UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
INSTITUTE OF COMPUTER SCIENCE

Yao, Yanjun

Automated Security Proofs of Secret Shared Protocols

Master Thesis (20AP)

Supervisor(UT): Jan Willemson, PhD
Co-Supervisor(TKK): Antti Ylä-Jääski, PhD

Author: .... ".........." May 2008
Supervisor: .... ".........." May 2008

Allowed to defend
Professor: .... ".........." ... 2008

TARTU 2008

Contents

1 Introduction **5**

1.1 Problem Statement . . . 5

1.2 General Solutions . . . 5

1.3 Outline of the Thesis . . . 6

2 Secure Multiparty Computation **7**

2.1 Definition . . . 7

2.2 Two-Party Computation . . . 8

2.2.1 Security Goal . . . 8

2.2.2 Important Primitives . . . 9

2.2.3 Circuit Evaluation . . . 10

2.3 Approach of Evaluating Security . . . 12

2.4 General Multiparty Computation . . . 14

2.4.1 Nonadaptive Case . . . 14

2.4.2 Adaptive Case . . . 16

3 Secret Sharing **20**

3.1 Definition . . . 20

3.2 Shamir's Scheme . . . 21

3.3 Verifiable Secret Sharing . . . 23

3.3.1 General Scheme . . . 23

3.3.2 Feldman's Scheme . . . 25

3.4 Computation with Shamir's Shares . . . 25

4 Sharemind Secure Computation Protocol **28**

4.1 Additive Secret Sharing . . . 28

4.2 Introduction of SHAREMIND . . . 29

4.3 Multiplication Protocol of Three Parties . . . 30

4.4 SHAREMIND Share Conversion and Bit Extraction Protocols . . . 33

5 Simulation **37**

5.1 Universal Composition . . . 37

5.2 Simulating the real world . . . 39

5.3 Formal Proof of SHAREMIND Protocols . . . 40

6 Overview of Our Implementation **44**

6.1 AutoProver Infrastructure . . . 44

6.2 Implementation Notes . . . 45

6.3 The Parser . . . 45

6.3.1 Introduction of the Grammar . . . 46

6.3.2 Generating the Full Protocol . . . 50

6.3.3 Future Extension . . . 52

6.4 The Protocol Analyzer . . . 52

6.5 The Simulation Generator 55

7 Conclusion 57

Hajusarvutusprotokollide automaatsed turvatõestused 58

References 59

A Manual of Our Program 61

A.1 Specific Features 61

A.2 Compilation and Execution 61

A.3 AutoProver API Routines 63

1 Introduction

1.1 Problem Statement

Nowadays, as a result of rapid decreasing of the price of storage devices, peoples prefer to store their information digitally in databases. Those databases, which contain the personal, medical and financial information of the data donors, are classified as sensitive. Only authorized organizations have the right to process the sensitive data.

However, the extensive implementation of online information systems not only make the use of data more convenient, but also provide an easier way to abuse the data. Hence, a lot of research organizations try to devise their methods for processing the sensitive data without compromising the privacy of individuals. In the last two decades, a lot of protocols meeting this requirement have been proposed. Consequently, how to persuade other persons that these protocols really keep the individuals' privacy becomes a new problem. In other words, they need to provide a method to prove the security of protocols.

In this thesis, we address the proof method on a specific infrastructure proposed by Dan Bogdanov. In his master thesis [4], he considers protecting sensitive data as a multiparty computation task. He also proposes several secret shared protocols for computing with the data without leaking the privacy of any person under the assumption that only few participants can be corrupted. Our goal is to devise a method, which can automatically prove that in all secret shared protocols no party can figure out other parties' secrets.

1.2 General Solutions

To give an impression of how to prove the soundness of a method, we will discuss two possible solutions. Suppose the method is written in a form as a protocol. For the multiparty computation tasks, the protocols depict which parties participate in the computation and what should they do. We want to give a persuasive proof of the security of these protocols.

Solution One: Informal proof. Let us denote the parties that can be corrupted by the adversary at the same instance of running the protocol as corrupted set. In this solution, we prove the security of a protocol by collecting all the messages generated and received by parties in the corrupted set, then manually check if those parties working together can somehow figure out the secrets they should not known. The proof result is presented in human languages, which indicates whether those messages can compromise some secrets. However, although the proof can be easily understood, it is not very convincing.

Solution Two: Formal proof. This solution is based on a theory that if for any adversary that the output distribution of the protocol and the output distribution of an ideal functionality are indistinguishable from each other, then the protocol is secure. We prove the security of the protocol by stating that there exists a simulator which can translate any message exchanging between the corrupted parties and the ideal functionality in such a way that the corrupted parties can not tell if they are using the protocol or the ideal functionality.

In this thesis, we build an implementation to prove the security of secret shared protocols based on the second solution.

1.3 Outline of the Thesis

In this thesis, we describes the concepts used in designing the secret shared protocols and the theories used in designing our implementation. The content of this thesis is as follows:

- Chapter 2 introduces the notion related to multiparty computation. It contains the basic definitions and the overview of the accomplishments in this area. The basic idea of how to prove the security of a protocol is also mentioned in this chapter as a preliminary to the following chapters.
- Chapter 3 includes the secret sharing schemes. Besides the basic concepts of secret sharing, we also introduce different secret sharing schemes and how to do computation on Shamir's shares.
- Chapter 4 shows the infrastructure of Dan Bagdanov's implementations and the protocols proposed by him. The informal proofs of the security of those protocols are also presented here.
- Chapter 5 includes the basic theory of our method for proving. We introduce the theory of universal composition and shows how to prove the security of a protocol by constructing a simulator.
- Chapter 6 presents our implementation AutoProver, which can automatically generate the simulation result. In this chapter, we show the infrastructure of AutoProver, and make a detailed introduction of each component of the program.
- Appendix A contains the introduction of how to use the AutoProver program.

2 Secure Multiparty Computation

Suppose Alice and Bob are two millionaires, and they want to know who is richer. Clearly, they could just decide to announce to each other how much money do they have, and both of them could determine who is the richer one. However, revealing one's net worth to the other party means losing one's face if the other party is much richer.

In that case, what they need is a method allowing them to figure out who is richer in such a way that both of them get no clue about the other party's net worth. Sometimes, they may even want to carry out this method over a distance.

In this section we will discuss such a method in detail. Section 2.1 contains the definitions related to the multiparty computation. Section 2.2 discusses two-party computation as a preliminary. The Real vs. Ideal world approach of evaluating secure multiparty computation is presented in section 2.3. The detailed discussion of multiparty computation is presented in section 2.4.

2.1 Definition

Assume there are n parties $P_1, \ldots, P_n$ work together to compute a multivariable function $F(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$.

Definition 2.1. *Secure Multiparty Computation is a protocol to evaluate function F in such a way that both the maximum privacy of the inputs is preserved and the correctness of the outputs is guaranteed.*

In other words, except for the values of input x_i and output y_i, each party P_i can get no more information.

In that case, we can model the two millionaires' example as follows. Let Alice's net worth be x_1 and Bob's net worth be x_2. According to the example, the function F here is a greater than function, which outputs the result of comparison without leaking any clue about the values been compared. Several solutions have been proposed by Yao [22].

Following the lines of [5, 10, 6, 8], in order to define what is the security of multiparty computation, we must define an adversary first. An adversary is a malicious entity, whose aim is to prevent the users from achieving their goals. An adversary may corrupt a set of parties. Once a party is corrupted, the adversary gets all data held by the party, such as all the messages the party has sent or received so far.

The adversaries can be distinguished as passive or active. *Passive adversaries* (also called eavesdropping adversaries) gather information without modifying the behaviors of the parties. Usually, they attack after the execution of a protocol has completed. However, *active adversaries* do not only read the messages, but also can modify the messages of corrupted parties. The other distinction is between static and dynamic adversaries. *Static adversaries* (also called nonadaptive adversaries) control an arbitrary but fixed set of corrupted parties. *Dynamic Adversaries* (also called adaptive adversaries) can choose which party to corrupt during the execution of the protocol, based on the information gathered so far. Both passive and active adversaries can be static or dynamic.

In order to let the parties collaborate on computing, the model of communication must be taken into account. There are two basic models of communication. The first one is *cryptographic model*. In this model, the adversary is able to read all messages sent between all parties, and modify messages exchanged between the corrupted parties. No message transmitted between honest parties can be modified by the adversary. The second one is *information-theoretic model*. In this model, parties communicate with each other over pairwise secure channels, which means the adversary can not get access to any message exchanged between honest parties. Hence, unlike the cryptographic model whose security can only be guaranteed in cryptographic sense, the second model is much stronger. Even an adversary with unbounded computing power can not read the messages exchanged between honest parties in the information-theoretic model.

However, if an adversary corrupts all parties, no protocol can be secure. Therefore, we need to specify the limitation on the subsets that can be corrupted by adversaries, which is denoted as *threshold adversary structure*. Suppose the protocol is secure while no more than t parties are corrupted and the set of all parties is denoted as P.

Definition 2.2. *The threshold adversary structure α is a set of all subsets of P, where $\alpha = \{R \subseteq P : |R| < t\}$, and R denotes the subset of P.*

The adversary may corrupt any one set of parties in the threshold adversary structure. If the adversary can corrupt a set of parties C, it can also corrupt all subsets of C.

2.2 Two-Party Computation

In two-party computation, there is no need to take extra security considerations such as collusion of parties into account. Hence secure two-party computation is considered as a simple scenario in secure multiparty computation. In this section, we will briefly talk about the primitives, the security of two-party computation and Yao's circuit evaluation as the preliminary of the next subsection. For further discussions on two-party computation please refer to papers [10, 16, 22].

2.2.1 Security Goal

According to the two millionaires' example as we mentioned above, which is a typical two-party computation scenario, secure two-party computation should meet the following requirements:

1. **Correctness**: all parties should get the correct end results.
2. **Privacy**: each party learns nothing more than his input and what is implied by the result.
3. **Fairness**: each party can get a result.

The first two requirements are compulsory, but in some systems such as the blind signature scheme introduced in article [11], the requirement of fairness is not met.

In order to achieve the correctness of the results, both of the parties should be *semi-honest*, they follow the protocol but each of them tries to figure out the private value of the other party.

2.2.2 Important Primitives

Usually secure two-party computation uses oblivious transfer, commitment schemes and some other computationally expensive primitives. In the following, we make a brief introduction of the first two primitives, both of them are based on the assumption that trapdoor one-way functions exist.

A trapdoor one-way function is a function that is easy to compute, but difficult to find its inverse without knowing a special information, called the "trapdoor". In other words, in a trapdoor one-way function $f(x)$, when the value of x is randomly chosen, the result of $f(x)$ can be computed in polynomial time. However, no probabilistic polynomial-time algorithm can compute the preimage of $f(x)$ with a non-negligible probability, unless it knows the "trapdoor".

Oblivious Transfer is a protocol by which a sender sends some information to a receiver, but remains oblivious about what is received. As a preliminary, we just talk about the general process of "One Out of Two Oblivious Transfer" in this thesis.

Suppose there are a sender S and a receiver R. S holds two secret inputs s_0 and s_1, and R holds a secret selection bit sr. After a number of exchanges of information between S and R, R gets the secret input s_{sr} at the end of the protocol without knowing any information about the other secret input s_{1-sr}. On the other hand, S can not figure out the value of the selection bit sr held by R. Hence, both the privacy of the sender and receiver are guaranteed.

This process can be visualized as S puts two secret values into two boxes, each box contains one value, then S locks the boxes and passes them to R. R just has a key which can open one of the boxes, he opens one box and gets the value.

In order to have a better understanding of what oblivious transfer is, we take the protocol proposed by Even, Goldreich, and Lempel [13] as an example. It is a general 1-out-of-2 oblivious transfer protocol which can be instantiated with any public key algorithm as follows:

1. Assume $\mathcal{M}$ is the message space. By using the public key algorithm, S generates a public key pk and a secret key sk. Then S randomly chooses two messages x_0 and x_1, where $x_0, x_1 \in \mathcal{M}$. After that, pk, x_0 and x_1 are sent to R.

2. R randomly generates a message $k \in \mathcal{M}$, and let $c = Enc_{pk}(k)$. $q = c + x_b$ is computed by R and sent to S, where $q \in \mathcal{M}$.

3. S computes $k_0 = Dec_{sk}(q - x_0)$ and $k_1 = Dec_{sk}(q - x_1)$, and sends $s_0 + k_0$ and $s_1 + k_1$ to R. Note that all messages computed and sent are in the message space $\mathcal{M}$.

4. As R knows k_b, he can subtract k_b from $s_b + k_b$ sent by S to obtain the secret s_b.

Commitment Scheme is a method that allows a user to commit to a secret value while preserving the user's ability to reveal the committed value later.

The sender S sends an encrypted value to a receiver R. S may send the decryption key to R after several message exchanges to reveal its secret. Hence, before the revealing, R can not figure out what the secret is, and S can not change the encrypted value that has already been sent to R.

This scheme can be imagined as follows: S puts a value in a locked box, and gives the box to R. The value s in the box is a secret, as R can not open the box. On the other hand, because R obtains the box, S can not change the value in the box any more. When S wants to show that he really put s in the box, S just needs to send the key of the box to R, then R can check the value by opening the box.

We take the bit commitment scheme proposed by Moni Naor [18] as an example. Assume that there is a cryptographically secure pseudo-random number generator G, which generates a $3n$-bit number from a n bits input. Following the protocol, the sender S commits to a bit b as follows:

1. Commitment: R selects a $3n$ bit random number r and sends r to the receiver S. Then S generates a n bit number x and computes $y = G(x)$. If $b = 1$, S sends y to R, otherwise S sends $r \oplus y$ to R.

2. Reveal: To reveal the secret, S sends x to R, then R computes $G(x)$ and compares the result with what he received from S to get the value of b.

2.2.3 Circuit Evaluation

Yehuda Lindell and Benny Pinkas [17] have given a detailed description and a sound proof of Yao's two-party computation protocol [22] in their work. Following their lines, in this section, we will present Yao's garbled circuit and show Yao's general protocol in details.

Assume there are two parties, Alice and Bob. Alice's input is x_A, and Bob's input is x_B, where $x_A \in \{0,1\}^{n_A}$ and $x_B \in \{0,1\}^{n_B}$. They want to get the result of the computation on these two inputs, without leaking their inputs to the other party. Hence Alice (or Bob) constructs a garbled circuit C, which evaluates the function $f : \{0,1\}^{n_A} \times \{0,1\}^{n_B} \rightarrow \{0,1\}^n$.

There are three kinds of gates in C: input gate, internal gate and output gate. Each gate has two input wires and an output wire. The values of input gates are the bits of x_A and x_B. The internal gates, which are determined by the function they compute, take inputs from other two gates. The output value of an output gate is its input, and its output wires can not be used as input wires of any other gates. In a word, a garbled circuit C is composed by a number of garbled gates.

As C is a boolean circuit, what each internal gate g needs to compute is a function $f_g : \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$. Assume g takes inputs from wires w_1 and w_2, and delivers output to the wire w_3. In order to keep the privacy of data, two random values $k_{w_i}^0$ and $k_{w_i}^1$ are specified for each wire w_i as the keys. They are generated in such a way that even if one party knows the value $k_{w_i}^b$, where $b \in \{0,1\}$, he can not figure out if $b = 0$ or $b = 1$.

Assume g gets two inputs k_1^{α} and k_2^{β}. What g wants to compute is $k_3^{f_g(\alpha,\beta)}$, while hiding other three values $k_3^{f_g(1-\alpha,\beta)}$, $k_3^{f_g(\alpha,1-\beta)}$ and $k_3^{f_g(1-\alpha,1-\beta)}$. As a result of that, we need to implement a special private key encryption scheme (G, E, D), in which the distribution ranges of the plaintext encrypted by different keys are different and the ranges of corresponding cyphertexts are elusive from each other, and the ciphertext can only be correctly decrypted by the correct key, otherwise the output will be $\perp$. Therefore, we use the four possible inputs k_1^0, k_1^1, k_2^0, k_2^1 as encryption keys and build a garbled computation table as Table 1:

Input Wire w_1	Input Wire w_2	Output Wire w_3	Garbled Computation
k_1^0	k_2^0	$k_3^{f_g(0,0)}$	$E_{k_1^0}(E_{k_2^0}(k_3^{f_g(0,0)}))$
k_1^0	k_2^1	$k_3^{f_g(0,1)}$	$E_{k_1^0}(E_{k_2^1}(k_3^{f_g(0,1)}))$
k_1^1	k_2^0	$k_3^{f_g(1,0)}$	$E_{k_1^1}(E_{k_2^0}(k_3^{f_g(1,0)}))$
k_1^1	k_2^1	$k_3^{f_g(1,1)}$	$E_{k_1^1}(E_{k_2^1}(k_3^{f_g(1,1)}))$

Table 1: Garbled Computation Table

According to the table, the output $k_3^{f_g(\alpha,\beta)}$ of gate g is computed as follows: for each possible garbled computation value in the fourth column of Table 1, compute $D_{k_2^{\beta}}(D_{k_1^{\alpha}}(E_{k_1^i}(E_{k_2^j}(k_3^{f_g(i,j)}))))$, where $i, j \in \{0, 1\}$. If less than three decryptions return $\perp$, then abort the output. Otherwise, the decrypted non-$\perp$ value is the value of $k_3^{f_g(\alpha,\beta)}$.

