TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master-Thesis

Web Application for Privacy-Preserving Assignments

Taha Tariq

April 3, 2018

CRISP
Center for Research
in Security and Privacy

Technische Universität Darmstadt
Center for Research in Security and Privacy
Engineering Cryptographic Protocols

Supervisors: Prof. Dr.-Ing. Thomas Schneider
M.Sc. Agnes Kiss

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Taha Tariq, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Taha Tariq, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt, April 3, 2018

Taha Tariq

Abstract

Privacy is gaining more and more attention due to users awareness of their personal information being compromised. One method of ensuring privacy is to use secure function evaluation, which allows two or more distrusting parties to jointly evaluate a function without sharing their inputs and learn nothing more than the result.

In this thesis, we design and implement a framework for privacy-preserving assignments. We solve the assignment problem in $\mathcal{O}(n^4)$ running time. Our system takes the preference list of the users as inputs and uses RSA encryption along with secret sharing to protect the input of the user so that no party can learn about the preference of the user from their share. CBMC - GC [HFKV12] and ABY [DSZ15] are then used to create and evaluate Boolean circuits using Yao's GC [Yao86] and the GMW protocol [GMW87], and after the evaluation, the results are then sent to the users. Our implementation of the Hungarian algorithm for Boolean circuits has been tested for up to 13 participants. It can support a large number of participants and can be used to generate large circuits for secure computation. We then benchmark our implementation performance and conclude that the runtime and number of AND gates increase with the increasing circuit size because more data needs to be transferred. Increase in circuit size means that the number of the participants have increased which means that the new circuit needs more gates to hold these values.

Acknowledgments

To begin with, I would like to express my deep gratitude to Agnes Kiss for supervising this thesis. She made herself available whenever I reached out to her for help through various phases of the thesis. She supported my ideas and gave me the liberty to implement my own methods while giving me helpful insights along the way.

Secondly, I would also like to thank my supervisor Prof. Dr.-Ing. Thomas Schneider for giving me an opportunity to work in his department and providing his valuable insights and feedback on the thesis.

Also, special mentions to Syed Muhammad Ali for his valuable comments, and Hoor Nazar and Fatima Akhlaq for proof-reading my thesis. I would like to thank all my friends for their motivation and continuous support during this time.

Finally, I would like to thank my family, especially my parents who supported and encouraged me throughout my studies especially in the thesis period. This would not have been possible without them.

Contents

1 Introduction

Privacy has become an important concern with increasing amount of data. In online transactions, where a transfer of personal user data is required, ensuring privacy is not always straight forward. Additionally, with an increasing number of mobile devices like smart watches, assistants, and home automation units, the users are losing control over their personal information submitting personal data knowingly or unknowingly. The information may be sold to third parties or shared with other partner companies. For e.g., in a recent scandal which involved a breach of personal information without authorization, a company named Cambridge Analytica harvested the profiles of more than 50 million users by exploiting Facebook. They collected the data using an app on Facebook which could access the data of user's friends without their consent and later used the data for targeting the individual users with personalized political advertisements for US presidential elections in 2016 [CG18]. This has caused users to move to platforms they think are secure and do not breach their privacy. Most online platforms rely on trusted third party providers for the processing of their data.

Secure computation has become an important solution in such scenarios where the users do not feel secure about sharing their private data but want to perform a particular computation. The idea of secure computation was first introduced by Andrew Yao [Yao86] which is commonly known as Yao's millionaire's problem. In this problem, two millionaires want to know who is wealthier without revealing their actual wealth. They want to do it without the involvement of any third party. The solution he provided for the problem formed the basis for secure-two party computation which is further discussed in Section 2.4. Since Yao's seminal work there has been a lot of research on secure multi-party protocols. However, practical implementation had not appeared until recently.

Since privacy is the right of every user, it is essential for almost every problem concerning the personal data of the user. Some examples include processing of employees data, processing of customer data, and data in banks. One such example can be the assignment problem. Consider two sets A and B, set A contains a list of students and set B contains a list of dormitories, all elements in set A needs to be assigned to all elements in the set B concerning their preference. Since preference is personal, the user might not feel comfortable disclosing it to others. Also, the confidentiality of the data should not be breached which might result in the outcome being influenced. In the course of this thesis, we focus on solving the assignment problem in a privacy-preserving manner.

We use ABY [DSZ15] along with CBMC - GC [HFKV12] to manage the secure computation protocol which ensures security with correctness. We use state of the art tools to achieve

this goal. The data is collected, processed, used and destroyed after we are done with the computation.

1.1 Motivation

Achieving data privacy is still considered to be a challenging problem because of the increasing use of smart devices. Companies hire third-parties to process sensitive information which in principle is also a breach of privacy because the information is mostly available in the clear and can be manipulated. To protect the data of the user many cryptographic techniques can be used. The companies may also use the data for research purposes. Often a user wants to be part of a system as an anonymous or private participant without revealing too much personal information. This can be made possible by using proper tools and processes that protect the privacy of the user and also practices like deleting the data of the user once it is processed. The user should have the right to get his data deleted whenever he wants.

Assignment problems, such as vendors bidding for projects, requires processing and storing sensitive information to award projects to the best candidates. A huge amount of money is involved in such cases, and a high level of trust factor is required for the processing of such information. This thesis focuses on solving the assignment problem in a privacy-preserving manner by utilizing secure computation techniques.

Assignment problems are widely used for different tasks, and one such example of what we focus on in this thesis is a student-topic assignment. Consider a case where some students register for a seminar course and can get a topic to work on. The students have priorities for the offered topics and would like to have the topic that interests them the most. The user does not want its preference to be shared or the results to be influenced in any way and a fair assignment to take place. We build such a system that would help the students get a topic of their choice while keeping their preference private.

1.2 Contributions

We design and implement a framework for privacy-preserving assignments. We utilize the frameworks from [HFKV12] and [DSZ15] to compute the assignments securely. We optimized the assignment algorithm for bounds and memory usage to reduce the computation and communication cost by using the smallest possible data structures for our data which helped in reducing the number of gates in the Boolean circuit. Also, for all loops, we specified the bounds inside the program which are based on the size of the participants which helps us in generating the Boolean circuit in CBMC - GC without the need for the framework to calculate them.

