4 SecurePL

4.1 Introduction

The main design goal for this thesis is to create a toolbox so that people with no knowledge about cryptography can create applications that use secure multiparty computation. To achieve this goal a two layered architecture is being used.

The first layer of SecurePL, which is also the layer interacting with the user, is the SecurePL programming language and compiler. In this layer the compiler takes the user created program and converts this SecurePL code to C++ code which is later fed into a C++ compiler to generate the final code.

The second layer acting transparently to the user, consists of arithmetic libraries which are used to do secure multiparty computation. These libraries handle all the operations needed to do secure multiparty computation from secret sharing to network communication. Even though this layer is transparent to the user, the modular architacture of SecurePL allows the end user to develop his/her own libraries to do secure multiparty computation.

In this section we will first introduce the tools used in the implementation of SecurePL and later on go into the implementation details of SecurePL by introducing the arithmetic libraries and programming language.

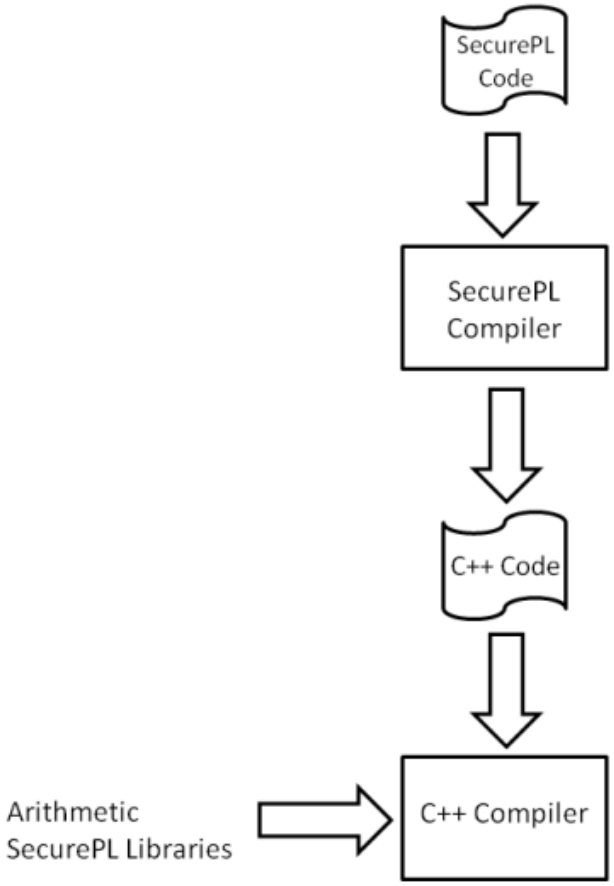

Figure 2: SecurePL Architecture

4.2 Base Protocols, Libraries and Tool kits

4.2.1 Flex

Flex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to flex. The flex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The flex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by flex, the corresponding fragment is executed.

Flex is a free software alternative to Lex the lexical analyzer. Flex is mostly used together with the parser generator Bison. The original flex was written by Vern Paxson around 1987. According to the GNU manual the description of flex is:

> "flex is a tool for generating scanners: programs which recognize lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, 'lex.yy.c', which defines a routine 'yylex()'. This file is compiled and linked with the '-lfl' library to produce

an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code..”

4.2.2 Bison

Bison is a general-purpose parser generator that converts an annotated context-free grammar description into a "Look Ahead Left-to-right Rightmost" (LALR) C or C++ parser which can parse a sequence of tokens that conforms to that grammar.In SecurePL Bison and flex are used in conjunction to generate a compiler that takes a simplified C-like language as input and produces a C++ code that works in parallel when run across different computers.

4.2.3 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers. It is basically a language independent protocol used to create parallel programs to run on different machines but in parallel. Both point to point and group communication is supported in MPI.

Its language independent structure and scalability and ease of use is the reason that MPI handles the communication operations in SecurePL. Also the player IDs used for the shamir secret sharing and other Id required algorithms are the rank values taken directly from MPI.