As mentioned above, a garbled circuit is composed from a number of garbled gates. Hence, C is constructed according to the following rules:

1. As C is constructed to evaluate the function $f : \{0,1\}^{n_A} \times \{0,1\}^{n_B} \rightarrow \{0,1\}^n$, it contains $n_A + n_B$ input gates and n output gates.

2. There are m wires in C, which are in charge of transferring the data between gates. The rule of labeling wires is that the output wires of the same gate are using the same label. The wires of the same label hold the same pair of keys, and all keys are chosen independently on the others.

3. The outputs of input gates and internal gates can be the inputs to an arbitrary number of other gates. However, the outputs of output gates can not be used as inputs.

4. A garbled computation table is built for each gate, and the output tables which depict how to decrypt the circuit outputs are created.

In a word, the entire garbled circuit C of function f consists of garbled gates, a garbled computation table for each gate and the output tables.

Now let us have a look at Yao's general two party computation protocol, which implements the garbled circuit to compute the function f and uses oblivious transfer to keep the privacy of the input data. The protocol is as follows:

1. Alice constructs a garbled circuit C, which evaluates the function $f : \{0,1\}^{n_A} \times \{0,1\}^{n_B} \rightarrow \{0,1\}^n$ as described above, and sends C to Bob.
2. Set the output wires of Alice's input gates be denoted as $w_1, \ldots, w_{n_A}$, which correspond to Alice's input bits $x_{A_1}, \ldots, x_{A_{n_A}}$. Alice sends Bob the keys $k_1^{x_{A_1}}, \ldots, k_{n_A}^{x_{A_{n_A}}}$.
3. Let the output wires of Bob's input gates be $w_{n_A+1}, \ldots, w_{n_A+n_B}$. Alice and Bob execute a 1-out-2 oblivious transfer protocol to obliviously transfer the keys $k_{n+i}^{b_i}$ to Bob, where b_i is the ith bit of Bob's input.
4. After Bob receives the garbled circuit C and the $n_A + n_B$ keys, he computes the circuit and gets the result $f(x_A, x_B)$. After that, Bob sends $f(x_A, x_B)$ to Alice, then both of them output the result.

2.3 Approach of Evaluating Security

Great effort has been put into formulating definitions that can adequately express the intuitive notion of the security of multiparty computation in different adversary models. The basic idea underlying all these efforts is to guarantee that the computational distance between running a secure protocol and carrying out an idealized computational process where security is guaranteed is negligible.

Beaver [1] introduced the following methodology for defining secure multiparty computation. First, an ideal model is formulated. In this model, the evaluation of a multiparty function is perfectly secure. Second, we execute a secure protocol π for evaluating some functions of parties' inputs in the real-life setting, under the requirement that it is "equivalent" to evaluating the function in the ideal model. In other words, an ideal world specifies the required behavior of a protocol and rules out unwanted ones, while the real world is where protocols and attacks are executed on. When the protocol π is secure, its output is indistinguishable from the output of the ideal world.

Based on the work of Cramer and Damgård [11], in the following part of this section, we will introduce **The Ideal vs. Real World Approach** in detail.

As shown in Figure 1, we assume that there is an incorruptible party called *Ideal Functionality* F in the ideal world structure. There is no communication between the parties, instead they hand their inputs to F, who computes the desired outputs and hands them back. As F is incorruptible, it always correctly executes the required commands in such a way that except for what is supposed to be sent to the party, no more information is leaked. F contains the following interfaces: an input and an output port for each party, and the input and output ports for communicating with the adversary. On the right part of Figure 1 is the structure of the real world. In the real world, all parties directly communicate with each other.

In both ideal world and real world, the adversary controls a set of corrupted parties. It only learns (passive adversary) and perhaps modifies (active adversary) the inputs and outputs of corrupted parties. In the ideal world, the adversary exchanges messages with ideal functionality F on behalf of corrupted parties through corrupted input and output ports. In the real world, except for the ports for communicating with other parties, each

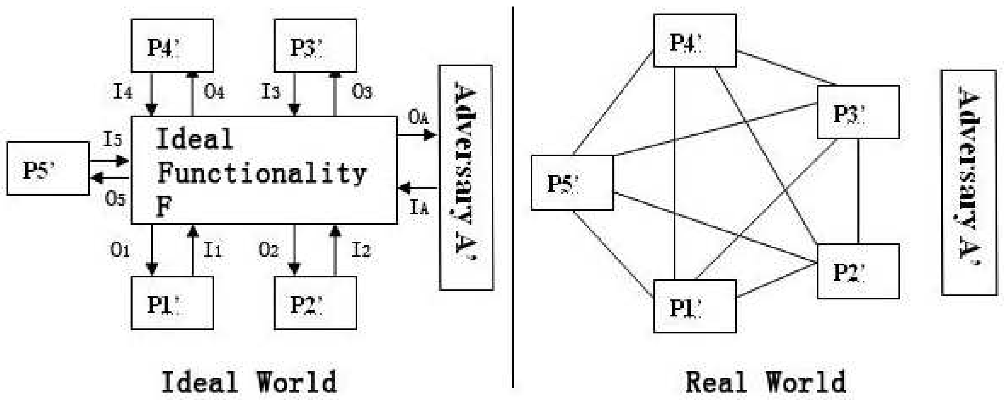

Figure 1: Structure of Ideal and Real world

party has corrupted input and output ports to exchange information with the adversary. We suppose the corrupted parties still follow the protocol, no party may abort the protocol before completing the computation.

Each round in the ideal world follows the following rules:

1. **Input:** Each party initializes its inputs. The corrupted parties send their inputs to the adversary. F reads input of uncorrupted party P_i from port I_i. F gets the inputs of corrupted parties through port I_A.

2. **Compute:** F does the computation on inputs and get the outputs.

3. **Output:** F sends output to each uncorrupted party P_i on port O_i, the outputs of corrupted parties are sent on port O_A.

Each round in real world follows the following rules:

1. **Input:** According to the protocol, all parties send messages to the parties they should send to.

2. **Compute:** Each party locally does computation on its inputs and gets the outputs.

3. **Output:** Each party sends the output to other parties according to the protocol. The corrupted parties send their inputs, outputs and received messages to the adversary.

We say that a protocol π is *perfectly secure* if for any adversary attacking π in the real world, there is an adversary in ideal world that can induce the same output distribution in the ideal world as the output distribution of the real world. In other words, the protocol π is perfectly secure if using the protocol and using the ideal functionality are indistinguishable to the real world adversary. How to simulate this process will be discussed in section 5 in detail.

2.4 General Multiparty Computation

Usually a n-party computation, where n is larger than two, is an extension of two-party computation. However, under the condition of n parties computing together, more security considerations such as collusion of parties should be taken into account. The problem of secure multiparty computation takes different forms according to the difference of the power of adversaries, the underlying network, the amount of distrust the parties have in each other and in the network.

In the past few decades, a lot of works on multi-party computation in various security models have been done. A breakthrough was achieved by Ben-Or, Goldwasser, Wigderson [2] and Chaum, Crepeau, Damgård [8] independently in late 1980s. They demonstrated that there exists a protocol that evaluates a function f with perfect security while less than a third of the total number of parties are corrupted. More specifically, the protocol can tolerate a malicious subset of size $\frac{n}{2}$ with an adaptive and passive adversary, and tolerate a malicious subset of size $\frac{n}{3}$ with an adaptive and active adversary. A year later, Rabin and Ben-Or [20] proved that in the presence of authenticated broadcast channels, statistical security of any function is guaranteed while less than $\frac{n}{2}$ partes are corrupted by active adversaries.

In the following part, we will discuss the multiparty computation protocols in semi-honest nonadaptive adversary case and semi-honest adaptive adversary case separately. The definition of these protocols follows the Ideal vs. Real world approach as we mentioned above. For Further reading on the composition of multiparty computation protocols please refer to papers [6, 5].

2.4.1 Nonadaptive Case

In this section, we define the semi-honest nonadaptive adversary case in the Ideal vs. Real world approach presented in section 2.3. We first describe the power of ideal adversary and the ideal process. After that, we present the real world adversary and real world model.

The goal of Ideal vs. Real world approach is to let the protocol and ideal functionality be indistinguishable in the entire environment. The environment contains not only the adversary, but also other objects such as the system that implements the protocol. Hence, we also need to consider auxiliary input from the environment.

Ideal Adversary V is a computationally unbounded machine, which controls the behavior of corrupted parties. As a nonadaptive adversary, suppose that V has already corrupted less than t parties ($t < \frac{n}{2}$), and it can not corrupt more parties since the protocol is executed. Once V corrupts a party, it gets the party's input, output and all messages sent and received by that party. A corrupted party stays being corrupted until the end of the protocol. As a semi-honest adversary, V represents the corrupted parties to follow the protocol while wondering about what the secret is. V can not modify any messages.

In **The ideal process** of semi-honest nonadaptive case, similar to what we mentioned in section 2.3, there is an incorruptible third party T which implements the ideal functionality F as defined in section 2.1. The trusted third party is in charge of computation and communication with parties and the adversary. The ideal process is as follows:

1. **Input:** Each party P_i gets its private data S_i. Note that the private data can be secrets generated by P_i itself, or shares computed from secrets by using secret sharing schemes introduced in section 3. The adversary V sees all the inputs of corrupted parties, but it can not get any information about uncorrupted parties' inputs.

2. **Computation:** Each party sends its input to the incorruptible third party T. After evaluating the function, T sends the output to corresponding parties.

3. **Output:** Each honest party follows the protocol and outputs y_i, however the corrupt parties output $y_j = \perp$, meanwhile the adversary outputs some function of the information gathered during the computation in the ideal process.

Denote the auxiliary input from the environment as z. Let $ADV_{F,V}(x_1, \ldots, x_n, z)$ denotes the output of ideal adversary V on auxiliary input z, while interacting with each party P_i, whose input is S_i, and the ideal functionality F. Let $IDEAL_{F,V}(x_1, \ldots, x_n, z)$ denote the output of ideal process, where

$$IDEAL_{F,V}(x_1, \ldots, x_n, z) = (ADV_{F,V}(x_1, \ldots, x_n, z), y_1, \ldots, y_n)$$

The **Real Adversary** A is similar to the ideal adversary V. A is also a computationally unbounded machine that controls the behavior of corrupted parties. A is an adversary, who halts when more than t parties are corrupted. As a nonadaptive adversary, A can just choose one subset of parties to corrupt. Once a party is corrupted, it stays as corrupted until the end of the protocol. Besides learning all information held by corrupted parties, A can also get auxiliary input from the environment.

Now we describe **The Real Process**. The protocol we need to evaluate is denoted as π. In the real world, there is no trusted third party. All parties and the adversary communicate with each other directly. The computation proceeds in rounds. In each round, the corrupted parties send their own messages after they get and learn the messages sent by uncorrupted parties. Each round works as follows:

1. **Input:** All parties generate their messages for this round according to the protocol. Then each party follows the protocol and sends the messages to other parties. At the end of this phase, uncorrupted parties receive all the messages that should be addressed to them in this round. Then the corrupted parties send all messages they hold, including the messages they generated and received, to the adversary.

2. **Computation:** All parties compute their output locally as required by the protocol.

3. **Output:** The uncorrupted parties output the results they computed in this round. On the other hand, the corrupted parties send their output to the adversary and output $\perp$. Then the adversary outputs some arbitrary function of all the data it knows.

The whole process of real world model, which integrates all rounds, is as follows:

1. **Input:** Each party P_i gets its private data S_i. Note that the private data can be secrets generated by P_i itself, or shares computed from secrets by using secret sharing schemes introduced in section 3. The adversary A sees all the inputs of corrupted parties and the auxiliary input z from the environment.

2. **Computation:** Computation proceeds in rounds:

   - Initialize the round number as 0.
   - According to the protocol, if any uncorrupted party has not finished its computation, do one round of computation as described above and increase the round number by 1

3. **Output:** The honest parties follow the protocol and output y_i, while the corrupt parties output $y_j = \perp$. The adversary outputs some function of the information, which is gathered during the computation in the real world process.

Let $ADV_{\pi,A}(x_1, \ldots, x_n, z)$ denote the output of real adversary A on auxiliary input z, while interacting with each party P_i, whose input is S_i, running a protocol π. Let $REAL_{\pi,A}(x_1, \ldots, x_n, z)$ denote the output of real process, where

$$REAL_{\pi,A}(x_1, \ldots, x_n, z) = (ADV_{\pi,A}(x_1, \ldots, x_n, z), y_1, \ldots, y_n) .$$

We require that for any real world adversary A there exists an ideal adversary V such that the output of real world model is computationally indistinguishable from the output of the ideal world model. More specifically, for any auxiliary input z, the difference between the distribution of $REAL_{\pi,A}(x_1, \ldots, x_n, z)$ and $IDEAL_{F,V}(x_1, \ldots, x_n, z)$ are negligible.

Definition 2.3. *Let us have an ideal functionality F and a n-party protocol π. If for any semi-honest nonadaptive real world adversary A, there exists a semi-honest nonadaptive ideal world adversary V such that $REAL_{\pi,A}(x_1, \ldots, x_n, z)$ and $IDEAL_{F,V}(x_1, \ldots, x_n, z)$ are computationally indistinguishable, where V's running time is polynomial in the running time of A, we say that the protocol π is perfectly secure in semi-honest nonadaptive case.*

2.4.2 Adaptive Case

In this section, we define the semi-honest adaptive adversary case using the same approach as the one used in the nonadaptive case. As an extension of the nonadaptive case, it is more complex. Thus more security concerns need to be considered:

1. The adversary may decide if he can get more information by corrupting one party than corrupting other parties based on the information it handled.

2. After several rounds of computation have taken place, the adversary may have advantages of figuring out the secrets, when it sees the internal data of newly corrupted parties.

In a word, as new parties are corrupted while proceeding with the computation, the private data of uncorrupted parties can not be regarded as safe.

As the adaptive case are quite similar with the nonadaptive one, in this section, we will concentrate on their difference. For the similar notation, please refer to the former section.

In the adaptive case, as an adaptive adversary, **Ideal Adversary** V can iteratively corrupt up to t parties ($t < \frac{n}{2}$).

Similar to nonadaptive case, there is an uncorrectable third party T which implements the ideal functionality F in the **Ideal Process**. The ideal process of Adaptive Case is as follows:

1. **First Corruption:** Before proceeding with the computation, V gets auxiliary input z from the environment. Based on z it decides the first subset of parties to corrupt. The corrupting process is done iteratively according to what information V gathered so far. A corrupted party remains corrupted for the rest of the computation. After repeating this process several times, the adversary gets its original set of corrupted parties at the end of this stage.

2. **Input:** Each party P_i gets its private data S_i. Each private data can be secrets generated by P_i itself, or shares computed from secrets by using secret sharing schemes. The adversary V sees all the inputs of corrupted parties, but it can not get any information about uncorrupted parties' inputs right now.

3. **Computation:** Each party sends its input to the incorruptible third party T. After evaluating the function, T sends the output to corresponding parties.

4. **Second Corruption:** After V gets computation outputs from the trusted third party, a new iteration of corruption begins. V decides the next party to corrupt based on the messages gathered. While a new party is corrupted, V sees new data, which includes the party's input, output and all exchanged messages. As required, the number of corrupted parties can not exceed t.

5. **Output:** Each honest party follows the protocol and outputs y_i, however each corrupt party outputs $y_j = \perp$, meanwhile the adversary outputs some function of the information gathered during the computation in the ideal process.

As defined in nonadaptive case, let $ADV_{F,V}(x_1, \ldots, x_n, z)$ denote the output of ideal adversary V on auxiliary input z, while interacting with each party P_i, whose input is S_i, and the n-party computation function F. Let $IDEAL_{F,V}(x_1, \ldots, x_n, z)$ denote the output of ideal process, where

$$IDEAL_{F,V}(x_1, \ldots, x_n, z) = (ADV_{F,V}(x_1, \ldots, x_n, z), y_1, \ldots, y_n) \ .$$

The **Real Adversary** A is an adaptive adversary, which collects auxiliary input z from the environment ahead. Then A iteratively chooses the original subset of parties to corrupt, the number of corrupted parties grows while executing the protocol. Once a party is corrupted, it stays as corrupted until the end of the protocol.