1.3 Outline

This thesis is divided into 7 Chapters. In Chapter 2, we define the necessary notations and the theoretical background required for this thesis. This chapter contains the introduction to logic gates, Boolean circuits, secure computation protocols and the two frameworks used by us CBMC - GC and ABY. In Chapter 3, we discuss the secure matching problems which are related to our work, for e.g., secure stable matching. In Chapter 4, we describe the design of our framework which includes details of our three modules and the Hungarian Algorithm. Chapter 5 contains the implementation details of our framework. In this chapter we explain our implementation choices and the technologies that we have used to achieve the system. In Chapter 6 we discuss the results. Chapter 7 concludes the thesis and gives the direction of the future work.

2 Preliminaries

In this chapter, we introduce the fundamental concepts and notations which are used later in the thesis and are required to understand the topic. We start by describing logic gates (Section 2.1) and operations and move on to explain Boolean circuits (Section 2.2) and related concepts which are the building blocks of this thesis. We then describe the Boolean circuit-based protocols for secure function evaluation which are Yao's Garbled Circuit Protocol (Section 2.4) and the GMW protocol (Section 2.5). They use Oblivious Transfer (Section 2.3) and Multiplication Triples (Section 2.5.1) which we further explain in this chapter. In Section 2.7, we introduce the ABY framework which is used for secure two party computation. In Section 2.6, we introduce different compilers which have been developed for Secure Two-Party computation (STC). The CBMC-GC framework [HFKV12] is discussed in detail, the whole construction and process. Its optimizations are then discussed which are mostly aiming to reduce size and depth. We then present a comparison of all of these compilers.

2.1 Logic Gates and Operations

Logic gates are the building blocks of a digital circuit. Most of the logical operations can have two or more inputs except for the NOT operation which takes only a single input. In this context, 0 denotes false, and 1 denotes true. Some of the basic logic gates include $AND, OR, NOT, XOR, NAND, NOR, XNOR$.

2.2 Boolean Circuit

Boolean Circuits can be defined as the standard representation of Boolean functions that are commonly used in multiple applications, e.g., digital electronics and secure computation. We denote the number of inputs by n, the number of outputs by m and the number of gates by g. Figure 1 shows an example where $n = 3$, $m = 1$ and $g = 4$.

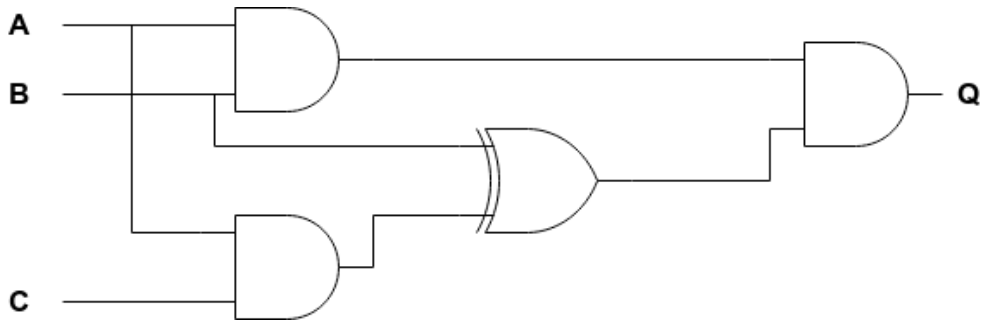

Figure 2.1: A Boolean Circuit with 3 inputs, 4 gates and 1 output

According to Arora and Barak [AB09], a Boolean circuit with n inputs, m outputs, and g gates is a Directed Acyclic Graph with n input nodes with no incoming edges and m output nodes with no outgoing edges. The edges are called wires, and all other nodes are called gates which are labeled with one of the logical operations, e.g., AND, OR or NOT. The size of the boolean circuit is measured by the number of nodes in it, and the depth of the circuit can be measured by the maximum distance from an input to an output.

2.2.1 Functional Completeness

A set of Boolean functions is called functionally complete if all other Boolean functions can be constructed from this set. For instance, 2-input, 1-output functionally complete set is $\{AND, XOR\}$ which is also the common basis for secure computation. This means, e.g., that we can create any circuit out of only AND and XOR gates.

2.2.2 Topological Order

The topological order of a Boolean circuit is ordering of its gates $G_1, \ldots, G_n$ which satisfies the property that no gate G_i has inputs that are outputs of a successive gate $G_{j>i}$. As long as all the current gate's inputs are known, the gates of a Boolean circuit can be evaluated. To ensure this property, the gates are sorted and evaluated in topological order for which, e.g., specific graph traversal algorithms can be utilized [Sch12].

2.3 Oblivious Transfer

Oblivious Transfer (OT) is a protocol which allows two parties a sender and a receiver - to transfer information. A sender sends m pieces of information of which the receiver

receives one piece of the information according to his choice. The sender remains oblivious to the information transferred while the receiver would only know about the piece received.

Suppose Alice wants to give discount vouchers with every product purchased online. Every time Bob purchases a product, they run an oblivious transfer protocol with Alice's input being $x_1, \ldots, x_n$ (where x_i is a valid pair of the message and signature) one of which is chosen at random by Bob, and in the end, Bob has a valid voucher without Alice knowing the exact one which would ensure Bob's privacy as he does not want Alice to know where is he exactly spending them. In this way, Bob can never guess the other vouchers causing any harm to Alice's business.

The concept was first introduced by Rabin in 1981 [Rab05]. To build protocols for secure multi-party computation a more useful form of OT was developed by Even, Goldreich, and Lempel [EGL85]. In a 1-out-of-2 OT protocol, the sender has two messages $m_i, i \in \{0, 1\}$, and the receiver can choose one out of them by providing a bit i. In this way, the receiver only learns about the received message, and the sender remains oblivious to the message transferred. The protocol was later generalized to 1-out-of-n oblivious transfer with n input values and one selection value instead of bit. OT is extensively used for secure computing protocols.

Oblivious transfer protocols traditionally require expensive public-key operations. To overcome this, [IKNP03] proposed OT extensions which are secure against semi-honest adversaries. They start by creating a small number of OTs (base OTs) using public-key cryptography and extend them by creating any polynomial number of OTs based on symmetric key cryptography.