4.2.4 NTL

NTL is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. Since the standard 32 bit integer type of C++ is not enough to handle the operations especially for comparison and secure multiplication which requires up to 67 bit numbers even while working in a 32 bit field made it necessary to use a high precision big number library. Also the supplied number theoretical functions and algorithms makes the operations in fields much easier.

4.3 SecurePL Compiler

The SecurePL compiler is the first and only component that the end user will interact with. The aim of this compiler is to convert the input program to a C++ compatible secure multiparty computation based parallel application. The input language for the compiler is a simplified version of ANSI-C programming language. Since all the operations for multiparty computation and paralleling are handled by the underlying libraries which are the second component of the toolbox the end user does not have to have prior knowledge about secure multiparty computation or parallel programming.

While developing an application to do secure multiparty computation across different users the developer only has to have knowledge about which input is coming from which user in the computation group. By getting input from specific users any application which can be generated by the input language grammar can be created so that all the data is secret shared and

all computations are secure.

4.3.1 Input Language

The current version of SecurePL supports two variable types, constraint blocks and loops.

Supported Variable Types and Data types. The current supported variable types in SecurePL are **int** and **bool.** The integer type implemented in the current secure multiparty computation library is of unbounded size. Thus values greater than default C++ 32 bit integers can be used without any overflow or precision loss. The boolean type is the same as C++ bool and can have two values true or false. Even though ANSI-C or C++ supports array types the current version of SecurePL does neither allow nor support arrays or pointers. Also even though the converted code is converted to C++ code user defined types like classess or structs are not supported in the current version.

Supported loops and constraints. Among the different loop types supported by C/C++ the only supported loop type is the **While** loop. Since the other looping types like "for" or "do/while" are syntactic variations of the same operation we chose to implement only one type of loop to keep the input language simple. As for constraint blocks both **IF** and **IF/ELSE** blocks are supported by SecurePL. The usage of both these block types is implemented the same as C/C++ programming language.

4.4 Implemented SecurePL Libraries and Functions

4.4.1 Configuration

The Configuration class is basically the storage class of SecurePL. This class is transparent to the user. By providing a config.xml file containing the necessary configuration values:

- Prime modulo p.
- PRSS required replicated secret shares.
- The max number of bits k which is basically a predetermined value set by players.
- The security parameter k which is mostly 30.

The values in the configuration class are auto-loaded during program start. Thus the user has no access to the class values.

The Configuration Class Definition with comments describing each member:

```
class Configuration
{
public:
  Configuration();   //The default constructor of the class
  static ZZ n;       //The number of players in the network
  static ZZ t;       //The threshold
  static ZZ p;       //The prime modulo
  static ZZ l;       //The theoretic max number of bits
  static ZZ k;       //The security parameter for
  static ZZ ID;      //The ID of the current player given by
      MPI
  static std::map<string,unsigned char*> rT;    //The
     replicated secret shares of a random secret
};
```

Figure 3: Configuration Class

4.4.2 sint

The sint class is also another user transparent class implemented. This class handles the secret shares of the data in the program. The main purpose of this class is to be an int class placeholder. For the current version of SecurePL when a player initializes an **int** type variable in the input program this variable's type is converted to **sint** while converting to C++ code. the constructor of sint is responsible for secret sharing and distributing actual values to secret shared values. When a sint variable is constructed using a constant in-code value the player with the lowest ID first secret shares the open data and then distributes the secret shares to other players as appropriate. Player specific construction of sint variables are handled by the read functions which will be explained later.