. In the **real world process**, the protocol we need to evaluate is denoted as π. The computation proceeds in rounds as follows:

1. **Input:** All parties generate their messages for this round according to the protocol. Then each party follows the protocol and sends the message to other parties. It is required that uncorrupted parties should receive all the messages addressed to them in this round. Then the corrupted parties send all messages they hold, which include the messages they generated and received, to the adversary A.

2. **Computation** All parties compute their output locally as required by the protocol.

3. **Second Corruption:** All corrupted parties send their outputs of this round to adversary A, then a new iteration of corruption begins. A decides who is the next corrupted party based on the messages gathered so far. A can get the following new information from the new corrupted party: the party's input, output and all exchanged messages. As required, the number of corrupted parties can not exceed t.

4. **Output** The uncorrupted parties output the results they computed in this round. Meanwhile, the corrupted parties send their outputs to the adversary and output $\perp$. Finally, the adversary outputs some arbitrary function of all data it knows.

The whole process of real world model, which integrates all rounds, is as follows:

1. **First Corruption:** Before proceeding with the computation, A gets auxiliary input z from the environment. Based on z it may decide the first subset of parties to corrupt. The corrupting process is done iteratively according to the information A gathered so far. A corrupted party remains corrupted for the rest of the computation. After repeating this process several times, the adversary gets its original set of corrupted parties at the end of this phase.

2. **Input:** Each party P_i gets its private data S_i. The adversary A sees all the inputs of corrupted parties, but he can not get any information about uncorrupted parties' inputs right now.

3. The **computation** and **adaptive corruption** proceeds in rounds:

   - Initialize the round number as 0.
   - While any uncorrupted party has not finished its final computation in the protocol, execute one round of computation mentioned above, and increase the round number by 1.

4. **Output:** The honest parties follow the protocol and output y_i, however the corrupt parties output y_j = $\perp$, then the adversary outputs some function of the information gathered during the computation in the ideal process.

Let $ADV_{\pi,A}(x_1,\ldots,x_n,z)$ denotes the output of the real adversary A on auxiliary input z, while interacting with each party P_i, whose input is S_i, π is the protocol to be evaluated. Let $REAL_{\pi,A}(x_1,\ldots,x_n,z)$ denote the output of real process, where

$$REAL_{\pi,A}(x_1,\ldots,x_n,z) = (ADV_{\pi,A}(x_1,\ldots,x_n,z), y_1,\ldots,y_n)\ .$$

We require that for any real world adversary A there exists an ideal adversary V such that the output of real world model is computationally indistinguishable from the output of the ideal world model.

Definition 2.4. *Let us have an ideal functionality F and a n-party protocol π. If for any semi-honest adaptive real world adversary A, there exists a semi-honest adaptive ideal world adversary V such that $REAL_{\pi,A}(x_1, \ldots, x_n, z)$ and $DEAL_{F,V}(x_1, \ldots, x_n, z)$ are computationally indistinguishable, where V's running time is polynomial in the running time of A, we say that the protocol π is perfectly secure in semi-honest adaptive case.*

3 Secret Sharing

Secret sharing, proposed by A. Shamir [21] and G.R.Blakley [3] in 1979, refers to any method for distributing a secret among a group of participants. In the end, each of the parties gets a share of the secret. Individual share is useless, the secret can only be reconstructed while enough number of shares are combined together. It is a useful technique to protect sensitive data. By distributing the secret into several shares and spreading shares among several parties, the adversary needs to compromise at least a threshold number of parties to gain the information of the secret, otherwise nothing can be figured out. We can say that while spreading the shares, the risk of compromising the secret is spreading with them.

Being a good technique to preserve the privacy of the data, secret sharing is an important component of multiparty computation. In this section, we will explore the possibility of getting correct output by performing operations on shares without reconstructing the secret value all the time. In Section 3.1, some definitions of secret sharing are introduced as preliminaries. Sections 3.2 and 3.3 present Shamir's Scheme and Verifiable Secret Sharing in detail. How to compute with Shamir's shares is discussed in section 3.4.

3.1 Definition

Formally, in a secret sharing scheme, there is a dealer and a set of parties. The dealer holds a secret s, and distributes shares of s privately to parties. Only certain specific subsets of parties polling their shares together can figure out what the secret is, while others have no information about it.

Following the lines of Shamir's paper [21], we define (t, n)-threshold scheme as follows:

Definition 3.1. *Assume a scheme A is able to divide a secret S into n pieces $S_1, \ldots, S_n$ in a way that:*

1. *From any subset of shares, while the size of the subset is at least t, $t \leq n$, secret S can be reconstructed uniquely and efficiently.*

2. *From the subsets containing less than t shares, no information of S can be released.*

*Such a scheme is called a (t, n)-**threshold scheme**.*

Shamir [21] illustrated a classical example of this scheme. Assume all cheques in a company need to be digitally signed. Each executive holds a magnetic card, and each card is entitled with different access right according to the holder's position. The company's signature generating device can be triggered while at least one of the following events happens: (1)The president presents his or her card; (2)The vice president and a board member present their cards together; (3)Three board members show their cards together.

This problem can be easily solved by using the $(3, n)$-threshold scheme, where n is the number of shares. To meet the requirement, we set that the card of the president to contain three shares, the vice president's card holds two shares and cards of board members each has one share. The signature generating device is perfectly secure, as it does not contain any internal sensitive data. No executive remembers the secret key. In that case, except for the president, only one person colluding with a malicious adversary can not sign the cheque.

In most cases, secret sharing schemes are threshold schemes with different threshold values. As we mentioned in section 2.4, the secure multiparty computation protocols can be perfectly secure while a passive adversary corrupts less than $\frac{n}{2}$ parties or an active adversary corrupts less than $\frac{n}{3}$ parties. In order to meet this requirement, the thresholds of secret sharing schemes must be at least $\frac{n}{2}$ or $\frac{n}{3}$ in corresponding adversary models.

In multiparty computation, we are more interested in the relation between computing on shares and computing on secrets. Because of the properties of (t, n)-threshold schemes that any single share is useless and the secret can not be retrieved from less than t shares, it is much more secure to transfer and compute on the shares than do the same operation on secrets. Hence, we prefer homomorphic secret sharing scheme, which is defined as:

Definition 3.2. *Let us have two secrets S and T, each of them being divided into n shares denoted as $S_1, \ldots, S_n$ and $T_1, \ldots, T_n$. For any binary operations $\oplus$ and $\otimes$, a scheme is ($\oplus, \otimes$)-homomorphic secret sharing scheme, if the reconstructed value of shares $S_1 \otimes T_1, \ldots, S_n \otimes T_n$ is the same as the value of $S \oplus T$.*

We can also define homomorphic secret sharing scheme with a constant as:

Definition 3.3. *Let us have a secret S and a constant C, where S is divided into n shares denoted as $S_1, \ldots, S_n$. For any binary operations $\oplus$ and $\otimes$, a scheme is ($\oplus, \otimes$)-homomorphic secret sharing scheme, if the reconstructed value of shares $S_1 \otimes C, \ldots, S_n \otimes C$ is the same as the value of $S \oplus C$.*

3.2 Shamir's Scheme

Shamir's scheme is a (t, n)-threshold scheme, which was proposed in paper [21]. It is based on Lagrange interpolation polynomial over finite fields. A Lagrange interpolation polynomial is an interpolation polynomial for a given set of data points in the Lagrange form, which is defined as follows:

Definition 3.4. *Let us have a finite field $\mathcal{F}$. For any set of numbers $x_1, \ldots, x_n \in \mathcal{F}$, the* ***Lagrange basis polynomials*** *are in the form:*

$$l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}$$

Definition 3.5. *Let us have a set of t data points $(x_1, y_1), \ldots, (x_t, y_t)$, where all x_i are different. The* ***Lagrange interpolation polynomial*** *is a linear combination of Lagrange basis polynomials:*

$$q(x) = \sum_{j=1}^{t} y_j l_j(x)$$

Theorem 3.1. *Let $x_1, y_1, \ldots, x_t, y_t \in \mathcal{F}$, where $\mathcal{F}$ is a finite field, such that the values $x_1, \ldots, x_t$ are all different. There exists exactly one polynomial q of degree at most $t-1$, such that $q(x_i) = y_i$ for all $i \in \{1, \ldots, t\}$.*

Let $\mathcal{F}$ be a finite field, the n parties participate in Shamir's (t, n)- scheme are denoted as $P_1, \ldots, P_n$, where $n < |\mathcal{F}|$. The field $\mathcal{F}$, threshold t, names of parties, and the content of the protocol are known to all parties.

The dealer plays an important role in this scheme. It computes shares of input secrets and distributes them to the parties. While the secrets need to be reconstructed, parties send their shares to the dealer, then the dealer can retrieve the secret. We assume that there is a separate private communication channel between each party and the dealer, hence the adversary can not eavesdrop the channel.

The protocol has two phases:

1. **Distribution:** The dealer has a secret $S \in \mathcal{F}$, let $f_0 = S$. The dealer randomly generates values $f_1, \ldots, f_{t-1} \in \mathcal{F}$ as the coefficients. Then he defines the polynomial as $q(x) = f_0 + f_1 x + f_2 x^2 + \ldots + f_{t-1} x^{t-1}$. The dealer secretely sends to each P_i his share $S_i = q(i)$.

2. **Reconstruction:** The parties $P_{i_1}, \ldots, P_{i_t}$ together know that $q(i_1) = S_{i_1}, \ldots, q(i_t) = S_{i_t}$ and the degree of $q(x)$ is less than t. From a collection of no less than t shares, the dealer can find the coefficients of $q(x)$ by interpolation, where $q(0) = S$.

Theorem 3.2. *The reconstruction phase of Shamir's secret sharing scheme can correctly reconstruct the secret.*

Proof. By analyzing the distribution phase, we can get that $x_i = i$, where $i \in \{1, \ldots, n\}$, hence all x_i are different. According to Theorem 3.1, the polynomial $q(x)$ of degree at most $t-1$ is uniquely determined. By Lagrange interpolation, it is clear that reconstruction works correctly. If no less than t points are given, the polynomial of degree $t-1$ that passes through these points is of course that unique polynomial $q(x)$. $\square$

Theorem 3.3. *In Shamir's secret sharing scheme, the secret can only be retrieved while t or more shares combining together.*

Proof. Suppose the dealer collects only $t-1$ shares, denoted as $S_{i_1}, \ldots, S_{i_{t-1}}$. Then for each possible secret $S' \in \mathcal{F}$, there are t points $(0, S'), (i_1, S_{i_1}), \ldots, (i_{t-1}, S_{i_{t-1}})$ uniquely determining a polynomial $q_i(x)$ of degree at most $t-1$ that passes through all of them.

As the coefficients $f_1, \ldots, f_{t-1} \in \mathcal{F}$ are randomly chosen, from the $t-1$ shares collected from parties, each possible value of the secret is uniformly distributed. So the real secret S can not be retrieved from $t-1$ shares. Similarly, collecting even less shares can not reconstruct the secret either. $\square$

An important property of Shamir's scheme is that it is possible to compute S without reconstructing the full polynomial $q(x)$. Assume that what the parties $P_{i_1}, \ldots, P_{i_t}$ are interested in is the secret value $q(0)$, not the polynomial. According to Lagrange interpolation formula

$$S = \sum_{j=1}^{t} S_{i_j} \prod_{k \neq j} \frac{i_k}{i_k - i_j} ,$$

where the secret is computed as a linear combination over the shares with public coefficients.

3.3 Verifiable Secret Sharing

When participants in Shamir's scheme are malicious, there are two threats must be considered:

1. Malicious dealer may send inconsistent shares to parties.
2. A malicious party can input a wrong share to the recovery protocol.

If any of them happens, the reconstructed result is not equal to the secret, and the honest parties may not figure out who are malicious. As a remedy, the verifiable secret sharing schemes, in which the parties commit to the shares they have sent, were first proposed by Chor, Goldwasser, Micali and Awerbuch [9] in 1985.

A secret sharing scheme is verifiable if auxiliary information is provided to let parties verify the consistency of their shares. It ensures that if the dealer is honest, the cheaters can not get any information of the secret S, and all honest parties are able to reconstruct S no matter what actions the cheaters have taken. If the dealer is malicious, either the honest parties can discriminate it from honest one and abort the protocol, or S is uniquely fixed by the shares held by honest parties and reconstructed correctly regardless of cheaters' behaviors.

As we mentioned in section 2, multiparty computation is accomplished by implementing secret sharing schemes on inputs, and manipulating the shares to evaluate the computation function. Verifiable secret sharing is an important component of secure multiparty computation, especially when adversaries are active and take full control of corrupted parties. As the shares need to be verifiable, the correctness of results is guaranteed.

3.3.1 General Scheme

There are four important primitives in verifiable secret sharing scheme: (t, n)-threshold secret sharing schemes, commitment schemes, zero-knowledge interactive proofs and point-to-point channels. The first two primitives have been introduced in former sections, the introduction of zero-knowledge interactive proof and point-to-point channels is as follows.

Zero-knowledge Interactive Proof is an interactive method for one party to prove to another party that a statement is true without revealing the content of the secret presented by this statement. In a zero-knowledge proof, if a statement is false, the probability of a cheating prover convincing the honest verifier that it is true is very small. If the statement is true, the honest verifier will be convinced of the statement's veracity by honest prover, and cheating verifier learns nothing more than this fact. For further reading please refer to [19].

Point-to-point Channel is also called a broadcast channel. We assume that not only private channels exist between the dealer and the parties, but there are also private channels between each pair of the parties. Hence, each party can send messages to all other parties, and each message is sent in such a way that, during the transmission, even malicious adversary can not change its content. The originator of messages can be easily established by recipients.

As we have handled all the primitives, we can now build the general verifiable secret sharing protocol as follows:

1. **Distribution:** The dealer holds a secret $S \in \mathcal{F}$, then he computes shares $S_1, \ldots, S_n$ by using a (t, n)-threshold secret sharing scheme. For each S_i, a commitment C_i is computed. After that, the dealer broadcasts the commitments to all parties, then he uses the zero-knowledge interactive proof scheme to convince all parties $P_1, \ldots, P_n$ that the commitments contain shares that are consistent with the secret. When the proof is accepted, the dealer sends S_i and C_i to each P_i.

2. **Reconstruction:** Not like a simple secret sharing scheme, reconstruction is not only done by the dealer. In this phase, each party P_i broadcast his share S_i and the opening information for C_i. Only the share, whose commitment is opened successfully, can be accepted by the honest parties to reconstruct the secret S.

In the general verifiable secret sharing protocol, the honest parties can detect the malicious dealer and abort the protocol. On the other hand, with a honest dealer, all honest parties are able to reconstruct the secret despite the actions of cheaters, while the malicious parties can not get any information.

Indeed, in the distribution phase, the dealer broadcasts commitments to all parties. Hence, each party acts as a verifier in a zero-knowledge proof scheme independently. Then they broadcast their proof results. In order to meet the security requirement as malicious parties can not figure out what the secret is, we suppose that the number of corrupted parties is less than t, where t is the threshold of the secret sharing scheme.

If the dealer is honest, all honest parties verify the consistency of shares. Even if all corrupted parties cheat, there are less than t parties complaining about the inconsistency of the shares. Hence the dealer is proved to be honest, and no honest party aborts the protocol. If the dealer is malicious, the honest parties can prove and report the inconsistency of shares except with negligible probability of error. While m ($m \geq t$) parties report the inconsistency, the honest parties accuse the dealer and abort the protocol.

In other words, if the dealer is honest, the distribution phase succeeds. If the corrupted parties are probabilistic polynomial time bounded, they can not get the content of commitments. Hence the commitment scheme, (t, n)-threshold secret sharing scheme and zero-knowledge proof guarantee the privacy of the secret.

In the reconstruction phase, assume that the shares are successfully distributed. If a party cheats, it can be easily detected while its commitment is not consistent with the share. Then the honest party may broadcast its complaint, the malicious party is pointed out when m parties complain. Then the false shares are ignored while reconstructing the secret.

To sum up, the only malicious action the corrupted parties can undertake is to abort the protocol in the reconstruction phase. As there are only less than t corrupted parties, where $t < \frac{n}{2}$ or $t < \frac{n}{3}$ in the presence of passive or active adversary, there are always t honest parties to reconstruct the secret S.