2.4 Yao's Garbled Circuit Protocol

Yao's Garbled Circuit Protocol [Yao86] allows two mistrusting parties without the presence of a trusted third party to evaluate a function over their private inputs securely. Yao's GC protocol is secure against semi-honest adversaries but has extensions to malicious security [ALSZ15].

2.4.1 Garbled Circuit

A garbled circuit is a cryptographic protocol which supports secure computing. The process of garbling the gate as illustrated in Figure 2.2 and shown in Table 2.1, the inputs, and outputs of the gates are replaced by keys, and the mapping is kept a secret so that even knowing the output of a gate does not give any information about its true value.

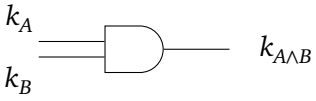

Figure 2.2: Garbled AND gate with two inputs and one output whose table is Table 2.1.

AND			
A	B	$A \wedge B$	GCT
k_A^0	k_B^0	$k_{A\wedge B}^0$	$\mathrm{E}k_A^0(\mathrm{E}k_B^0(k_{A\wedge B}^0))$
k_A^0	k_B^1	$k_{A\wedge B}^0$	$\mathrm{E}k_A^0(\mathrm{E}k_B^1(k_{A\wedge B}^0))$
k_A^1	k_B^0	$k_{A\wedge B}^0$	$\mathrm{E}k_A^1(\mathrm{E}k_B^1(k_{A\wedge B}^0))$
k_A^1	k_B^1	$k_{A\wedge B}^1$	$\mathrm{E}k_A^1(\mathrm{E}k_B^1(k_{A\wedge B}^1))$

Table 2.1: Truth table for two-input Garbled AND operation.

A garbling scheme consists of three operations, i.e., garble, encode and evaluate. Garbling is a way to convert a plain circuit C into an encoded circuit $\hat{C}$. In the encoding operation, the plain input x is converted into a garbled input $\hat{x}$ by using the secret randomness that was used to garble the circuit and is transferred to the other party using oblivious transfer. In the evaluation mode, the circuit $\hat{C}$ and input $\hat{x}$ are taken to compute the output of the circuit $C(x)$ obliviously. This is done by going through all the gates and decrypting the rows in the garbled tables until the output labels are obtained [BHR12].

2.4.2 Classic Protocol

Alice has private input a and Bob has private input b, and they both agree on some function f. They both want to learn $f(a,b)$ but do not want the other one to learn anything more than that.

The parties express function f as a circuit. Alice (garbler) then garbles the circuit C_f to $\hat{C}_f$ and sends $\hat{C}_f$ along with her garbled input $\hat{a}$ to Bob. Bob now has his private input b, but only Alice knows how to encode any input for C_f into a garbled input. The parties use Oblivious Transfer (Section 2.3) for Bob to learn the garbled version $\hat{b}$ without Alice learning about b. Since Bob has the garbled circuit $\hat{C}_f$ and the garbled inputs $\hat{a}$, $\hat{b}$, now he can run the evaluation process and learn the garbled output of the circuit. Since both of them need to learn $f(a,b)$ so, Alice will send the mapping of garbled output to real values to Bob, and then he can reveal it to Alice. The protocol also works if only one party wants to learn the output of the function.

technique	size per gate		calls to $A \| B$ per gate			
			generator		evaluator	
	XOR	AND	XOR	AND	XOR	AND
classic [Yao86]	4	4	4	4	4	4
point-and-permute [BMR90]	4	4	4	4	1	1
free XOR [KS08]	0	4	0	4	0	1
GRR3 + free XOR [NPS99]	0	3	0	4	0	1
half gates [ZRE15]	0	2	0	4	0	2

Table 2.2: Optimization to Yao's Garbled Circuits [ZRE15]

2.4.3 Optimizations

Yao's garbled circuit protocol is one of the most efficient approaches for secure two-party computation (STC) until today, and several optimizations for the protocol have been introduced over time. Yakoubov [Yak17] has discussed in detail about the optimizations to Yao's GC. We summarize some relevant ones below.

2.4.3.1 Point and Permute

In the point-and-permute technique introduced by Beaver, Micali and Rogaway [BMR90], random permutation bits s_0 and s_1 are associated with each wire w. The permutation bits can be safely revealed to the evaluator as they are independent of the wire label's value. The ciphertext of the garbled gate is then ordered according to the permutation bits of the input wires. The permutation bits can then be used to point to the evaluator which ciphertext should he decrypt without revealing the wire label itself. This way, the evaluator has to decrypt only one ciphertext per gate.

2.4.3.2 Garbled Row Reduction GRR3

Garbled row reduction allows one ciphertext to be eliminated which reduces the size of the garbled tables from four rows to three rows as shown in the Table 2.2. This is achieved by picking one label in such a way that we can eliminate one ciphertext per gate (AND/XOR). Moreover, the technique is compatible with free XOR [Yak17].

2.4.3.3 Free XOR

In the classic and point-and-permute techniques, the computation and communication complexity is the same for an AND gate as for an XOR gate. Kolesnikov and Schneider [KS08] introduced a mechanism which allows XOR gates to be evaluated for free (i.e., without

communication cost or cryptographic operations). When three wires touching an XOR gate have the same offset, then the XOR gate can be garbled for free, so they arrange these three wires in the circuit to have the same offset. The wire labels are chosen of the form $(A, A \oplus R)$, where R is secret but common to all wires. An evaluator with one of $(X, X \oplus R)$ and one of $(Y, Y \oplus R)$, can simply XOR the two wire labels to perform the XOR operation resulting in Z or $Z \oplus R$ (where $Z = X \oplus Y$) which is the correct representation of the result [ZRE15].

2.4.3.4 Half Gates

Zahur, Rosulek and Evans [ZRE15] introduced the idea of half gates where an AND gate can be garbled with two ciphertexts, and an XOR gate can be evaluated for zero ciphertext.