The class definition with comment for explaining each member is as fol-

lows:

```
class sint
{
public:
        sint(); //default constructor which sets value to 0
        sint(ZZ val); //consructor that secret shares val
           and assigns the secret shared value to won
        sint(sint& rhs); //copy constructor
        ~sint();
        ZZ value;   // value of the secret share of input
           integer
        void Hide(ZZ val);   // hides an integer input and
           returns the corresponding secret share[ID]
public:
        sint operator+(sint& b);          //mathematical
           operators to do algebraic operations
        sint operator-(sint& b);
        sint operator*(sint& b);
        sint operator/(sint& b);
        sint operator*(unsigned int& b);
        sint operator=(unsigned int& b);
};
```

Figure 4: sint Class

4.4.3 ShamirSharer

The shamirsharer class is the class that handles creation of secret shares. When constructing the class the shamir secret sharing parameters have to be specifically defined for each instance of the class and there is no method for setting some static parameter. This implementation choice was on purpose because especially for comparison both secret sharing over $GF(p)$ and $GF(256)$ are needed to accommodate this need each instance has to be re-initialized with new parameters. And also just like the previous classes this

class is also user transparent which means the user does not have to know anything with regards to shamirsharer to write its SecurePL code.

The class definition with explanatory comments:

```
class ShamirSharer
{
public:
        ShamirSharer(void); //Default constructor
        ~ShamirSharer(void);
        ShamirSharer(ZZ num, ZZ t, ZZ prime); //Constructor
private:
        ZZ n; // number of participants
        ZZ r; // number of participants needed for recovery
        ZZ p; // prime p for the field to work in
public:
  SharedData* ShareData(ZZ data); // secret shares an input
      based on n,r parameters
  ZZ ReconstructData(SharedData* shares); //reconstructs
     the sharer by lagrange interpolating them
  ZZ ReconstructData(SharedData* shares, ZZ recombPoint);
     //reconstruct by lagrange interpolation over an
     arbitary point
};
```

Figure 5: ShamirSharer Class

4.4.4 ShamirLib

This header file contains all the secure multiparty computation operations that are currently implemented. The previous classes are used by this class to do any algebraic operation securely. Addition, subtraction, multiplication and comparison of sint values are implemented in this library. also initialization of common parameters and communication through MPI with other players is also an integral part of this library. Configuration settings are also

handled by this library. There are some routines implemented in SecurePL for interaction with the player or mathematical operations.

Getting Input From User. The implemented input output operations to get data from sure are also integral parts of the Shamble library. The currently supported read operations which has to be known by the end user to get some information from the user while writing SecurePL code are:

- fRead(File source file, int &destination, unsigned int userId): Here the source file is a File object which contains the data that has to be read by the user, destination is the variable that will be loaded, and lastly the last parameter userId is the user that has the data for that specific variable.

- sRead(int &destination,int userId): this function is the same as the above function but the only difference is that instead of using a file to read the data this function gets the input from the standard input.

Revealing secret variables Besides reading and getting information revealing secret shared variables is also implemented in this class. Two modes of revealing are implemented in the current version. when the user wants to reveal some variable in SecurePL the user can choose either to reveal it to only one player with specific Id or to the whole network. The functions that implement this operation are

- RevealVariable(int variable,int userId): This function reveals the variable to a specific user ith userId

- RevealAll(int variable): This functions reveals the secret variable to all users in the network.

Comparison The current version of SecurePl allows only comparing of two secret shared values and no comparison against a constant values is allowed.

Algebraic Operations. The implemented and currently available algebraic operations in SecurePL are addition, multiplication, subtraction and division. Other algebraic operations are planned to be added in future versions of SecurePl.

4.5 User Implemented Libraries

As of now there is only ShamirLib implemented that can be used to do secure multiparty computation, but there is no restriction on that and the end user can implement its own secure multiparty computation library. In the case where the user wants to use his/her own library and not the built-in library while compiling the SecurePL code the user has to supply the compiler with the classes to replace **int** and **bool** also has to supply with the **library header file name** that will replace the built-in library. Also another requirement is that all library implementations for SecurePL have to have a **Initialization()** function to initialize library parameters. If there no such need for a function like this it can be implemented as an empty function. The compilation options for SecurePl are:

- ***SecurePL source_file destination_file***: To use SecurePl compiler with built in libraries

- ***SecurePL source_file destination_file int_replacement_class bool_replacement_class main_library_header***: To use with custom libraries.