3.3.2 Feldman's Scheme

We briefly introduce Feldman's Scheme [14] as an example in this section. Feldman's scheme is based on Shamir's secret sharing scheme and any homomorphic encryption scheme. Let $\mathcal{Z}_p$ be a finite field and G a cyclic group of prime order p. The discrete logarithm is hard in G, and g is a generator of G.

Based on these assumption, the Feldman's scheme is as follows:

1. **Distribution:** The dealer uses Shamir's scheme to share a secret S. For the detailed process please refer to section 3.2. Then the dealer computes and broadcasts the following commitment

$$y_0 = g^S, y_1 = g^{f_1}, \ldots, y_{t-1} = g^{f_{t-1}}$$

   and sends the share S_i to party P_i.

2. **Verification:** When party P_i gets its share S_i, it verifies the consistency of its share by checking if the following equation holds

$$g^{S_i} = \prod_{j=0}^{t-1} y_j^{i^j} \ .$$

   If it holds, the share is accepted, else the party broadcasts a complaint.

In Feldman's scheme, no one can survive cheating, as the commitments $y_0, \ldots, y_{t-1}$ uniquely determine the polynomial $q(x)$. Hence, everyone can check if the share is consistent. However, this scheme is only secure for computationally bounded adversaries, and the public value $y_0 = g^S$ leaks some information about S.

3.4 Computation with Shamir's Shares

Assume that the Shamir's secret sharing scheme is used to compute shares of two secrets S and T as $S_1, \ldots, S_n$ and $T_1, \ldots, T_n$, and each party P_i gets shares S_i and T_i. We can observe that Shamir's scheme is homomorphic on addition and multiplication by a constant. The computation algorithms are as follows:

1. **Addition of Shares:** Each party P_i computes $R_i = S_i + T_i$ locally. Suppose R is the reconstructed value of secret $R_1, \ldots, R_n$, then $R = S + T$.

2. **Multiplication by a Constant:** Suppose there is a public constant c. Each party P_i computes $R_i = S_i * c$ locally. Suppose R is the reconstructed value of secret $R_1, \ldots, R_n$, then $R = S * c$.

3. **Multiplication of Shares:** As introduced in section 3.2, Shamir's scheme implements polynomials to compute the shares. Hence multiplication of shares is not a linear transformation any more. How to do the multiplication on Shamir's shares will be introduced in detail in the following part.

Suppose there are two polynomials, one with degree d, the other with degree e. After multiplying two polynomials, the degree of the resulting polynomial $d+e$. If $d+e > n$, it is not possible to reconstruct the secret. In order to solve this problem, re-sharing of shares is used to make multiplication possible.

Here we introduce the *Gennaro-Rabin-Rabin multiplication protocol* [15] to give a brief overview of how to multiply Shamir's shares. The protocol is as follows:

- **Computation:** Each party P_i computes $R'_i = S_i * T_i$.
- **Re-sharing:** Each party P_i secretly shares R'_i by using Shamir's scheme and gets shares $R'_{i_1}, \ldots, R'_{i_n}$. Then it sends share R'_{i_j} to each party P_j
- **Recombination:** There exist public reconstruction parameters $r_1, \ldots, r_n$, where

  $$r_i = \prod_{1 \le j \le n, j \ne i} \frac{j}{j-i} \ .$$

  Each party P_i then computes share

  $$R_i = \sum_{j=1}^{n} r_j * R'_{j_i} \ .$$

  Suppose R is the reconstructed value of the secrets $R_1, \ldots, R_n$, then $R = S * T$.

Theorem 3.4. *Addition of Shares algorithm and Multiplication by a Constant algorithm are correct.*

Proof. While sharing the secrets, the Shamir's secret sharing scheme computes shares by evaluating the polynomial with n different inputs. This evaluating process f is a linear transformation, which has the additivity and homogeneity properties.

Hence in the Addition of Shares algorithm, $f(R) = f(S+T) = f(S) + f(T)$. In Multiplication with a Constant algorithm, $f(R) = f(S * c) = c * f(S)$. So Addition of Shares algorithm and Multiplication with a Constant algorithm are correct. ☐

Theorem 3.5. *Gennaro-Rabin-Rabin multiplication protocol is correct, and it guarantees the privacy of the secret.*

Proof. In the re-sharing phase, each party P_i randomly generates a polynomial f_i of degree at most $t-1$, where $f_i(0) = S_i * T_i$. Then it sends the share $f_i(j)$ to party P_j. Hence, after the re-sharing phase, each party P_i gets shares $f_1(i), \ldots, f_n(i)$.

Then P_i computes

$$R_i = \sum_{j=1}^{n} \prod_{1 \leq i \leq n, i \neq j} \frac{i}{i-j} * f_j(i) \ .$$

According to theorem 3.1, ($R_1, 1, R_2, 2, \ldots, R_n, n$) uniquely determines the polynomial

$$f(x) = \sum_{i=1}^{n} r_i * f_i(x) \ .$$

If $x = 0$, we have

$$f(0) = \sum_{i=1}^{n} r_i * f_i(0) = \sum_{i=1}^{n} r_i * S_i * T_i = S * T = R \ .$$

We can observe that the polynomial shares the secret $S * T = R$, therefore this protocol works correctly.

On the re-sharing phase, as each party P_i implements the Shamir's scheme independently and separately, all the shares $R'_{1_1}, \ldots, R'_{1_n}, \ldots, R'_{n_1}, \ldots, R'_{n_n}$ are uniformly distributed and independent with each other. As a result of that, in the recombination phase, the R_i computed based on the values $R'_{1_i}, \ldots, R'_{n_i}$ is also uniformly distributed. Hence, no party can get more information than he has originally known. Since at least t of the values $r_1, \ldots, r_n$ are non-zero, according to the theorem 3.1, a unique (t, n)-threshold polynomial is determined, which keeps the correctness and privacy of the secret data. □

4 Sharemind Secure Computation Protocol

Based on what has been introduced in the former sections, we will make a detailed description of SHAREMIND multiparty computation protocols, which are proposed by Dan Bogdanov in his master thesis [4]. Although Dan Bogdanov gives a detailed introduction of all the protocols, he just presents a informal security proof for each protocol. In the informal proof, the messages generated and received by the corrupted parties are analyzed manually, and the proof results are written in human language. Even though this informal proof can be easily understood, it has the following two disadvantages:

1. It is not suitable for complex protocols, because there would be too many messages to be analyzed manually.

2. It is not convincing, as its validity is hard to be checked.

Hence, what we need to do is to figure out a method to automatically analyze these protocols in such a way that the correctness of the results can be easily checked. The method and result of analyzing SHAREMIND protocols are shown in section 5 and section 6. Note that throughout this paper, all parties are semi-honest, which follow the protocol while wondering what the secret is.

Before presenting the SHAREMIND secure computation protocols, information related to additive secret sharing scheme is shown in section 4.1. The infrastructure of SHAREMIND is introduced in section 4.2. The detailed descriptions of SHAREMIND secure computation protocols are shown in the following parts. As what we are interested in is the security proof of those protocols, for the correctness proof of SHAREMIND protocols, please refer to Bogdanov's thesis [4]. As this section is presented as a preliminary of the next two sections, only informal proofs are included here. For formal proofs, please refer to section 5.

4.1 Additive Secret Sharing

In SHAREMIND protocols, a n-out-of-n additive secret sharing scheme is used to share the secrets. The additive secret sharing scheme is very simple and efficient. There is no complex algorithm to compute shares, and no single share leaks any information about the secret. However, it is not very convenient, since the secret can only be reconstructed while all parties are presenting and combining their shares together.

The formal definition of this scheme is as follows:

1. **Distribution:** Assume there are n parties, $P_1, \ldots, P_n$, sharing the secret S. The dealer first generates $n-1$ uniformly distributed random values $S_1, \ldots, S_{n-1}$, then computes the last share as $S_n = S - S_1 - \ldots - S_{n-1}$. Each party P_i gets the share S_i.

2. **Reconstruction:** All parties send their shares to the dealer, the dealer reconstructs the secret by adding all shares together.

Theorem 4.1. *The n-out-of-n secret sharing scheme is a $(+,+)$-homomorphic secret sharing scheme.*

Proof. Suppose there are two secrets S and T. By using the n-out-of-n secret sharing scheme, we get the shares $S_1, \ldots, S_n$ and $T_1, \ldots, T_n$, where $S_1 + \ldots + S_n = S$ and $T_1 + \ldots + T_n = T$. After each party P_i gets its shares, it computes $R_i = S_i + T_i$ locally. Hence $R = R_1 + \ldots + R_n = S_1 + T_1 + \ldots + S_n + T_n = S + T$. $\square$

Theorem 4.2. *According to definition 3.3, while multiplying the secret by a constant, the n-out-of-n secret sharing scheme is a $(\times, \times)$-homomorphic secret sharing scheme.*

Proof. Assume the n-out-of-n secret sharing scheme distributes secret S into shares $S_1, \ldots, S_n$, where $S_1 + \ldots + S_n = S$. There is a public constant c. After each party P_i gets its share, it computes $R_i = S_i \times c$ locally. Hence $R = R_1 + \ldots + R_n = S_1 \times c + \ldots + S_n \times c = S \times c$. $\square$

However, while multiplying the shares of two secrets, the n-out-of-n secret sharing scheme is not $(\times, \times)$-homomorphic any more. Hence, for this more complex scenario, we will discuss it in detail in the following parts of this chapter.

4.2 Introduction of SHAREMIND

In 2007, Dan Bogdanov has proposed a framework for secure computations in his master thesis [4]. As shown in Figure 2, in SHAREMIND, there are several controllers and miners. The number of controllers is not restricted, each of them provides data to and gets analyze results from miners. In the latest version of SHAREMIND, there are only one controller and three miners. The miners work together to perform computations and run data mining algorithms on the data. It is assumed that each miner is semi-honest, and they perform the computation synchronously. The number of miners is extensible based on the number of corrupted miners. Suppose the adversary can corrupt n miners, there should be at least $2n + 1$ miners to keep the privacy of data.

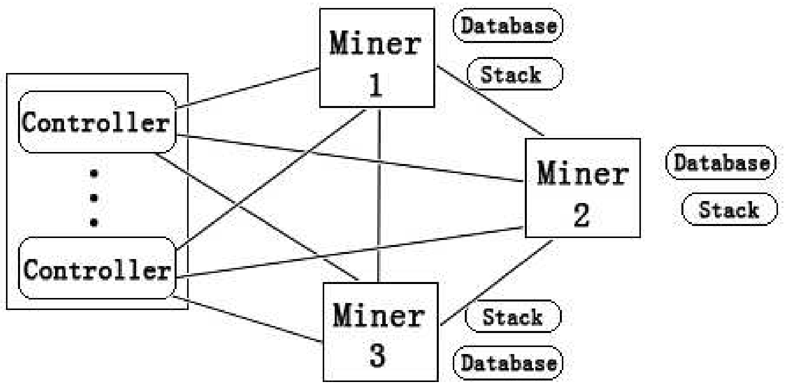

Figure 2: The infrastructure of SHAREMIND

We can observe that there are private communication channels between each pair of miners, and the communication channels exist between each controller and all miners. As the controllers work as data providers, we suppose that they are concerned with their privacy. Hence, there is no need to provide communication channels between controllers.

Each miner has a local database for saving shares and a stack for saving intermediate messages. Once a controller inputs a data, it would compute the shares of it by using the additive secret sharing scheme and distribute the shares to miners respectively. Then the miners would check the validity of shares and save them in their local database. It is required that all shares saved in databases and stacks should be in $\mathcal{Z}_{2^{32}}$, and the Share Conversion protocol, which will be introduced in section 4.4, is proposed to convert the shares in $\mathcal{Z}_2$ into $\mathcal{Z}_{2^{32}}$.

As each miner is only capable of running a number of basic operations, the SHAREMIND secure computation protocols are proposed to direct miners to get correct results by following complex algorithms built from these basic operations.

4.3 Multiplication Protocol of Three Parties

Assume Alice, Bob and Charlie are three participants of a multiparty computation protocol. We denote Alice as A, Bob as B and Charlie as C. There are two secrets u and v. By using the 3-out-of-3 additive secret sharing scheme, u and v are shared as u_A, u_B, u_C and v_A, v_B, v_C, which are held by Alice, Bob and Charlie, respectively. The messages exchanged between parties are denoted as $name_{xy}$, where $name$ is the name of the message, x is the index of the party who sends the message, and y is the index of the party who gets it, $x, y \in \{1, 2, 3\}$ and $x \neq y$.

As a semi-honest adversary, the adversary can only corrupt at most one party. Once a party is corrupted, the adversary can get its shares, outputs and all internal data such as the random numbers generated by that party, and the messages sent to or received by it. The adversary always tries to get some information about the secret through what it gets. Figure 3 shows the model of three party computation.

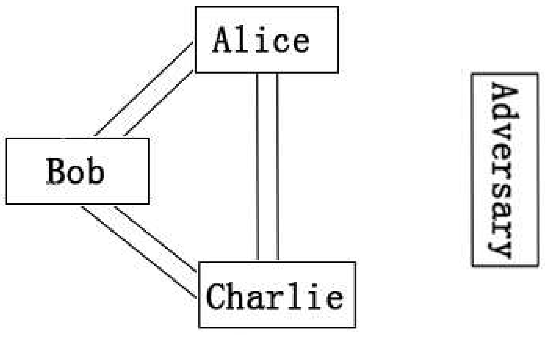

Figure 3: Communications Between Three Parties in the Real World

As shown in section 4.1, addition of shares and multiplying shares by a constant can be easily executed. Hence, we omit them here. In the following parts, we will discuss the multiplication of two secrets in detail.

What we want to compute is

$$\begin{aligned} u * v =& (u_A + u_B + u_C) * (v_A + v_B + v_C) \\ =& u_A * v_A + u_A * v_B + u_A * v_C + u_B * v_A + u_B * v_B + u_B * v_C + u_C * v_A + \\ & u_C * v_B + u_C * v_C \end{aligned}$$

However, each party just knows its own shares. The problem we need to solve is how to compute $u_i v_j$ in such a way that none of them can figure out another party's secret, where $i, j \in \{A, B, C\}$ and $i \neq j$?

The protocol proposed by Du and Atallah [12] is the right answer to the question. It assumes that two parties P_1 and P_2 want to secretly multiply two values x_1 and x_2. There is a third party P_3, who works as a random number distributor. Let us denote the multiplication result as S, and its shares are denoted as S_1, S_2 and S_3. Hence, they follow the following protocol 1:

Protocol 1 Du and Atallah Protocol

```
**Input:** $P_1$: $x_1$, $P_2$: $x_2 \in \mathcal{Z}_{2^{32}}$
**Output:** $P_1$: $S_1$, $P_2$: $S_2$ and $P_3$: $S_3$

Round One
    $P_3$ generates two uniformly distributed random numbers $a_1$ and $a_2 \in \mathcal{Z}_{2^{32}}$
    $P_3$ sends $a_1$ to $P_1$ and $a_2$ to $P_2$

Round Two
    $P_1$ computes $x_1 + a_1$ and sends the result to $P_2$
    $P_2$ computes $x_2 + a_2$ and sends the result to $P_1$

Round Three
    $P_1$ computes $S_1 = -(x_1 + a_1) * (x_2 + a_2) + x_1 * (x_2 + a_2)$
    $P_2$ computes $S_2 = x_2 * (x_1 + a_1)$
    $P_3$ computes $S_3 = a_1 * a_2$
```

Theorem 4.3. *In Du and Atallah protocol, the privacy of secret data is guaranteed.*

Proof. In the first round, P_1 receives a uniformly distributed number a_1, and P_2 receives a uniformly distributed number a_2. Hence, both of P_1 and P_2 can not figure out other party's secret.