E.g., consider an AND gate with a, b as input wires and c as output wire. An AND gate is broken into two half AND gates for which one party knows one input. This way only one ciphertext is needed to garble one half-gate. First half-gate is when the garbler knows the value of a. He further needs to have two ciphertexts. Second half-gate is when the evaluator knows the value of a. He further needs to have two ciphertexts from the garbler. In any case, the maximum number of ciphertexts needed to garble the circuit are two. A gate $c = a \wedge b$ for $r \in \{0, 1\}$ can be expressed as:

$$c = (a \wedge r) \oplus (a \wedge (b \oplus r)) \tag{2.1}$$

In the first half gate $(a \wedge r)$ a garbler knows r, and in the second half gate, $(a \wedge (b \oplus r))$ the evaluator knows $(b \oplus r)$ from the point-and-permute random bit. Hence, every AND gate needs two ciphertexts, and free-XOR also holds true.

2.5 GMW Protocol

The Goldreich, Micali and Wigderson protocol (GMW) is another protocol for secure Boolean circuits evaluation which was introduced in 1987 [GMW87]. It is an alternative approach to Yao's Garbled Circuits which can be extended to multiple parties. The protocol is secure against semi-honest adversaries. Assuming that the parties are honest but curious about the information from another party. It only allows XOR and AND gates which do not pose boundaries, because they are functionally complete (Section 2.2.1). Every input and output is shared between the two parties in such a way that each party $i \in \{0,1\}$ holds a seemingly random share v_i where $v = v_0 \oplus v_1$.

2.5.1 Multiplication Triples

A Multiplication Triple (MT) [Bea91] can be used as an alternative to OT for the evaluation of AND gates in the GMW protocol. It is a triple of bits (a, b, c) which satisfies the equation for two parties P_i with $i \in \{0,1\}$.

$$c_1 \oplus c_2 = (a_1 \oplus a_2)(b_1 \oplus b_2). \tag{2.2}$$

MTs can be securely generated by a trusted third party or by using 1-out-of-4 OT protocol before the computation as they are independent of the input from parties. They are used to mask the actual value, so it is necessary that each party only knows about its MT values. Each MT can only be used once so for every AND gate a new MT is calculated.

2.5.2 The Protocol

The GMW protocol can be divided into two phases: the setup phase and the online phase. In the setup phase, all operations that do not require an input value are performed while the online phase needs input values. Most of the work is done in the setup phase which is generating Multiplication Triples (MT) for each AND gate via OTs. In the online phase, input sharing, function evaluation and output sharing between the parties takes place. The Boolean circuit is evaluated gate wise in topological order, and the AND gates are evaluated using MTs.

XOR gates with the output c can be evaluated as: $c = c_1 \oplus c_2$ which is equivalent to $(a_1 \oplus a_2)$ $\oplus$ $(b_1 \oplus b_2)$ equals to $(a_1 \oplus b_1) \oplus (a_2 \oplus b_2)$. Therefore, each party P_i can compute its output share locally without communication effort by $c_i = a_i \oplus b_i$.

AND gates can be evaluated using Multiplication Triples. Both parties generate a random MT for each AND gate using OT operations which can be done in the offline phase. Each party P_i has the values x_i,y_i,z_i satisfying $z_i = x_i \oplus y_i$.

In the next step, the values are secretly shared between the parties. Both parties calculate $d_i = a_1 \oplus x_1$ and e $= b_1 \oplus y_1$ and exchange the results which further allows them to compute $d_i = d_0 \oplus d_1$ and e $= e_0 \oplus e_1$. After computing d and e they can calculate their part of the result as :

$$c_0 = (d \oplus e) \oplus (y_0 \oplus d) \oplus (x_0 \oplus e) \oplus z_0, \tag{2.3}$$

$$c_1 = (y_1 \oplus d) \oplus (x_1 \oplus e) \oplus z_1. \tag{2.4}$$

There have been continuous improvements in the GMW protocol. To accelerate the process, OT execution and MT generation can be performed in parallel [CHK+12; SZ13]. Schneider and Zohner [SZ13] have also discussed in detail some improvements like depth-optimized circuit constructions for the GMW protocol.

2.6 Compiling Boolean Circuits

The practical implementation of Secure Two-Party computation (STC) is facilitated by the existing secure computation protocols. A program must be converted to a Boolean circuit before it gets evaluated securely. Since such conversions are not straightforward, so in the past hand-crafted Boolean circuits or domain-specific languages were used for STC. C Bounded Model Checker for Garbled Circuits (CBMC-GC) is the first open-source ANSI C compiler for STC.

2.6.1 Related Compilers

This section summarizes the discussion by Buescher et al. [BFH+17] on related work on compilers for multi-party computation (MPC). Compilers for MPC can be classified into two categories based on what purpose they serve: standard programming language or domain-specific language (DSL). Standard programming languages are general purpose programming languages that can be used broadly across application domains but do not contain features for a specific domain, e.g., C, C++, and Java. Domain-specific languages are computer languages which are designed to focus on a particular domain. They are written to deal with a specific set of concerns, e.g., maven and make for describing software builds. There are two different ways in which a DSL can be classified, internal or external. Internal DSL can be called integrated compiler as it is integrated into an MPC framework while external DSL is parsed independently of the MPC framework so it can be called independent compiler.

2.6.1.1 Internal DSL

An intermediate representation is produced by the integrated compilers, which is only instantiated by the circuit (interpreted) while an MPC protocol is being executed. These interpreted circuit descriptions commonly allow the circuit to be represented compactly.

Some integrated compilers support the execution of mix-mode secure computation. Mix-mode computation allows code to be written in such a way that it can differentiate between public and private computation, which lets the mix-mode program to be expressed in a single language but also tightens the coupling between the compiler and the execution framework. Furthermore, some integrated compilers also support the compilation of programs for hybrid secure computation protocols, which combine different cryptographic approaches.

2.6.1.2 External DSL

The independent compilers, as its name would suggest, allow the circuit to be created independently from the framework, it is being executed. The advantage of an independent compiler is that the produced circuits can be optimized to their full extent during compile time and thus prove to be more versatile in their use in MPC frameworks.