5 Conclusion

Since the concept of secure multiparty computation was introduced lots of different secure computation methods were proposed. But despite the fact that there exist many different algorithms and methods to do secure multiparty computation none of them had any practical usage. The reason was either because there was no practical method to do secure multiparty computation or because the methods and algorithms were too complicated and no framework existed to ease the burden on developers trying to develop secure multiparty enabled applications. The aim of this thesis was to make a programming interace and toolbox so that **even people with no knowledge about security or cryptography** could develop secure multiparty applications.

We achieved this goal with the current version of SecurePL compiler and implemented ShamirLib. Thanks to the compiler and accompanying library it is possible for a person to write an application without being concerned about security or cryptography. A simple example would be a solution to the problem introduced by Yao in 1982[15], the very same problem that started the concept of secure multiparty computation: The millionaire's Problem. There is a sample solution offered by Yao which we destribed in 1.1.1 but to use this solution one has to have knowledge of encryption, oblivious transfer and circuit evaluation besides knowing how to write code. To contrast this we offer a solution where a person who knows how to code can write a solution to Yao's millionaire problem without getting hindered by lack of knowledge about security or cryptography. An example source code to solve Yao's Millionaire problem would be

```
void main()
{
    int a;
    int b;

    sRead(a,1);
    sRead(b,2);

    if(a<b)
    {
        cout << "B is richer";
    }
    else
    {
        cout << "A is richer";
    }
    return;
}
```

Figure 6: Sample SecurePL Code

In this example the program asks Alice and Bob to input their wealth using standard input and then reveals which one of these two players is richer. Since all data handled by the implemented library is secret shared no one gains information about the other player's wealth. Also since the implementation of the library stores **all** data as secret shared values even the player cannot extract the his/her original data once the data has been secret shared and distributed to all players, because after distribution the original data is not kept in memory whatsoever.

Further development of SecurePL might allow users to differentiate between secret and non secret integers. At the moment there is no difference between a regular and a secret integer. By this non discriminating structure

we can be sure that everything is secure but the main drawback resulting from this operation is that even for simple loop operations or iterations more than necessary processing power is needed. So giving the ability to a user to choose which variables have to be secret and which ones not will increase the performance drastically.

Input output operations in SecurePL also have some room for improvement. Especially the data read function has no iterative read property and data is read from a source file always starting from the beginning. This operation can limit the use of read from file operations.

Finally we can say that the current version of SecurePL is ready to be used for daily operations so that these operations can be done in parallel and securely. But just as mentioned above there is still some room for improvement so that the practical secure multiparty computation toolbox becomes more powerfull and fast and will be capable of handling more complex operations.

References

[1] Andrew. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.

[2] C. Asmuth and J. Bloom. A modular approach to key safeguarding. *Information Theory, IEEE Transactions on*, 29(2):208–210, 1983.

[3] Kendall Atkinson. *An Introduction to Numerical Analysis*. Wiley, 2 edition, 2006.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1988. ACM Press.

[5] Josh C. Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In Shafi Goldwasser, editor, *CRYPTO*, volume 403 of *Lecture Notes in Computer Science*, pages 27–35. Springer, 1988.

[6] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography*, pages 342–362, 2005.

[7] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *PODC '98: Proceedings of the seventeenth annual*

ACM symposium on Principles of distributed computing, pages 101–111, New York, NY, USA, 1998. ACM Press.

[8] M. Itoh, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *IEEE Globecom*, pages 99–102, 1987.

[9] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[10] Y. Lindell and B. Pinkas. A proof of yao's protocol for secure two-party computation, 2004.

[11] Mikkel Krøigård Thomas Pelle Jakobsen Jakob Illeborg Pagter Sigurd Meldgaard Martin Geisler, Tomas Toft. Viff, the virtual ideal functionality framework. World Wide Web electronic publication.

[12] Janus Dam Nielsen. Secure multiparty computation language. World Wide Web electronic publication.

[13] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[14] Tomas Toft. Secure integer computation with applications in economics, July 2005.

[15] Andrew C. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, 1982.