In the second round, P_1 gets $x_2 + a_2$. As P_1 does not know the value of uniformly distributed number a_2, $x_2 + a_2$ is uniformly distributed to him. For the same reason, $x_1 + a_1$ is uniformly distributed to P_2. P_3 has no incoming messages. Consequently, all three parties can not figure out other parties' secrets. □

As mentioned above, Du and Atallah protocol is what we need to securely compute $u_i v_j$, where $i, j \in \{A, B, C\}$ and $i \neq j$. While composing three party share multiplication protocol, we import Du and Atallah protocol as a sub-protocol in the form $Subprotocol$: $A : ShareName1$, $B : ShareName2$, $C : ShareName3 = DuAtallah(Name1 : (Input1), Name2 : (Input2))$, which means that the party with name $Name1$ and the party with name $Name2$ import a instance of Du and Atallah protocol to compute $Input1 * Input2$. After running the instance of sub-protocol, Alice gets result share $ShareName1$, Bob gets $ShareName2$ and Charlie gets $ShareName3$. Following the lines of Dan Bogdanov's master thesis [4], the protocol of three party multiplication we mentioned above is as follows:

Protocol 2 Three Party Share Multiplication Protocol

```
Input: Alice: $u_A$, $V_A$; Bob: $u_B$, $V_B$; Charlie: $u_C$, $V_C \in \mathcal{Z}_{2^{32}}$
Output: Alice: $d_A$, Bob: $d_B$, Charlie: $d_C \in \mathcal{Z}_{2^{32}}$

Round One
    Alice computes $s_A = u_A * v_A$ locally
    Bob computes $s_B = u_B * v_B$ locally
    Charlie computes $s_C = u_C * v_C$ locally

Round Two
    Assume that all random numbers are uniformly distributed and independent
    Subprotocol: $A : a_{12}, B : b_{12}, C : c_{12}$ = DuAtallah$(A : (u_A), B : (v_B))$
    Subprotocol: $A : a_{13}, B : b_{13}, C : c_{13}$ = DuAtallah$(A : (u_A), C : (v_C))$
    Subprotocol: $A : a_{21}, B : b_{21}, C : c_{21}$ = DuAtallah$(B : (u_B), A : (v_A))$
    Subprotocol: $A : a_{23}, B : b_{23}, C : c_{23}$ = DuAtallah$(B : (u_B), C : (v_C))$
    Subprotocol: $A : a_{31}, B : b_{31}, C : c_{31}$ = DuAtallah$(C : (u_C), A : (v_A))$
    Subprotocol: $A : a_{32}, B : b_{32}, C : c_{32}$ = DuAtallah$(C : (u_C), B : (v_B))$

Round Three
    Alice computes $d_A = s_A + a_{12} + a_{13} + a_{21} + a_{23} + a_{31} + a_{32}$.
    Bob computes $d_B = s_B + b_{12} + b_{13} + b_{21} + b_{23} + b_{31} + b_{32}$.
    Charlie computes $d_C = s_C + c_{12} + c_{13} + c_{21} + c_{23} + c_{31} + c_{32}$.
```

Theorem 4.4. *In the three party share multiplication protocol, the privacy of secret data is guaranteed.*

Proof. We start by proving that except for u_A, v_A, Alice does not know the secrets of other parties. As this protocol is symmetrical, if we prove that in each round the messages received by Alice are all independent and uniformly distributed, so are the messages received by Bob and Charlie.

In round one, Alice has not received any messages. In round two, it is assumed that every random number is uniformly distributed, therefore in each call of Du and Atallah protocol, new random numbers are generated. According to Theorem 4.5, shares

$a_{12}, a_{13}, a_{21}, a_{23}, a_{31}$ and a_{32} are uniformly distributed and independent of each other. As a consequence of that, the incoming messages of Alice are all uniformly distributed and independent. Hence, the privacy of secret data is guaranteed. $\square$

4.4 SHAREMIND Share Conversion and Bit Extraction Protocols

Besides the binary operations on secrets we mentioned above, the operations on bits are also very important. In this section, we will introduce Share Conversion protocol and Bit Extraction protocol. The former one is used to convert a bit share of value in range $\mathcal{Z}_2$ into range $\mathcal{Z}_{2^{32}}$, and the later one is in charge of extract the shares of the bits of a share in range $\mathcal{Z}_{2^{32}}$.

As a data donor, each controller in SHAREMIND infrastructure can input boolean numbers in $\mathcal{Z}_2$ as the secrets. By using the additive secret sharing scheme, the shares each miner gets are in $\mathcal{Z}_2$ too. As we mentioned above, the valid data must be in $\mathcal{Z}_{2^{32}}$. Hence, we must implement a protocol to convert the secrets in $\mathcal{Z}_2$ to be uniform in $\mathcal{Z}_{2^{32}}$ while keeping the privacy of these secrets.

Assume there is a boolean number u, and u_A, u_B, $u_C \in \mathcal{Z}_2$ are computed from it by using the additive secret sharing scheme. We can use the following equation to convert the shares into $\mathcal{Z}_{2^{32}}$:

$$f(u_A, u_B, u_C) = u_A + u_B + u_C - 2u_Au_B - 2u_Au_C - 2u_Bu_C + 4u_Au_Bu_C \ .$$

Hence, we can use Du and Atallah protocol to compute shares of u_Au_B, u_Bu_C and u_Au_C. On the other hand, we can let Charlie additively shares u_C into three shares, then run the Three Party Share Multiplication Protocol to get the shares of $u_Au_Bu_C$. To sum up, the Share Conversion Protocol is as follows:

Protocol 3 Share Conversion Protocol

```
Input: Alice: $u_A$, Bob: $u_B$, Charlie: $u_C \in \mathcal{Z}_2$
Output: Alice: $d_A$, Bob: $d_B$, Charlie: $d_C \in \mathcal{Z}_{2^{32}}$

Round One
    Charlie generate two uniformly distributed numbers $c_{31}, c_{32} \in \mathcal{Z}_{2^{32}}$
    Charlie computes the third share of $u_C$ as $c_{33} = u_C - c_{31} - c_{32}$
    Charlie sends $c_{31}$ Alice and $c_{32}$ to Bob

Round Two
    Assume that all random numbers are uniformly distributed and independent in $\mathcal{Z}_{2^{32}}$
    Subprotocol: $A : x_A, B : x_B, C : x_C$ = DuAtallah($A : (u_A), B : (u_B)$)
    Subprotocol: $A : y_A, B : y_B, C : y_C$ = DuAtallah($C : (u_C), A : (u_A)$)
    Subprotocol: $A : z_A, B : z_B, C : z_C$ = DuAtallah($B : (u_B), C : (u_C)$)
    Subprotocol: $A : w_A, B : w_B, C : w_C$ = Multiplication($A : (x_A, c_{31}), B : (x_B, c_{32}), C : (x_C, c_{33})$)

Round Four
    Alice computes its share $d_A = u_A - x_A - x_A - y_A - y_A - z_A - z_A + w_A + w_A + w_A + w_A$
    Bob computes its share $d_B = u_B - x_B - x_B - y_B - y_B - z_B - z_B + w_B + w_B + w_B + w_B$
    Charlie computes its share $d_C = u_C - x_C - x_C - y_C - y_C - z_C - z_C + w_C + w_C + w_C + w_C$
```

Theorem 4.5. *In the Share Conversion protocol, no party can figure out the value of other parties' secret.*

Proof. This protocol is not symmetric to all parties, as Charlie has to implement additive secret sharing scheme to share its share u_C into three sub-shares. However, as Charlie does not receive as many messages as the other two parties, it is trivial that Charlie's view is secure if we prove that Alice and Bob's views are secure.

For Alice, in round one, what Alice get is a uniformly distributed number, hence no secret is compromised. In round two, three Du and Atallah protocols and one Three Party Share Multiplication Protocol are called. According to theorem 4.3 and 4.4, the privacy of secrets are kept, hence no other secrets are leaked to Alice. In the last round, Alice gets no new messages. To sum up, Alice can not figure out the secret of other parties.

For the same reason, Bob also can not get secrets he should not know. Consequently, we can say that the privacy of secrets is guaranteed in the Share Conversion protocol. □

Although Share Conversion protocol is very useful to convert $\mathcal{Z}_2$ shares into $\mathcal{Z}_{2^{32}}$ ones, it can not handle all the bitwise operations. Hence, we need the Bit Extraction protocol as the basics of complex bitwise operations. As shown in Protocol 4, the Bit Extraction protocol can extract shares of each bit of a share u in $\mathcal{Z}_{2^{32}}$, where $u_A^{(0)}, \ldots, u_A^{(31)}$, $u_B^{(0)}, \ldots, u_B^{(31)}$ and $u_C^{(0)}, \ldots, u_C^{(31)}$ denote the shares of the 1st to 32nd bit of the share u.

Protocol 4 Bit Extraction Protocol

```
Input: Alice: $u_A$, Bob: $u_B$, Charlie: $u_C \in \mathcal{Z}_{2^{32}}$
Output: Alice: $u_A^{(0)}, \ldots, u_A^{(31)}$; Bob: $u_B^{(0)}, \ldots, u_B^{(31)}$; Charlie: $u_C^{(0)}, \ldots, u_C^{(31)} \in \mathcal{Z}_{2^{32}}$

Round One
    Alice generates uniformly distributed numbers $r_A^{(0)}, \ldots, r_A^{(31)} \in \mathcal{Z}_2$.
    Alice converts all these numbers into $\mathcal{Z}_{2^{32}}$ by using the Share Conversion protocol, and the result shares are denoted as $t_A^{(0)}, \ldots, t_A^{(31)}$.
    Bob generates uniformly distributed numbers $r_B^{(0)}, \ldots, r_B^{(31)} \in \mathcal{Z}_2$.
    Bob converts all these numbers into $\mathcal{Z}_{2^{32}}$ by using the Share Conversion protocol, and the result shares are denoted as $t_B^{(0)}, \ldots, t_B^{(31)}$.
    Charlie generates uniformly distributed numbers $r_C^{(0)}, \ldots, r_C^{(31)} \in \mathcal{Z}_2$.
    Charlie converts all these numbers into $\mathcal{Z}_{2^{32}}$ by using the Share Conversion protocol, and the result shares are denoted as $t_C^{(0)}, \ldots, t_C^{(31)}$

Round Two
    Alice computes $r_A = \prod_{j=0}^{31} 2^j t_A^{(j)}$.
    Alice computes $v_{11} = u_A - r_A$.
    Bob computes $r_B = \prod_{j=0}^{31} 2^j t_B^{(j)}$.
    Bob computes $v_{21} = u_B - r_B$, and sends $v_{21}$ to Alice.
    Charlie computes $r_C = \prod_{j=0}^{31} 2^j t_C^{(j)}$.
    Charlie computes $v_{31} = u_C - r_C$, and sends $v_{31}$ to Alice.

Round Three
    Alice computes $a_A = v_{11} + v_{21} + v_{31}$ and finds its bits $a_A^{(0)}, \ldots, a_A^{(31)}$, and output $u_A^{(i)} = a_A^{(i)} + t_A^{(i)}$ for $i \in \{0, \ldots, 31\}$.
    Bob outputs $u_B^{(i)} = t_B^{(i)}$ for $i \in \{0, \ldots, 31\}$.
    Charlie outputs $u_C^{(i)} = t_C^{(i)}$ for $i \in \{0, \ldots, 31\}$.
```

Theorem 4.6. *The Bit Extraction protocol is perfectly secure.*

Proof. In round one, each party generates 32 uniformly distributed numbers in $\mathcal{Z}_2$, and converts them by using the Share Conversion protocol. According to Theorem 4.5, no party can figure out other parties' secret while converting shares, hence no secrets are exposed in this phase, and all shares in $\mathcal{Z}_{2^{32}}$ are uniformly distributed and independent from each other.

In round two, as the values of shares r_A, r_B and r_C are computed on uniformly distributed and independent random numbers, they are uniformly distributed and independent from each other. As a result of that, the values of v_{11}, v_{21} and v_{31}, which are computed by subtracting a uniformly distributed numbers from a secret, are also uniformly distributed. Hence, although Alice gets v_{11}, v_{21} and v_{31}, what she can get is only the difference between the randomly generated number r and the secret u. In a word, Alice gets no more secret. As Bob and Charlie get no information in this round, their view is secure.

In round three, there are no messages exchanged between parties. To sum up, in this Bit Extraction protocol, no party is capable of getting other parties' secret, hence it is perfectly secure. □

5 Simulation

As discussed in section 2.3, a protocol π is perfectly secure if for any adversary attacking π in the real world, the output distribution of the real world is indistinguishable from the output distribution of the ideal world. In other words, if the protocol π is perfectly secure, then the environment is not able to tell the real world model and the ideal world model apart. To achieve this requirement, we can use a simulator to translate the messages sent and received by the real world adversary to the ideal world model, while keeping the real world adversary thinking that he works in the real world.

In the following part, we will make an elaborate discuss on the approach of simulation. The notion of universal composition is presented in section 5.1 as a preliminary. In section 5.2, we will show how to simulate the real world. After that, we will present the formal proof of the protocols shown in the former chapter by importing the notion of simulation.

5.1 Universal Composition

In 2001, Ran Canetti [7] proposed a framework to represent and analyze the security of cryptographic protocols. This framework is called universal composable security framework. Within this framework, a method called universal composition, which composes protocols in such a way that the security is preserved while composing, is defined. The definitions of security in this framework have the property of universal composablility. Another definition of universal composability was presented by Ronald Cramer and Ivan Damgård [11], which covers both information theoretic and cryptographic models in synchronous communication model. The following discussion is based on their works.

First, let us have a look at the general universal composable security framework in Figure 4, which is similar to the structure in the Ideal and Real world approach introduced in section 2.3.

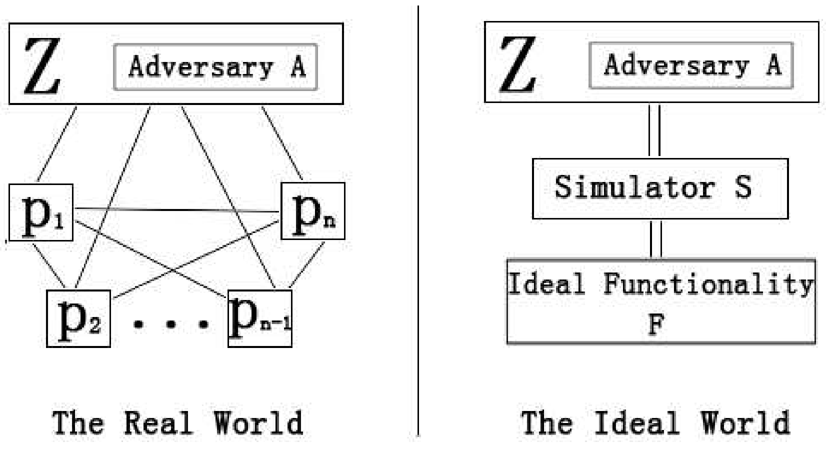

Figure 4: General Universal Composable Security Framework

Assume there is an environment Z, which contains everything that is external to the protocol execution, such as the adversaries and users who supply inputs to the protocol. Since adversaries are included, Z is capable of doing anything the adversaries can do. In other words, Z can actively (or passively) and adaptively (or non-adaptively) corrupt parties, which are constrained by the adversary structure $\mathcal{A}$.

In the real world, besides the environment Z, there are n parties denoted as $P_1, \dots, P_n$. which are modeled as interactive Turing Machines. All parties communicate with each other in a synchronous communication model. While executing the protocol π, the adversary in the environment Z corrupts a set of parties according to the adversary structure $\mathcal{A}$. Meanwhile, all honest parties follow the protocol π. Hence, the environment Z interacts twice with the honest parties in each round, it first chooses random inputs and sends them to each party, then it collects the outputs from the parties.

In the ideal world, there is an ideal functionality F and a simulator S. As introduced in section 2.3, F is incorruptible and it provides perfect security. Since parties do not communicate with each other directly, and all data are collected and sent out by the ideal functionality F, we suppose that there are no parties in the ideal world. Instead, we set a simulator S, which is in charge of translating the traffics between the environment Z and the ideal functionality F.

After the computation process is finished, Z outputs a single bit to indicate whether Z thinks itself has interacted with the protocol π or the ideal functionality F. This bit is a random variable, whose distribution depends on the protocol π and the environment Z in the real world, and the ideal function F, simulator S and environment Z in the ideal world.

We can say that a protocol π securely realizes an ideal functionality F, if there is a polynomial time simulator S such that for any environment Z, Z can not distinguish the output of protocol π from the output of ideal functionality F.

Hence, requiring that the outputs of corrupted parties in real world and ideal world are distributed indistinguishably from each other grantees data secrecy. It forces the information gathered by the real world adversary to be computable in the ideal process, hence the protocol does not release more information to corrupted parties than it should. On the other hand, correctness of results is guaranteed by requiring that the outputs of honest parties in real world are distributed similarly to the outputs of honest party in ideal world. Since if the real world adversary has more influence on the outputs than the ideal world adversary, the environment Z can easily tell the real world and the ideal world apart.

Before presenting the universal composition theorem, let us have a look at a special model, which is a hybrid between ideal world model and real world model. It is called $\mathcal{G}$-hybrid model, which includes subroutine calls to the ideal functionality $\mathcal{G}$. This model can invoke several instances of $\mathcal{G}$, and these instances may run concurrently. Let π be a real world protocol that uses subroutine calls to the ideal functionality $\mathcal{G}$. Hence as described in the ideal world, each honest party P_i communicates with ideal functionality $\mathcal{G}$ through ports I_i and O_i, and the adversary represents the corrupted parties to communicate with $\mathcal{G}$ through ports I_A and O_A.