2.6.1.3 Compilers for MPC

In this section, some of the MPC Compilers are discussed and compared based on the discussion in [BFH+17]. The KSS Compiler by Kreuter et al. [KSS12] uses gate-level optimization methods, such as constant propagation and dead-gate elimination and is also the first compiler that shows scalability up to a billion gates. KSS compiles a DSL into a flat circuit format. The PCF compiler by Kreuter et al. [KSMB13] compiles C programs by using the portable LCC compiler [FH95], which is an ANSI C compiler for various platforms, as a frontend. PCF converts an intermediate bytecode representation of LCC into an interpreted circuit format. PCF supports comparably limited optimization methods, that too can be applied only locally to all functions, but it is still more scalable than CBMC-GC.

ObliVM by Liu et al. [LWN+15] allows the efficient development of oblivious algorithms by compiling a DSL which supports the combination of oblivious data structures with MPC. According to Beuscher et al. [BFH+17], ObliVM provides insufficient optimization methods. Obliv-C by Zahur and Evans [ZE15] compiles a modified variant of C into an executable application that supports oblivious data structures and mix-mode computations.

TinyGarble by Songhori et al. [SHS+15] compiles circuits from hardware description languages, e.g., VHDL, by using commercial hardware synthesis tools. This approach requires the developer to be seasoned with hardware design but also enables the vast scope of existing functionality in the hardware synthesis to be used. The Frigate compiler by Mood et al. [MGC+16] also compiles DSL but the compilation is extensively tested and highly scalable. Both TinyGarble and Frigate produce compact circuit descriptions that compress sequential circuit structures.

2.6.2 CBMC-GC Framework

CBMC-GC [HFKV12] is a compiler based on a software verification tool CBMC [CKL04]. CBMC implements bit-precise bounded model checking for ANSI C. It translates an input C program to a Boolean constraint which represents a program behavior. CBMC-GC adapts this capability to provide circuits needed for STC. Figure 2.3 illustrates how CBMC-GC works in the STC toolchain. CBMC-GC translates a C program to a Boolean circuit which is then deployed by the two parties A and B using the STC platform. The two parties then use an STC framework also explained in Section 2.4 to evaluate the circuit and obtain the results.

In comparison with other state-of-the-art compilers, CBMC-GC compiles functionalities into up to four times smaller circuits than other compilers using the same source code for the function. Also, CBMC-GC requires a number k as an input to bound the size of the program. This constraint allows CBMC-GC to be terminated in a finite number of steps. The circuit compilation in CBMC-GC can be divided into five steps as shown in Figure 2.3 and are discussed below. The first three steps are a part of the standard CBMC processing while the other two are specific to STC tasks [FHK+14]. Now each step is described in detail below.

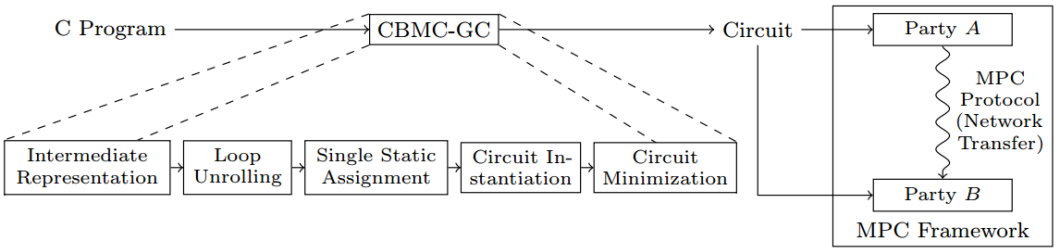

Figure 2.3: Tool chain of CBMC-GC[BFH+17]

2.6.2.1 Intermediate Representation

In this step, the C program gets translated into an intermediate representation called GOTO program. All non-linear control flows such as loops and jumps, switches and if statements are translated to equivalent guarded goto statements. Guarded GOTO statements are branch conditions which include (optional) conditions, and guarded GOTOs are the only control instructions that remain in the GOTO program. One GOTO program for each function gets generated [KT14].

2.6.2.2 Loop Unrolling

Loop unrolling, also know as loop unwinding is a loop transformation technique to optimize the execution time of a program. To have a loop-free representation of the program, the recursive function calls and loops are unrolled up to a specific depth. The loops are replaced by a sequence of k nested if statements and the recursive function calls are expanded up to k times which makes the program acyclic. This depth of k is automatically calculated by a static analysis but can also be specified by the user in case it fails to calculate it automatically. Static analysis is a method of gathering information about a program without executing it. Listing 2.1 and Listing 2.2 show the code example of loop unrolling where a loop which is expanded up to 3 times. The loop is simply calling an add function which adds the value of i until the condition is met.

```
1  i=0;
2  while(i!=3) {
3      func_add(i);
4      i++;
5  }
```

Listing 2.1: Before loop unrolling

```
1  i=0;
2  if (i!=3) {
3      func_add(i);
4      i++;
5      if (i!=3) {
6          func_add(i);
7          i++;
8          if (i!=3) {
9              func_add(i);
10             i++;
11         }
12     }
13 }
```

Listing 2.2: Loop-free representation of the program for 3 iterations of the loop

2.6.2.3 Single Static Assignment

Single Static Assignment (SSA) is a property of intermediate representation (Section 2.6.2.1) which requires that every variable is defined before it is used, and each variable should be assigned only once. The program can be converted to SSA form since it is acyclic. For e.g., $x = x + 1$ is replaced by $x_1 = x_0 + 1$, and further on, x_1 is used where the incremented x value is to be applied.

2.6.2.4 Circuit Instantiation

In this step, CBMC replaces the variables by bit vectors. For example, depending on the system architecture, an integer would be converted to a bit vector of size 16 or 32. The operations over variables are also translated to Boolean functions. CBMC-GC changes circuit generation in some places to reflect actual computation because CBMC aims to produce reliable instances for a SAT solver and has the freedom to generate circuits which are equisatisfiable with the expected circuits of CBMC-GC but are logically inequivalent.

2.6.2.5 Circuit Minimization for STC

This step aims to minimize the number of AND gates by using XOR gates in the resulting circuit while keeping the circuit size small since the XOR gates can be evaluated for free (Section 2.4.3.3). AND-INVERTER graphs (AIG) are directed acyclic graphs whose nodes represent logical AND gates, and the edges represent wires between gates. Also, some of these wires can negate the transmitted signal. Constant propagation is the process of substituting values of a known constant in expressions during compile time. In CBMC-GC zeros and ones are substituted while performing constant propagation. Circuit rewriting is a greedy optimization algorithm used in logic synthesis [MCB06]. A type of circuit rewriting is pattern based rewriting which allows reducing the number of non-linear gates by applying patterns that favor linear gates. In circuit rewriting all gates are first ordered in topological order (Section 2.2.2) by their circuit depth. Then the patterns are matched against all gates and sub-circuits by iterating over the gates. The sub-circuit becomes a candidate for substitution whenever a match is found, but it will only be substituted if the replacement will lead to an improvement in the circuit size. Structural hashing is a technique of combining and reusing multi-input expressions and subexpressions; it helps in removing duplicated gates structures in CBMC-GC. SAT sweeping is a minimization tool which is widely used for equivalence checking of combinatorial circuits [Kue04]. The idea of SAT sweeping is to prove that the output of a sub-circuit is either constant or equivalent to some other sub-circuit (which also helps in detecting duplicates). In both of these cases, the sub-circuit can be removed because it is unnecessary. For circuit minimization, the program should first be translated to an intermediate circuit. CBMC-GC uses AIGs as an intermediate circuit representation because they translate each program statement into a circuit which encodes the bit-precise semantics of the computation the statement performs. During the generation of the intermediate circuit, structural hashing, and constant propagation are performed to keep the resulting circuit size small. To generate the intermediate representation of the circuit, CBMC-GC uses the ABC framework [SG] as it provides state-of-the-art logic synthesis methods. After the generation of an intermediate circuit, a pattern based sub circuit rewriting is performed in a repeated manner in combination with structural hashing, constant propagation, and a simplified version of SAT-sweeping [Kue04]. When the source code is compiled with CBMC-GC, a description of the circuit performing the computation, and a mapping between the input and output identifiers and the corresponding circuit wire is obtained.

2.6.3 Compiling Low Depth Circuits

Most of the compilers for STC focused on creating size-minimal circuits for Yao's Garbled Protocol (Section 2.4). However, MPC protocols such as the GMW protocol (Section 2.5), have round complexity that is dependent on the circuit depth. The round complexity becomes a considerable bottleneck when these protocols are deployed in the real world network settings, as the network latencies are usually in the range of tens or hundreds of milliseconds.

ShallowCC [BHWK16] is a compiler extension for CBMC-GC that creates depth-minimized Boolean circuits from ANSI C also while keeping the size minimal. It introduces optimized building blocks that are up to 50% shallower than the previous constructions. It also implements multiple high-level and low-level depth minimization techniques.

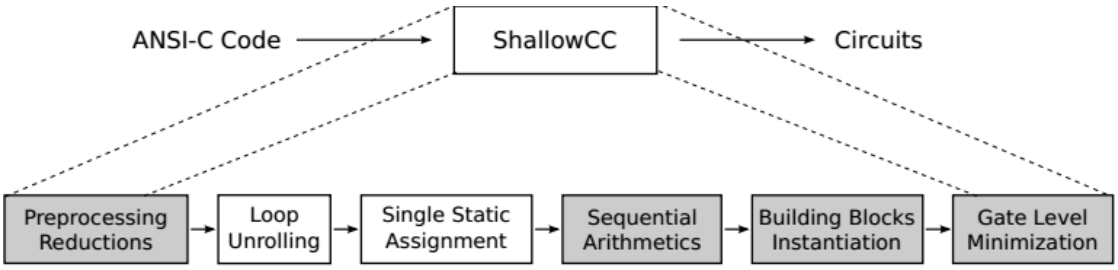

Figure 2.4: ShallowCC's compilation chain from ANSI-C to Boolean circuits [BHWK16]

ShallowCC adopts the CBMC-GC's compilation approach for depth-minimization as illustrated in Figure 2.4. The parts marked in gray are adaptations and extensions. The compiler needs the input arguments and output variables in an ANSI C code to follow a particular naming convention. In the first step, the code is preprocessed to identify and transform the reduction statements on the source code level. In the second and third steps, all loops and function recursions are unrolled using symbolic execution and the resulting code is transformed into Single Static Assignment (SSA) form (Section 2.6.2.2). In the fourth step, the SSA form (Section 2.6.2.3) is utilized to detect and annotate successive composition of arithmetic statements. In the fifth step, all statements are instantiated with hand-optimized building blocks. In the final step, gate-level minimization takes place (Section 2.6.2.5).

ShallowCC is capable of compiling circuits that are up to 2.5 times shallower than the hand optimized circuits and up to 400 times shallower than the circuits compiled from size optimizing compilers, while as yet keeping up a competitive circuit size. The evaluation results on two examples, biometric matching with 1024 bits and matrix multiplication with 5x5 bit elements can be seen in Table 2.3. We can observe that the depth in biometric matching improved by 98.7% while the depth in matrix multiplication improved by 59.7%.

Functionality	**CBMC-GC**		**ShallowCC**		**Improvements**
	size	**depth**	**size**	**depth**	**depth**
Biometric matching 1024	2.9 M	**7,181**	2.9 M	**90**	**98.7%**
Matrix multiplication 5x5	127,225	**42**	128,225	**17**	**59.7%**

Table 2.3: Gate-level minimization limit: 600 seconds [BHWK16]

2.6.4 Optimized Building Blocks

When designing complex Boolean circuits, optimizations play a significant role. Optimized building blocks speed up compilation as they just need to be highly optimized once and can be instantiated at compile time with low cost. We discuss a few examples optimized building blocks that are used in this work.

2.6.4.1 Adder/Subtractor

An adder is a Boolean circuit that can perform the addition of two numbers. There are two types of adders: half-adder and full-adder. A half adder has two inputs A and B, and two outputs sum S and carry C. The sum is the XOR the AND of the inputs. A full adder candders. A full adder has three inputs A, B and a carry input C_{in}, and two outputs sum S and carry output C_{out}. First, a half adder would be used to calculate the sum of A and B; then the second half adder would be used to add C_{in} to the sum calculated by the half adder. C_{out} will be an OR function of the half adder carry outputs, just in case a carry is produced. Parallel Prefix Adders (PPA) are used in logic synthesis to accomplish speedier addition under size trade-offs by using a tree-based prefix network with logarithmic depth. Beuscher et al. [BHWK16] proposed a design for PPA which allows halving the number of non-linear gates in every aggregation layer of a PPA. They applied the design to Skylansy adder which is known to have the lowest depth in all PPAs [Har03] and achieved a depth of $[log(n)] + 1$ and a size of $n[log(n)]$, for e.g., the size and depth of a 16 bits adder is 64 and 5, for a 32 bits adder is 160 and 6, and for a 64 bits adder is 384 and 7 respectively. They also applied this design to Brent-Kung adder [Har03] which is an alternate to Skylansy adder. The Brent-Kung adder exhibited a trade-off between size and depth with a depth of $2[log(n)] - 1$ and size of $3n$.