Now let ρ be a protocol that securely realizes an ideal functionality $\mathcal{G}$. Therefore, we can compose a protocol $\pi^{\rho/\mathcal{G}}$, which means that the protocol π calls protocol ρ instead of the ideal functionality $\mathcal{G}$. In that case, the protocol π is modified to send all inputs provided for $\mathcal{G}$ to protocol ρ, and to treat outputs received from protocol ρ as outputs received from

$\mathcal{G}$. Hence, we can give the following definition:

Definition 5.1. *The composed protocol $\pi^{\rho/\mathcal{G}}$ universal composably emulates protocol π, if π is a $\mathcal{G}$-hybrid protocol and protocol ρ securely realizes the ideal functionality $\mathcal{G}$.*

Assume that protocol π securely realizes an ideal functionality F, hence the following definition can be generalized:

Definition 5.2. *The composed protocol $\pi^{\rho/\mathcal{G}}$ securely realizes an ideal functionality F, if π is a $\mathcal{G}$-hybrid protocol, which securely realizes F, and protocol ρ securely emulates ideal functionality $\mathcal{G}$*

5.2 Simulating the real world

As mentioned above, a protocol π is considered as perfectly secure if the environment Z can not distinguish the outputs of using the protocol π from the outputs of using an ideal functionality F. Therefore, we require that the messages the environment Z gets in the real world have the same distribution as messages it gets in the ideal world. In other words, Z can act in the same way no matter if it is in the real world or in the ideal world, and the distribution of the outputs he gets form the two worlds are indistinguishable from each other. One way to achieve this requirement is to simulate the real world on the ideal world.

As described in the former section, in the ideal world, there is a simulator S that translates messages between the environment Z and the ideal functionality F. The function of the simulator S is to communicate with environment Z in such a way that Z can not find out that it is in the ideal world. Therefore, while communicating with Z, S needs to provide messages in the form that is exactly the same as what Z should see. Hence, S must know the protocol in advance, then it can go through the protocol and send the valid messages to Z in the right order and right time.

However, before we can do the simulation, we need to know how environment Z acts. We can observe that there are two kinds of activities of Z. First, Z provides inputs to honest parties, and gets output from them. This action is easy to handle, what we need to do is to send inputs and outputs to the relevant input and output ports of the ideal functionality F. Second, as Z includes the adversary, it can corrupt parties and see all their internal data, which contain the inputs, outputs and messages they send and receive. Hence, the simulator S is in charge of communicating with Z on behalf of the corrupted parties.

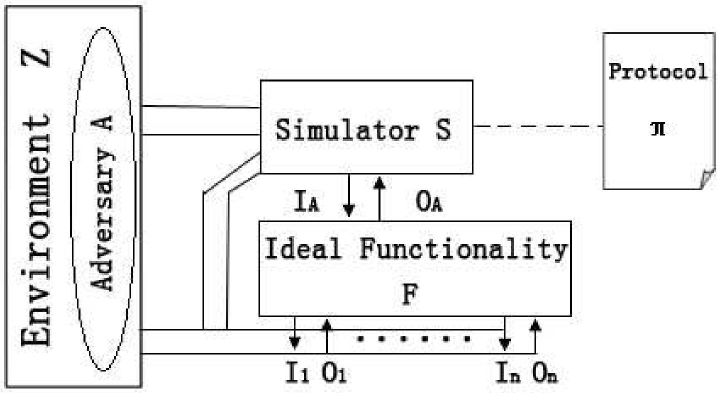

Figure 5: Simulating the Real World

As shown in Figure 5, what the simulator does is representing the corrupted parties to use the corrupt input and output ports of the ideal functionality F to provide inputs to and get outputs from F. In other words, in each round, S relays the inputs of corrupted parties chosen by Z to F, and after the computation, S sends the outputs of corrupted parties generated by F to Z. While Z requests to corrupt party P_i, it notifies both S and F. Then F sends all inputs and outputs of P_i to S through port O_i. After that, S represents the corrupted party P_i to exchange messages with F on corrupted input and output ports.

5.3 Formal Proof of SHAREMIND Protocols

As mentioned above, if we can simulate the real world on the ideal world so that the real world adversary can not distinguish which world it is in, the real world protocol π is proved to be secure. Hence, while simulating a real world protocol, we need to do the following things:

1. First, construct the real world and define the power of the real world adversary A.
2. Then, define the ideal functionality F in the ideal world.
3. At last, build a simulator which communicates with both F and A in such a way that each message required by A is computable in F.

Based on this approach, we will do simulation on the SHAREMIND protocols and prove their security properties by showing that they are perfectly simulatable.

As shown in Protocol 1, there are three parties P_1, P_2 and P_3 participate in Du and Atallah protocol, two parties P_1 and P_2 want to compute the multiplication value S of their inputs x_1 and x_2. Let us denote this three party function as $F(x_1, x_2, null) = (S_1, S_2, S_3)$, where $S = S_1 + S_2 + S_3$. It is supposed that all parties are semi-honest, and the adversary can only corrupt one party. Since this protocol is not symmetric to every party, in the following part, we will discuss the simulation on each party separately.

As the simulator must know the Du and Atallah protocol in advance and follow the protocol while communicating with the real world adversary A, we rewrite Protocol 1 to the following form, which is easier to follow while processing computations:

Protocol 5 Formal Du and Atallah Protocol

```
Input: $P_1$: $x_1$, $P_2$: $x_2$
Output: $P_1$: $S_1$, $P_2$: $S_2$ and $P_3$: $S_3$

Round One
    $P_3$: Random($a_1, a_2$);
    $P_3 \rightarrow P_1$: $a_1$;
    $P_3 \rightarrow P_2$: $a_2$;

Round Two
    $P_1$: $r_{12} = x_1 + a_1$;
    $P_1 \rightarrow P_2$: $r_{12}$;
    $P_2$: $r_{21} = x_2 + a_2$;
    $P_2 \rightarrow P_1$: $r_{21}$;

Round Three
    $P_1$: $S_1 = -(x_1 + a_1) * (x_2 + a_2) + x_1 * (x_2 + a_2)$;
    $P_2$: $S_2 = x_2 * (x_1 + a_1)$;
    $P_3$: $S_3 = a_1 * a_2$;
```

In this formal protocol, we can see that there are three types of actions: First, generating random items, which is denoted as "Random(. . .)", the items inside the parentheses are the random items to be generated. Second, sending item from one party to another, which is denoted as "Sender $\rightarrow$ Receiver", and the item after the ":" is the item to be sent. Third, computing the values, which is denoted as mathematical expressions.

Assume that P_1 is corrupted by the real world adversary A, therefore, A knows all the internal data of P_1. While executing the protocol, P_1 gets the following messages from other parties: a_1 and r_{21}. It is trivial that based on what P_1 knows, it can not compute the secret value x_2. Hence, in P_1's view, each message is random and independent from each other. As a result of that, while communicating with A, the simulator just needs to generate a random item and send it to A at each time A needs to get a message. Therefore, we can do the simulation as follows:

Protocol 6 Simulating Du and Atallah Protocol while P_1 is corrupted

```
Input: $P_1$: $x_1$, $P_2$: $x_2$
Output: $P_1$: $S_1$

Round One
    $Simulator$: Random($a_1$);
    $Simulator \rightarrow P_1$: $a_1$;

Round Two
    $P_1$: $r_{12}$ =$x_1 + a_1$;
    $P_1 \rightarrow Simulator$: $r_{12}$;
    $Simulator$: Random($r_{21}$);
    $Simulator \rightarrow P_1$: $r_{21}$;

Round Three
    $P_1$: $S_1 = -(x_1 + a_1) * r_{21} + x_1 * r_{21}$;
```

Theorem 5.1. *The Du and Atallah protocol is perfectly secure.*

Proof. By observing Protocol 6 and Protocol 5, we can conclude that the incoming messages of P_1 from Protocol 6 and incoming messages of P_1 from Protocol 5 are indistinguishable. As the ideal functionality F is incorruptible, any computation can be executed securely. Since the same function is evaluated in both real world and ideal world, the output of honest parties should be the same. We can say that while P_1 is corrupted, Du and Atallah protocol is perfectly secure.

The simulation is similar while P_2 is corrupted, hence we can infer that Du and Atallah protocol is perfectly secure while P_2 is corrupted. As P_3 just works as a random item distributor, it gets no incoming messages, thus it is trivial that the protocol is perfectly secure if A corrupts P_3. To sum up, we can conclude that Du and Atallah Protocol is perfectly secure under the condition that the semi-honest adversary A can only corrupt one party. □

Theorem 5.2. *The Three Party Share Multiplication protocol is perfectly secure.*

Proof. By obseeving the Three Party Share Multiplication protocol shown in Protocol 2, we can get that the messages are exchanged only in round two. Precisely speaking, the messages are exchanged in the six instances of Du and Atallah protocols, which are imported by Three Party Share Multiplication protocol as sub-protocols. As shown in the proof of Theorem 5.1, each party of the Du and Atallah protocol is simulatable. It is required that all random numbers in all instances of Du and Atallah protocols are uniformly distributed and independent from each other, so the output shares of each instance are uniformly distributed and independent from each other.

Hence, while simulating the Three Party Share Multiplication protocol, we just need to construct a simulator which takes the simulator of each instance of Du and Atallah protocol as its component. Assume that the Du and Atallah protocol securely realizes an ideal functionality F. Let us denote the ideal functionality of three party multiplying additive shares as T, which contains six instances of F. According to the theory of universal composition, we can get that the Three Party Share Multiplication protocol universal composably emulates T. Hence, the Three Party Share Multiplication protocol is perfectly secure. $\square$

Theorem 5.3. *The Share Conversion protocol is perfectly secure.*

Proof. As shown in Protocol 3, in the first round, what Alice and Bob get are uniformly distributed random numbers. In the second round, three instance of Du and Atallah protocols and one instance of Three Party Share Multiplication protocol are imported.

Hence, while simulating the Share Conversion protocol, we just need to follow the protocol and import the simulation on the sub protocols in the right time. Hence, each party is simulatable and none of them can figure out more secrets. On the other hand, the output distribution of honest parties form the ideal world and real world are indistinguishable. To sum up, the Share Conversion Protocol is perfectly secure. $\square$

Theorem 5.4. *The Bit Extraction protocol is perfectly secure.*

Proof. According to the Protocol 4, as the Share Conversion protocol is perfectly secure, while simulating the round one of Bit Extraction protocol, we just need to import the simulator of each instance of Share Conversion protocols.

In the round two, Alice gets two messages v_{21} and v_{31}, and other two parties have no incoming messages. Hence, the simulation on Bob and Charlie is done in round one, and their views are perfectly secure. To Alice, the incoming messages v_{21} and v_{31} are uniformly distributed random numbers, since they are computed by subtracting a uniformly distributed random number from a secret. Therefore, while simulating Alice in round two, the simulator just needs to follow the protocol to generate and send the random numbers at the right time. So Alice is not able to figure out more secrets.

According to the theorem of universal composition, since the Share Conversion protocol is perfectly secure, the Bit Extraction protocol is perfectly secure. $\square$

6 Overview of Our Implementation

Based on the theory of universal composition, which was given in the former chapter, we have built an implementation called AutoProver to analyze the SHAREMIND protocols. The name of the implementation shows its functionality as the automatic prover of the security of secret shared protocols.

Our implementation is a software program, which can be executed on personal computers. The software can be divided into three components. In the following parts, the infrastructure of the software is introduced in section 6.1, and the introduction of each component is shown in section 6.3 to 6.6.

As a new grammar is defined in our implementation, all the SHAREMIND protocols described above must be rewritten in our grammar manually. Only the valid protocols can be analyzed by our program. Besides the SHAREMIND protocols, the AutoProver also can analyze other valid protocols. The grammar of the protocols is introduced in section 6.3.

6.1 AutoProver Infrastructure

The infrastructure of our implementation is shown in Figure 6. There are three components, the protocol parser, the security analyzer and the simulation generator. The protocol parser parses the input protocols and checks if there is any grammar error. The security analyzer is in charge of analyzing the valid protocols to see if the corrupted parties, which are designated by the user input, can figure out more secrets than they should know. The simulation generator can generate the simulation results based on the analysis results.

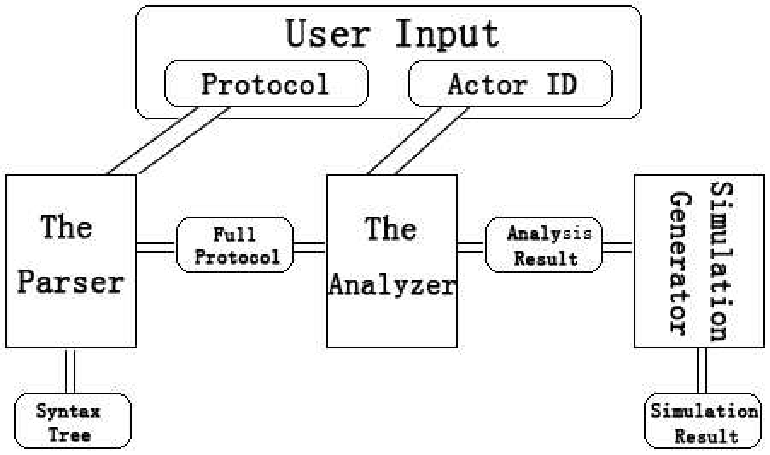

Figure 6: Infrastructure of AutoProver

The approach of processing the input protocol based on the designated corrupted actors is as follows:

1. First, the parser will parse the input protocol and check the grammar of the protocol. If the input protocol imports other protocols as its sub-protocols, check the grammar of the sub-protocols, too. If any grammar error is detected, the program halts and reports the parsing error on screen.

2. If there is no grammar error of the protocol and its sub-protocols, the parser would insert the content of the instances of sub-protocols into the input protocol. Then it will output the full protocol and the syntax tree of the full protocol.

3. The analyzer takes the full protocol and the corrupted actors' IDs as its input. Then it analyzes if the corrupted actors can figure out more secrets than they should. After finishing analyzing, the analyzer outputs the analysis report.

4. If the full protocol is analyzed to be secure, then the simulation generator takes the analysis report as its input and does simulation based on it.

6.2 Implementation Notes

Our implementation is programmed in Java language. Hence, our implementation is platform independent. Since our goal is to build a tool, which can automatically evaluate the security of secret shared protocols, there is no need to use computer networks. Therefore, our implementation works on single computer.

The Java Compiler Compiler (JavaCC), which is a parser generator, is used as an important tool to generate our parser code. JavaCC can read the grammar specification and convert it to a Java parser program. The parser program can recognize the content that matches the grammar. JavaCC also provides sub-tools related to the parser generation, such as JJTree, which is in charge of tree building, and JJDoc, which can produce documentation for the BNF grammar. In our implementation, we use JavaCC version 4.0. It works with any Java virtual machine, version 1.2 or greater.

The inputs and outputs of our implementation are controlled in the following approach. We use the *FileInputStream* class in *java.io* package to obtain the inputs, and the *PrintWriter* class to print the outputs. Both the inputs and outputs are stored locally.

Our implementation can automatically generate the simulation results for the protocols provided by us in advance. However, there is a restriction on the current version of our implementation. Because of the disadvantage of the grammar and the variable renaming algorithm in the parser code, our implementation can only deal with the protocols with up to five actors, and it can only generate the full protocol with less than 2025 variables. But it can be extended easily by adding new actor tokens into the parser code and enlarging the length of variable names.

6.3 The Parser

If we want to build a tool that can analyze any secret shared protocol in SHAREMIND, it is important to use some uniform grammar. Hence, as the first component of our implementation, the parser is in charge of the following two things:

1. Excluding and reporting the protocols which are not written in the valid form.
2. Importing the content of instances of sub-protocols and constructing the full protocol of the valid input protocols.

As we mentioned above, the parser code is generated by JavaCC. Before we present the approach of generating the parser code, let us have a look at JJTree in details. JJTree is a very important sub-tool included in JavaCC. It generates code to construct parse tree nodes and works as a preprocessor for JavaCC. JJTree takes input from the *.jjt* file and outputs a *.jj* file, where the *.jj* file is used as the input of JavaCC. Hence, to generate the parser code, we first define the actions of nodes in the *.jjt* file, then use the JJTree to generate the input of JavaCC, and at last we use JavaCC to generate the parser.

In the following subsections, we will discuss the two function of the parser separately.