Since the subtractor can be designed using the same approach as of an adder, it benefits to the same extent from optimized addition.

2.6.4.2 Equivalence Comparator

An equivalence comparator (EQ) checks if two input bit strings of length n are equal and outputs a single bit result. It can be implemented by a successive OR composition over pairwise XOR gates that compare single bits. This results in a size of $s^{nX}{}_{EQ}(n) = n - 1$ gates [KS08].

2.6.5 Compiler Comparision

In this section, we discuss different properties of the compilers. Table 2.4 which is taken from [BFH+17] shows the comparison of the compilers.

Compiler	Language	Interpreted	Mix-mode	Maturity	Const. prop.	Gate level opt.
CBMC-GC'12 [HFKV12]	ANSI-C	No	No	Yes	Global	Dead gate elimination Constant propagation Theorem rewriting SAT sweeping
KSS'12 [KSS12]	DSL	No	No	No	No	Constant propagation Dead gate elimination
PCF'13 [KSMB13]	ANSI-C	Yes	No	No	Limited	Local constant propagation, Dead gate elimination
ObliVM'15 [LWN+15]	DSL	Yes	Yes	No	No	No
OblivC'15 [ZE15]	DSL ANSI-C	Yes	Yes	No	Local	Local and limited [MGC+16] constant propagation, Dead gate elimination
TinyGarble'15 [SHS+15]	Verilog VHDL	No	No	Yes	n/a	Dead gate elimination Constant propagation Rewriting SAT sweeping
Frigate'16 [MGC+16]	DSL	No	No	Yes	Local	Local constant propagation, Dead gate elimination

Table 2.4: Comparision of different compilers [BFH+17]

In the table, we can see that CBMC-GC, KSS, TinyGarble and Frigate are integrated compilers (Section 2.6.1.1) while PCF, ObliVM and OblivC are interpreted compilers (Section 2.6.1.2). Only CBMC-GC, PCF, and OblivC support ANSI-C while others support domain-specific languages. All of them use dead gate elimination as an optimization technique which is a method of removing gates which do not affect the output values. They also use constant propagation to propagate values. Also, the table shows the compiler correctness (maturity) which is based on the study by Mood et al. [MGC+16], along with the applied source and gate-level optimization techniques. Mood et al. [MGC+16] identified compilation aborts with the input/output notation that have been fixed in the current version of CBMC-GC. PCF uses a frontend compiler LCC that generates optimized bytecode for RAM-based architectures. TinyGarble and Frigate store sequential building blocks in a compact way.

2.7 ABY Framework

Demmler et al. [DSZ15] created a framework for efficient mixed-protocol secure two-party computation called ABY. The framework allows two parties to compute a function using sensitive data while preserving the privacy of the data. The framework combines secure computation schemes based on Yao's Garbled circuit protocol (Section 2.4) along with Arithmetic sharing and Boolean sharing (Section 2.5) but for our work we have used the ones based on Boolean circuits.

2.7.1 Basic Definitions

In this section, we explain the terminologies that are relevant for our work. The content is based on the developers guide of the ABY framework [DSZ15].

Sharings `Sharings` in the ABY context means the secure computation techniques that are used to preserve the privacy of the data. ABY has three types of available sharings: Arithmetic sharing (*S_ARITH*), Boolean sharing using the GMW protocol (*S_BOOL*), and Yao's sharing (*S_YAO*) based on Yao's Garbled protocol. We use the GMW protocol (*S_BOOL*) for evaluating our circuit securely.

Gates and Circuits ABY has a circuit object named Circuit; two sub-classes are derived from this object called ArithmeticCircuit and BooleanCircuit. In our case, only BooleanCircuit is used.
PutINGate and PutOUTGate can be used by the two parties to secret share their plaintext input and reconstruct the plaintext output, respectively.

Wires In ABY a wire is uniquely identified by a wire ID which is a global identifier of type `uint32_t`.

Shares A `share` object is used to bundle one or multiple wires which helps in simplifying the design of the circuit. In ABY, a share is a variable which can be passed to gates to perform operations and can also hold the output of gate operation. A `share` object holds the plaintext values returned by the output gate after the execution of the secure computation protocol.

A `share` object stores an array of `uint32_t` wire IDs and the length of this array is denoted by `bitlength` of the share.

SIMD Gates Single Instruction Multiple Data (SIMD) allows the same operation to be applied to multiple data items simultaneously. In secure-computation, SIMD gates can be used to improve the circuit evaluation time. They also reduce the memory footprint of the circuit.
A `share` is normally a single dimension array. SIMD gates allow multiple elements to be stored on a wire which extends the share to a second dimension.

2.7.2 Setting up ABY

An instance of ABYParty class has to be generated for ABY to work. The ABYParty object can then be used to start the secure computation process and also for defining the functionality. A `Circuit` object of a specific type sharing, i.e., `BooleanCircuit` in our example is needed to define this functionality, which can be obtained from the ABYParty object. `ExecCircuit()` function of the ABYParty object needs to be called to run the secure computation protocol, the function can be called once the functionality has been defined.

2.7.3 Necessary Gate Types for our Implementation

In this section we introduce the input/output gates required by our framework. We also discuss a few function gates from ABY which are relevant to our work. Function gates compute on secret shared values. We introduce a few function gates that are supported by ABY for Boolean circuits.

2.7.3.1 PutINGate

To load the plaintext inputs of the computing parties to their shares, a method called PutINGate is used. The method returns a share object holding a share or encryption of plaintext.