6.3.1 Introduction of the Grammar

Before we can define the grammar, we must constrain the valid form of the words used in the protocol. Hence, we set that the following signs can be used:

Name	Sign	Name	Sign	Name	Sign
LPAREN	(	RPAREN	)	SEMICOLON	;
COLON	:	COMMA	,	SENDTO	->
EQUALS	=	PLUS	+	MINUS	-
TIMES	*	MOD	%	NUMBER	(["0"-"9"])+

Table 2: Available Signs

As for the additive secret sharing scheme, the content of the share multiplication protocol changes corresponding to the number of parties that can be corrupted by the adversary. In our implementation, we just provide three party share multiplication protocol and five party share multiplication protocol in advance. In the parser, we assume that there are five actors, whose names are Alice, Bob, Charlie, David and Eve, and these names are denoted as A, B, C, D and E separately.

We also need to formalize the names of different variables, which is shown as in Table 3.

Variable	Format
VARIABLE	(["a"-"z"])+["1"-"5"]["1"-"5"]
SHARE	(["a"-"z"])+["A"-"E"]
BITS	["a"-"z"]["A"-"E"](["0"-"9"])+

Table 3: Valid Variable Names

The *VARIABLE* is the defined as the name of random numbers or the name of the item computed by an assignment. The *SHARE* is classified as the name of additive shares. The *BITS* is used to represent the name of the bits extracted from a 32 bit long data.

By observing the protocols in chapter 4, we can generalize the following common actions:

1. The actors can generate uniformly distributed random numbers.
2. The actors may send messages to other actors.
3. The actors need to compute some results based on the information they already know.
4. The protocol may import other protocols as its sub protocols.

Correspondingly, we define the grammar for these actions as follows:

1. For presenting that an actor generates random numbers, we use the sentence as

   $$\textit{Actor\_Name} : \textit{Random}(\ \text{VARIABLE}_1, \ldots, \text{VARIABLE}_n\ ) \quad .$$

   The *Actor_Name* is the name of the actor who generates the random numbers, the actor name can be A, B, C, D or E. The sign *Random* indicates that the variables, whose names are shown in the parenthesis, are initialized as uniformly distributed random numbers.

   For example, the statement

   $$\textit{A}: \textit{Random}(r12, r13, r14)$$

   means that Alice generates three uniformly distributed random numbers, whose names are $r12$, $r13$ and $r14$.

2. To show that an actor sends a message to the other actor, we write a sentence as

   $$\textit{Sender\_Name} \rightarrow \textit{Receiver\_Name} : \textit{Message\_Name}_1, \ldots, \textit{Message\_Name}_n \quad .$$

   The *Sender_Name* and *Receiver_Name* are actor names, which can be chosen from $A, \ldots, E$. The sign $\rightarrow$ indicates that the messages *Message_Name*$_1, \ldots,$ *Message_Name*$_n$ are sent from the actor *Sender_Name* to the actor *Receiver_Name*. The *Message_Name* can be in form of *VARIABLE, SHARE or BITS*.

   For example, the statement

   $$A \rightarrow B : r12,\ sA,\ rA0$$

   means that Alice sends three messages to Bob, and these messages are a variable, a share and a 32 bit long message that represents the first bit of the share rA.

3. For computing the assignment, we can use the sentence as

Actor_Name : *Result_Name* = *Expression* .

The *Actor_Name* is the name of the actor who computes the *Expression*. The *Result_Name* is the name of the computing result, which should be in the form of *VARIABLE*. The *Expression* can be any expression, as long as all the arithmetic signs used in the expression are included in Table 2, and all the elements' names are in the form of what shown in Table 3.

For example, the statement

$$A : d12 = -2 * a12(sA + b12)$$

means that Alice computes the expression $-2 * a12(sA + b12)$ and the result is denoted as $d12$.

4. To import other protocols as sub protocols, we can use the following sentence

   Subprotocol : *Actor_Name*$_1$: *Output_Name*$_1$, . . . ,*Output_Name*$_n$,
   . . . ,*Actor_Name*$_n$: *Output_Name*$_1$, . . . ,*Output_Name*$_n$
   = *Subprotocol_Name (Actor_Name*$_1$: (*Input_Name*$_1$, . . . ,*Input_Name*$_n$),
   . . . ,*Actor_Name*$_n$: (*Input_Name*$_1$, . . . ,*Input_Name*$_n$)) .

   The sign *Subprotocol* indicates that a call of an instance of the sub protocol, whose name is *Subprotocol_Name*, is made. In current version of the program, the *Subprotocol_Name* should be *Multiplication, DuAtallah, TwoOutFive* or *shareconversion*.

   The sentence *Actor_Name : Output_Name*$_1$, . . . ,*Output_Name*$_n$ means that by importing the instance of the sub protocol, the name of the n outputs of an actor, whose name is *Actor_Name*, will be *Output_Name*$_1$, . . . ,*Output_Name*$_n$. It is required that the outputs of all actors participating in the sub protocol should be specified.

   On the other hand, the sentence *Actor_Name : (Input_Name*$_1$, . . . ,*Input_Name*$_n$*)* means that the actor *Actor_Name* will input n messages, whose names are *Input_Name*$_1$, . . . ,*Input_Name*$_n$ while calling the instance of the sub protocol.

   The setting of inputs and outputs of each actor should be in the same form as what is declared in the original content of the sub protocol.

   For example, the statement

   Subprotocol: *A:a12,B:b12,C:c12,D:d12,E:e12 = TwoOutFive(A:(uA),B:(vB))*

   means that an instance of *TwoOutFive* protocol is called as a sub protocol, and it is specified that while Alice inputs share uA and Bob inputs share vB, the name of the results of this instance of *TwoOutFive* protocol should be $a12$, $b12$, $c12$, $d12$ and $e12$.

After we have shown the rules of how each sentence can be composed, we will present the requirements of how to construct the whole protocol. Each protocol can be divided into three parts:

1. **Declaration of inputs**: the grammar of declaring the form of inputs is

   Input: *Actor_Name*$_1$: (*Input_Name*$_1$, . . . ,*Input_Name*$_n$),
   . . . ,*Actor_Name*$_n$: (*Input_Name*$_1$, . . . ,*Input_Name*$_n$)) .

   The word *Input* indicates that the following part of this sentence specified which actors will submit inputs in this protocol and what are the names of those input items.

2. **Main body**: the content of the protocol is written in this part. What actions each actor takes and which instances of protocols are imported are specified here.

3. **Declaration of outputs**: the outputs are declared by using the sentence

   Output: *Actor_Name*$_1$: *Output_Name*$_1$, . . . ,*Output_Name*$_n$,
   . . . ,*Actor_Name*$_n$: *Output_Name*$_1$, . . . ,*Output_Name*$_n$.

   The word *Output* states that the rest of this sentence defines the names of the outputs of this protocol for all actors, who participate in this protocol.

Let us take the Du and Atallah protocol as an example. Following our grammar rules, the Du and Atallah protocol can be written as:

Protocol 7 Du and Atallah Protocol

```
$Input : A : (uA), B : (vB);$                    //Declaration of input

$C : Random(r31, r32);$
$C \rightarrow A : r31;$
$C \rightarrow B : r32;$
$A : f12 = uA + r31;$
$B : f21 = vB + r32;$
$A \rightarrow B : f12;$
$B \rightarrow A : f21;$

$A : dA = -f12 * f21 + uA * f21;$
$B : dB = vB * f12;$
$C : dC = r31 * r32;$

$Output : A : dA, B : dB, C : dC$                //Declaration of output
```

The last rule of writing a protocol is that except for the last sentence, we must use a semicolon to indicate that one statement ends. In other words, the sentence without a semicolon would be taken as the last sentence.

6.3.2 Generating the Full Protocol

One of the special features of our implementation is that it can automatically import the instances of other protocols that already exist in our program. Hence, while writing the input protocol, we do not need to specify every message exchanged between the parties any more. Instead, we can designate what instance of which protocol we need to import, then the parser will generate the full protocol automatically and correctly.

As we mentioned in the former section, the statement of importing an instance of a sub protocol is as follows:

Subprotocol : *Actor_Name*$_1$: *Output_Name*$_1$, . . . ,*Output_Name*$_n$,
. . . ,*Actor_Name*$_n$: *Output_Name*$_1$, . . . ,*Output_Name*$_n$
= *Subprotocol_Name (Actor_Name*$_1$: (*Input_Name*$_1$, . . . ,*Input_Name*$_n$),
. . . ,*Actor_Name*$_n$: (*Input_Name*$_1$, . . . ,*Input_Name*$_n$)) .

This statement specifies the following information:

1. Which protocol will be imported as the sub protocol.

2. Who participates in this instance and who submits the inputs.

3. What are the names of inputs and outputs of the sub protocol instance.

As shown in Figure 7, the full protocol is constructed using the following approach.

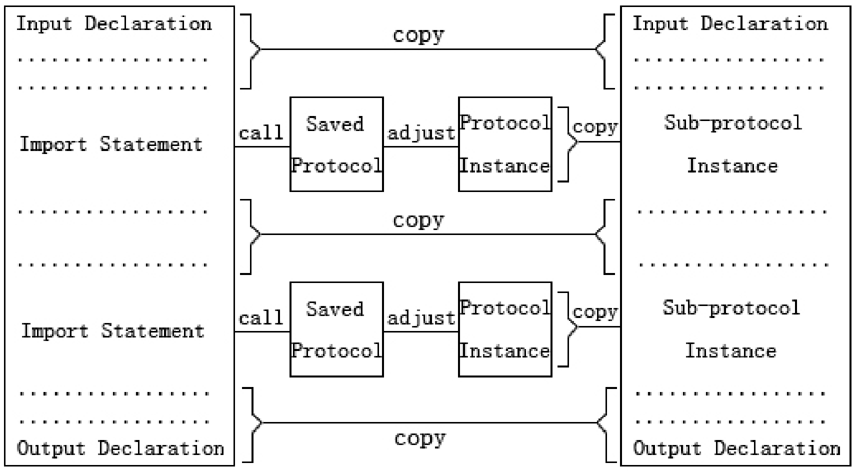

Figure 7: Approach of Constructing the Full Protocol

The input protocol is processed from statement to statement. Except for the statement of importing a sub protocol, all other statements are copied intactly into the full protocol. If an instance of sub protocol needs to be imported, the content of the designated sub protocol saved in our system is modified to meet the requirement of the sub protocol importing statement. After that, the content of the instance of the protocol will be copied into the full protocol.

The content of the full protocol must meet the following requirements:

1. There is no grammar error in the full protocol and different messages have different names.

2. It is the exactly correct extension of the input protocol, and each instance of sub protocols is correctly formed.

To meet these requirements, we implemented an algorithm to construct the instances of the imported protocols as follows:

- First, while parsing the input protocol, the parser does not only check the protocol's grammar, but also saves the name of each item of the input protocol into a vector V.

- Second, after the grammar is checked, the construction of sub protocol instances begins:

  1. For each import statement, the algorithm first saves the items which specify the participating actors and their outputs into a vector Out, and the items which indicate the actors who submit inputs and their inputs into a vector In.

     For example, for the statement

     $Subprotocol$: $A : a23, B : b23, C : c23$ = $DuAtallah(B : (uB), C : (vC))$,

     the algorithm will generate two vectors as $Out = [A, a23, B, b23, C, c23]$ and $In = [B, uB, C, vC]$.

  2. The algorithm opens the sub protocol, which is designated in the import statement, and gets its input and output declaration statements. Then the algorithm extracts the input and output information from the sub protocol, and saves them into two vectors $Sout$ and Sin, where $Sout$ contains the actors who is participating in the sub protocol and their outputs, and Sin includes the actors who submit inputs and their inputs.

     For example, for the statement in the former step, the algorithm opens the Du and Atallah protocol as Protocol 7, and generates two vector as $Sout = [A, dA, B, dB, C, dC]$ and $Sin = [A, uA, B, vB]$.

  3. The algorithm goes through the sub protocol, and changes all the items in vectors $Sout$ and Sin to the items in the same position of vectors Out and In. All the items been changed are marked to prevent being changed again.

  4. For each other item in the sub protocol, the algorithm will replace them using the following approach:

     – First, the algorithm generates two vectors $Svariable$ and $Variable$, where $Svariable$ saves the items need to be changed and $Variable$ saves the final item name.

- If the item is not in V, the algorithm will not change it, but this item needs to be added into V.
- On the other hand, if the item is already in V, check if it is in $Svariable$. If it is in $Svariable$, do nothing.
- If it is not in $Svariable$,then the algorithm adds the name into the vector $Svariable$ and finds a new available name for it. After the name is generated, the algorithm adds the name into the vector $Variable$ and the vector V.
- After all items are checked, the algorithm will go through the sub protocol and substitute the items in $Svariable$ to the item in the same position in $Variable$.

After the instance of sub protocol is constructed, except for its input and output declarations, other statements will be copied into the full protocol. Therefore, the full protocol contains one input declaration, one main body without sub protocol import statements and one output declaration. Except for the output declaration, each other statement ends up with a semicolon.

6.3.3 Future Extension

The current version of our implementation is especially designed for analyzing the SHAREMIND protocols with no more than five actors. Hence, only five actor names are defined in our protocol grammar. It can be easily extended to analyze protocols with more than five parties by defining more actors in the parser code. Note that when new actor names are defined, the scope of valid variable names should be changed too.

When new protocols are composed, their names can be added into the parser code. Hence they can be imported as the sub protocols of more complex protocols. On the other hand, while finding the new name for the item in sub protocols, we can increase the number of iterations of the renaming algorithm to enlarge the number of proper item names.

6.4 The Protocol Analyzer

The most important component of our implementation is the protocol analyzer. It is in charge of evaluating the security of the protocols. The normal approach of proving if a protocol is secure or not is to define an attack scenario first, then analyze whether the protocol can resist this attack. Hence, while analyzing the security of the protocols, the names of corrupted parties must be specified in advance.

After the analyzer gets the full protocol from the parser and the name of corrupted parties from the user input, the process of evaluating the security of the full protocol, which is under the attack of designated corrupted parties, is executed as in Figure 8.

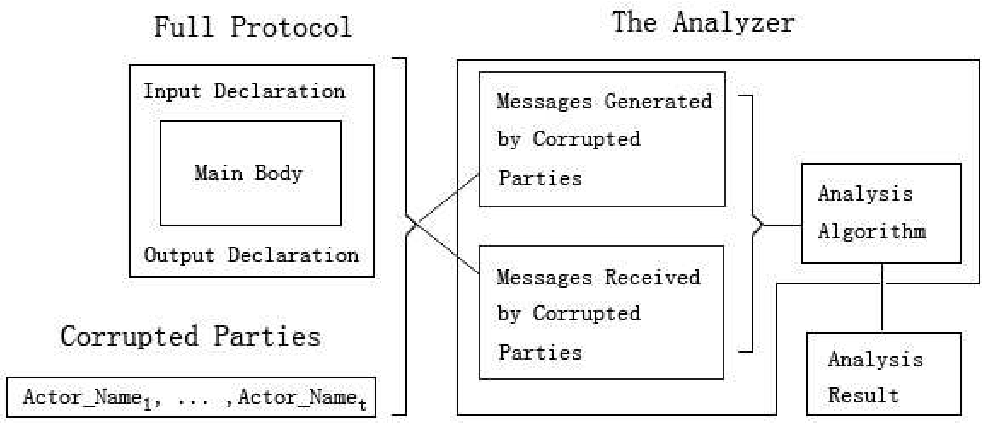

Figure 8: Approach of Evaluating the Full Protocol

The analyzer works using the following approach:

1. According to the corrupted parties' names, the analyzer looks through the input full protocol, and extracts all the messages generated or received by those corrupted parties, and saves messages generated by corrupted parties in one list and their received messages into another list. The sequence of the extracted messages is the same as their sequence in the full protocol.

2. The analysis algorithm takes the messages extracted in the first step as is input, and does the analysis on those messages.

3. After analyzing, the analysis algorithm outputs its analysis result as the result of the analyzer.

We can observe that the essential part of the analyzer is the analysis algorithm. In the analysis algorithm, the messages received by corrupted parties are classified into five categories:

1. A random number.

2. The sum of several random numbers.

3. The sum of a secret and a random number.

4. The sum of a secret and several random numbers.

5. The sum of items, which are values of expressions needed to be computed from other items.

In addition, the analysis algorithm adds one of the following two pads to each message:

1. *Known*: it means that all items used to compute this message are originally generated by the corrupted parties.

2. *Not*: it means that there is at least one item used to compute this message unknown to the corrupted parties.

The goal of the analysis algorithm is to determine if the corrupted parties can figure out more sensitive information from the messages they hold than they should. It requires that each message should be uniformly distributed to the corrupted parties. In that case, the analysis algorithm analyzes the incoming messages of corrupted parties sequentially, and records the items that make these messages random. We set a randomness determinant be the item, which makes the value of a message uniformly distributed. Each message is processed as follows:

1. First, the analysis algorithm checks if the message can be padded as *Known*. If it can, then the analysis algorithm adds *Known* as its pad. It is trivial that the corrupted parties get no more information from the messages padded as *Known*.

2. If the message is not known by the corrupted parties, then the analysis algorithm adds *Not* as its pad. After that, the analysis algorithm checks which type the message is, and does different operations according to the different types of messages:

   - If the message is a random number, the analysis algorithm goes through the former messages in the list of received messages and checks if there is any message, which is computed from secrets and other items, has only one randomness determinant and it is the same as this random number.

     For example, the message is $r12$ and we found a former message which can be expressed as $r12 + sA$, then the corrupted parties can get secret sA. These two messages compromise the security of the protocol.

   - If the message is the sum of several random numbers, the analysis algorithm checks if all these random numbers are used in the computations of the former messages. If the answer is yes, then the algorithm counts the number of the time each random number appears in the former messages. If the counting results are the same as the times of the corresponding random numbers appears in this message, then the protocol is insecure.

     For example, the new message is $r12+r13+r14+r23$, the protocol is insecure if we find the former messages as $sA + r12$, $sB + r13$ and $sC + r14 + r23$, because the corrupted parties can compute the value of $sA + sB + sC$.

   - If the message is the sum of a secret and a random number, the process of checking the security of the protocol is the same as for the first type of messages. If the corrupted parties still can not figure out more information after they received this message, the algorithm records the random number as the randomness determinant of this message.

   - If the message is the sum of a secret and several random numbers, the process is the same as for the second message type. Its randomness determinants are the random numbers that have not been used as randomness determinants of other messages.

- If the message is the sum of items needed to be computed by other items, the algorithm extends it to the form that each item used to compute the message is a random number or a secret. Then the analysis algorithm determines which type this message is and follows the process of that message type.

While evaluating messages received by corrupted parties, if any message is detected to compromise the former messages, the analysis algorithm stops the evaluating process and makes the report. The report shows the index of the message that will compromise the privacy of secrets.

If after receiving all the messages, the corrupted parties still cannot get more information than they should, then a security report is made by the analysis algorithm. The security report includes all the received messages, their randomness determinants and their status paddings.

6.5 The Simulation Generator

As mentioned above, the real world protocol securely realizes the ideal functionality F if for any real world adversary, there exists a simulator communicates with F in the ideal world that the output distribution of the adversary communicates with the real world protocol and the output distribution of the adversary communicates with the simulator are indistinguishable from each other. Hence, the simulator is very important in the security proofs of protocols. The goal of our implementation is to analyze the input protocol and prove its security by presenting the simulation result of corrupted parties.

The simulation generator in our implementation plays the role of a simulator. It takes the analysis report of the protocol analyzer and the full protocol as its input. If the report shows that the protocol is insecure, the simulation generator halts.

If the analysis report indicates that the protocol is secure, the simulation generator reads statements sequentially from the full protocol and generates the simulation result using the following approach:

1. For each statement, if no corrupted parties participate in the action depicted in the statement, the simulator generator does nothing.

2. If the corrupted parties participate in the action, then the simulation generator operates differently according to the type of the statement:

   - For the statement meaning that the corrupted parties send messages to honest parties, the simulation generator changes the name of the message receiver as the name of the simulator.
   - For the statement showing that the honest parties send messages to the corrupted parties, the simulation generator replaces the name of the message sender to the name of the simulator, then makes further operations according to the tag of that message in the analysis report:
     - If the message is tagged as *Known*, the rest of the statement remains the same.

  - If the message is tagged as *Not*, the simulator generator prints a statement to generate a random number, and changes the content of original statement to the name of the new random number.
- For other types of statements, the simulation generator copies them intactly into the simulation result report.

In a word, the output of the simulation generator is a protocol which shows the process of how the corrupted parties communicate with the simulator. By comparing the simulation result with the full protocol, we can observe that for the corrupted parties, the distribution of the messages received from the simulator and the distribution of the messages received from other parties are indistinguishable. It also proves that our analysis result is correct.

7 Conclusion

In this thesis, we present a framework for analyzing the security of secret shared protocols. Our main goal is to build a software, which can automatically prove the security of the secret shared protocols. We achieved this goal, and the practical implementation is presented in this work.

Our solution is based on the theory that a real world protocol securely emulates an ideal functionality if for any real world adversary there exists a simulator in the ideal world such that the adversary can not distinguish if it is communicating with the protocol or the simulator.

Our solution is especially designed for the secret shared protocols with no more than five computing parties. The additive n-out-of-n secret sharing scheme is used for distributing the secrets between parties. In this thesis, we present three party computation protocols for several operations on shares, which are computed by additive secret sharing scheme. These operations are addition, multiplication by a constant, multiplication, share conversion and bit extraction. We use both informal and formal methods to prove that in the semi-honest adversary model, where the adversary can only corrupt one party, no party can figure out more secrets than he should in these protocols.

The result of our solution is the implementation software called AutoProver, which is written in Java programming language. A tool called JavaCC is used for generating the protocol parser. AutoProver is a cross-plarform application which can be executed on Java virtual machine version 1.2 or greater. As no network connection is needed, the AutoProver works on single computer. We also present the manual and the source code of the current version of AutoProver.

Our implementation consists of three components, the protocol parser, the protocol analyzer and the simulation generator. A set of protocol grammar is defined for the parser. Except for checking the grammar of input protocols, the parser can construct complex full protocols by automatically importing the instances of other protocols as what are specified in the input protocols. The protocol analyzer takes the full protocol and the names of corrupted parties as its inputs. By evaluating the messages generated and received by the corrupted parties, the analyzer reports whether any secret can be compromised by these corrupted parties. If no party can get more information than he should, the simulation generator constructs the simulation results. For protocol debugging, the execution result of each component is saved in a *.txt* file.

We have composed a number of protocols, which are saved in our software. We have tested our implementation on these protocols, and it works very well. The instruction of how to compile and execute the AutoProver is presented in the appendix. In future, the AutoProver can be easily extended to meet the requirements of more complex protocols.

Hajusarvutusprotokollide automaatsed turvatõestused

Kokkuvõte

Käesolev magistritöö esitab raamistiku hajusarvutusprotokokollide turvalisuse automaatseks tõestamiseks. Raamistik on realiseeritud tarkvarapaketina, mis vastavaid tõestusi protokollide formaalsetest kirjeldustest genereerida suudab.

Protokollitõestuste genereerimisel kasutame reaalse *vs* ideaalse maailma ideoloogiat ning turvadefinitsiooni, mille kohaselt protokoll on turvaline, kui reaalset protokolli saab niimoodi simuleerida, et ründaja ei suuda eristada, kas ta töötab reaalses või ideaalses maailmas. Töös loodud turvatõestuste leidja põhiline komponent ongi vastavate simulaatorite genereeraator.

Esitatud tarkvara suudab automaatselt tõestada protokolle, mis kasutavad aditiivset ühissalastusskeemi ning pakuvad turvalisust ründajate vastu, kes suudavad passiivselt korrumpeerida n arvutavat osapoolt $2n + 1$-st. Praktikas valitakse enamasti $n = 1$, aga töö tulemusena valminud automaattõestaja saab hakkama ka suurema hulga osalistega protokollidega. Töös on toodud levinumate protokollide (liitmine, konstandiga korrutamine, korrutamine, bitieraldus ja osade teisendamine) mitteformaalsed tõestused ning näidatud, kuidas muuta neid formaalselt genereeritavateks ja verifitseeritavateks.

Automaattõestaja on realiseeritud programmeerimiskeeles Java (virtuaalmasina versioon 1.2 või kõrgem), kasutades JavaCC parseriraamistikku. Programmipaketi sisendiks on protokolli formaalne kirjeldus töö käigus loodud kõrgkeeles, mis sisaldab vajalikke primitiive (juhuväärtuste loomine, sõnumite saatmine, aritmeetilised tehted, alamprogrammide väljakutsed). Väljundiks annab pakett analüüsitud protokolli ja juhul, kui selle turvalisus õnnestus tõestada, ka vastava simulaatori. Programmipakett koos dokumentatsiooni ja rea testprotokollidega on esitatud käesoleva magistritöö lisades.

References

[1] D. Beaver. Foundations of secure interactive computing. *Lecture Notes in Computer Science: Advances in Cryptology CRYPTO' 91*, Volume 576/1992:377–391, 1992.

[2] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *Annual ACM Symposium on Theory of Computing archive, Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, 1988.

[3] G. R. Blakley. Safeguarding cryptographic keys. *in proceedings of the National Computer Conference*, 48:313–317, 1979.

[4] D. Bogdanov. How to securely perform computations on secret-shared data. 2007.

[5] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, 1996.

[6] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, pages 143–202, 2000.

[7] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. *Extended Abstract appeared in proceedings of the 42nd Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[8] D. Chaum, C. Crepeau, and I. Damgård. Multiparty unconditionally secure protocols. *Annual ACM Symposium on Theory of Computing Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[9] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. *In Proc. 26th IEEE Symp. on Foundations of Comp. Science*, pages 383–395, 1985.

[10] R. Cramer. Introduction to secure computation. *In Lectures on data security : modern cryptology in theory and practice*, 1561 of Lecture Notes in Computer Science:16–62, 1999.

[11] R. Cramer and I. Damgård. Multiparty computation, an introduction. *Lecture Notes*, 2004.

[12] W. Du and M. Atallah. Protocols for secure remote database access with approximate matching. *In Proc. of the First Workshop on Security and Privacy in E-Commerce*, 2000.

[13] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28:637 – 647, 1985.

[14] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. *IEEE Symposium on Foundations of Computer Science*, pages 427–437, 1987.

[15] R. Gennaro, M. Rabin, and T. Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. *Annual ACM Symposium on Principles of Distributed Computing*, pages 101 – 111, 1998.

[16] J. Katz and R. Ostrovsky. Round-optimal secure two-party computation. *Lecture Notes in Computer Science: Advances in Cryptology CRYPTO 2004*, 3152/2004:335–354, 2004.

[17] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Electronic Colloquium on Computational Complexity, Report No. 63*, 2004.

[18] M. Naor. Bit commitment using pseudo-randomness. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 4, 1991.

[19] J.-J. Quisquater, L. C. Guillou, and T. A. Berson. How to explain zero-knowledge protocols to your children. *Advances in Cryptology - CRYPTO '89: Proceedings*, 435:628–631, 1990.

[20] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. *Annual ACM Symposium on Theory of Computing*, pages 73–85, 1989.

[21] A. Shamir. How to share a secret. *Communications of the ACM*, Volume 22:612 – 613, 1979.

[22] A. C. Yao. Protocols for secure computations (extended abstract). *In Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science*, pages 160–164, 1982.

A Manual of Our Program

Our program named AutoProver is an automated analyzer of secret shared protocols, and it is a generator of the protocols that can be easily followed by the simulators. AutoProver is written in Java programming language, and it can be executed on any Java virtual machine version 1.2 or greater. In this appendix, we will introduce the features of AutoProver, how to compile and run the program and the API of each class of the source code.

A.1 Specific Features

The specific features of AutoProver are as follows:

1. **Simple Inputs:** It is hard to write long and complex protocols. The protocol writer may spend a lot of time to check the validity of the protocol's content. As a cure for this, AutoProver provides an automatic protocol extension function. Hence, the protocol writer just need to specify the instances of the sub protocols he or she needs to import, then AutoProver will do the rest to form the complex full protocols.

2. **Automatic Analysis:** While running the program, the user just needs to specify the name of the protocol which is going to be analyzed, and the names of the corrupted parties. Then AutoProver will automatically analyze the protocol and output the simulation results based on the analysis results.

3. **Understandable Outputs:** After running the program, we can get the syntax tree of the parser, the full protocol, the analyzing result and the simulation results. Each of them are saved in a *.txt* file. The user can check if there is any error during the process of executing the program.

A.2 Compilation and Execution

Before compiling the program, the user should copy the program from the CD to his or her hard disk. Let us denote the directory that the program is saved in as %AutoProver Home%. The program directory should contain the following files:

- %AutoProver Home%:
  - NewParser: %code of the parser written in JavaCC
    - NewParser.jjt
    - SimpleNode.java
  - Analyzer.java %code of automatic protocol analyzer
  - Simulator.java %code of the automatic simulation result
    % generator
  - Protocols: %predefined secret shared protocols
    - shareconversion

- ∗ DuAtallah
- ∗ Multiplication
- ∗ TwoOutFive
- ∗ TwoOutFiveMultiplication
- ∗ BitExtraction

If all files are saved, the user can compile the program by using the following commands:

1. To compile the parser:

   (a) Go into the directory *NewParser*, then print the following command:

   ```
   jjtree NewParser.jjt
   ```

   to compile the JJtree file. Then the JavaCC would generate the following files automaticly:

   - NewParser.jj
   - JJTNewParserState.java
   - NewParserTreeConstants.java
   - Node.java

   (b) Then the user should print the following command to compile the JavaCC file:

   ```
   javacc NewParser.jj   .
   ```

   If the compilation succeeds, JavaCC would automatically generate the following files:

   - NewParser.java
   - NewParserConstants.java
   - NewParserTokenManager.java
   - ParserException.java
   - SimpleCharStream.java
   - Token.java
   - TokenMgrError.java

   (c) Finally, the user needs to return to the %AutoProver Home% directory and compile the parser by using the command:

   ```
   javac NewParser\NewParser.java
   ```

2. To compile the analyzer, the following command is needed:

   ```
   javac Analyzer.java
   ```

3. The user can use the following command to compile the simulation generator code:

   ```
   javac Simulator.java
   ```

After successfully compiling the program, the user can use the following commands to run the program:

1. To run the parser only, please use the command:

   java NewParser.NewParser *protocol_name*

   The parser outputs are saved in files:

   - *ParseOuput.txt*: it contains the syntax tree of the protocol.
   - *FullProtocol.txt*: it contains the full protocol which includes the content of the sub-protocols specified in the input protocol.

2. To run the analyzer, which imports the parser code, please input the command:

   java Analyzer *protocol_name [actor_name]+*

   The analyzing outputs are saved in file:

   - *AnalysisOutput.txt*: it contains the analysis result.

3. To run the simulator, which imports the analyzer code, please print in the command:

   java Simulator *protocol_name [actor_name]+*

   The simulation result is saved in file:

   - *Simulator.txt*: it depicts the simulation.

A.3 AutoProver API Routines

We will introduce a comprehensive list of all classes, methods and variables, which are available for further code extensions, as follows:

1. The **SimpleNode** class provides functions for the nodes of the resultant parser tree of the input protocol. It also contains the functionality of generating the full protocol of the input protocol. Its API is shown in Table 4.

Name	**Description**
void process (PrintWriter ostr)	Print the tokens into a file
public void preprocess()	Save all tokens into a vector
public void getVeriable(Vector v)	Get the names of items in the input protocol
public void importProtocol (Vector v, PrintWriter ostr)	Change the parameter names of the sub-ptotocol according to what is specified in the input protocol and print the result to a file
void dumptofile (PrintWriter ostr, String prefix)	Print the parser tree to a file

Table 4: API of Class SimpleNode

2. The **Analyzer** class provides functions to analyze the parsed protocol and print out the analysis results, whose API is shown Table 5.

Name	**Description**
public void preparseFile (String file_name) throws Exception	Implement the parser to generate the full protoc -ol of the input protocol
public void parseFile (String file_name) throws Exception	Parse the full protocol
public Vector getTokens()	Save the tokens of the full protocol into a vector
public Vector getAllRandom()	Generate a vector which contains the names of all random numbers
public void getRandomItems (Vector actor)	Generate a vector which contains the names of random numbers generated by corrupted actors
public Vector getAssignments()	Generalize all the assignments and save them in a vector of vectors
public void getActorView (Vector actor)	Get all the messages sent to corrupted actors, which excludes the messages they send to each other
public Vector getActorAssignment (Vector actor)	Get the assignments computed by the corrupted actors
public Vector extend(Vector temp)	Extend the assignment till no items can be represented as other assignments
public Vector analysis() throws Exce -ption	Analyze whether the joint view of corrupted ac -tors is secure or not

Table 5: API of Class Analyzer

3. The **Simulator** class generates the simulation results if the protocol is determined as secure by the Analyzer, and its API is as in Table 6:

Name	**Description**
public void simulate(Vector input) throws Exception	To generate the simulation results based on the vector which contains the analysis outputs
public String getVarName(Vector v1 , Vector v2, Vector actor, Vector vi)	Find a proper name for the random numbers generated by the simulator

Table 6: API of Class Simulator