The method has three parameters: `val`, `bitlen`, and `role`. Parameter `val` is the input value which is loaded by one of the parties to generate the shared secret. This variable can be of type `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` or a pointer to an array of such datatype. The second parameter is `bitlen` which specifies the length of the plaintext, i.e., how many bits of inputs should be read. It also specifies the number of wires in the generated share. The last parameter is `role` which defines that which party provides the input of this share. It can only have two values, i.e., CLIENT or SERVER.

2.7.3.2 PutSIMDINGate

To allow the creation of SIMD circuits, the PutINGate is replaced by PutSIMDINGate. It has an extra parameter $nval$. The nval variable helps to define the number of variables to be shared on a wire. The method returns a share object which can be used equivalently to a non-SIMD share. This object encapsulates the shared secrets.

The PutSIMDINGate method also has three parameters, but unlike PutINGate (Section 2.7.3.1) its first parameter is $nvals$ which specifies the number of SIMD elements to be stored on a single wire. The other three parameters val, bitlen and role serve the same function as defined earlier (Section 2.7.3.1). However, a call to PutSIMDINGate with $nvals = 1$ is equivalent to the calling PutINGate.

2.7.3.3 PutOUTGate

The PutOUTGate method is used for the decryption of a share in an interactive manner or to reconstruct a shared-value to plaintext and store the result in a share. The PutOUTGate has two parameters: s_out and $role$. The s_out parameter is the share object which would hold the result after the evaluation of the circuit. The role parameter defines which party can see the output; it can have one of the three values CLIENT, SERVER, or ALL.

2.7.3.4 PutANDGate

The PutANDGate method performs a bit-wise AND operation on the two input shares and returns a share object of the same bitlength but longer input share as a result.

2.7.3.5 PutORGate

The PutORGate method performs a bit-wise OR operation on the two input shares and returns a share object of the same bitlength but longer input share as a result.

2.7.3.6 PutXORGate

The PutXORGate method performs a bit-wise XOR operation on the two input shares and returns a share object of the same bitlength but longer input share as a result.

3 Secure Matching

In this chapter, we describe problems related to stable matching, such as secure stable matching and how it has been implemented for STC by different frameworks. The frameworks by Doerner et al. [DES16] and Riazi et al. [RSS+17] have been discussed in detail as they are currently the most optimized frameworks for secure stable matching.

3.1 Matching Problems

In this section we discuss matching problems in general as we are solving an assignment problem. There are many available matching problems such as stable roommates problem and Nash equilibrium, and so forth. The stable matching problem is closest to the underlying problem of this thesis, because it works with two sets of the same size and also deals with matching pairs. Also, the Hungarian algorithm can be reduced to Weighted Bipartite Matching Problem [KMV91].

3.1.0.1 Stable Matching Problem

The stable matching problem is a problem to find a stable match between two equally sized sets A and B. The two sets which are required by the algorithm can be considered two parties. Both the sets should be of the same size which is denoted by n so that each participant (element in the set) would have a match at the end of the execution. Members of set A are called proposers while members of set B are called reviewers. The match is based on the preferences of the elements and is not stable if some element x of set A prefers an element y of set B over the currently assigned element of set B and the element y also prefers element x over its current assigned element [GS13]. The stable matching algorithm is used by many organizations, some famous examples include, college admissions and in supply chain management that is matching of suppliers and consumers, but they currently rely on third parties to process sensitive data. Since STC revolves around the concept of processing information securely without the involvement of a trusted third party, so we recapitulate some of the secure stable matching techniques.

3.1.0.2 Stable Roommates Problem

The stable-roommate problem is a problem to find a stable match between a set of size s where s is an even number. The matching creates n disjoint set of pairs of roommates where $n = s/2$. The match is stable if there are no two participants that are not roommates and prefer each other over their assigned partners.

3.1.0.3 Nash Equilibrium

The Nash equilibrium is amongst the fundamental concepts of game theory. Two players P_1 and P_2 are in Nash equilibrium if P_1 takes the best decision he can, taking into account the decision of P_2 which remains unchanged and P_2 takes the best decision he can, taking into account the decision of P_1 which remains unchanged. The same algorithm can also be applied to a group of players; all players are in Nash equilibrium if they take decisions while taking into account the decisions of other players which remain unchanged.

3.1.1 Secure Stable Matching

Golle [Gol06] created a privacy-preserving version for the classic stable matching algorithm (Gale-Shapley Algorithm) [GS13] where the matching protocol is performed by an honest group of matching authorities. He argued that the generic MPC protocols are too impractical to implement a complex algorithm like Gale-Shapley. The protocol requires $O(n^5)$ asymmetric key operations, and it has never been practically implemented. Franklin et al. [FGM07] identified cases where the Golle protocol would fail and proposed two of their own protocols one of which was based on Golle's protocol and relies on XOR secret sharing scheme and requires $O(n^4)$ public key operations and the other one uses garbled circuits in combination with Naor-Nissim's protocol for SFE [NN01] which had a complexity of $O(n^4)$. Both of these protocols have $O(n^2)$ communication rounds.

Keller et al. [KS14] were the first to propose implementing the algorithm by using ORAM and garbled circuits. Oblivious RAM (ORAM) allows the execution of a RAM program while hiding the access pattern to the memory. They reported matching 128 x 128 participants in around 2.5 hours but did not include 1000 processor-days of the offline compute time. Zahur et al. [ZWR+16] used ORAM to implement the book version of Gale-Shapley and took over 33 hours to compute the matching of 512 x 512 participants. Blanton et al. [BSA13] proposed a secure but complex construction of Gale-Shapley using Breadth First Search (BFS). Their idea was to permute the rows and column of an adjacency matrix. BFS allowed the algorithm to iterate over a whole column of the adjacency matrix at once, but the algorithm originally has to shift between the proposers which is why this was not beneficial.

3.1.1.1 Scalable Secure Stable Matching

Doerner et al. [DES16] adapted the classic stable matching algorithm (Gale-Shapley Algorithm) [GS13] in the context of secure two-party computation. The two input sets A and B are lists containing preference of each participant which can be divided amongst the two parties either by partitioning or by XOR-sharing (XORing the inputs that no party would be able to see the preference).

The algorithm iterates through the list of proposers based on their preference and swaps them when the match is validated or invalidated because of other matches. The cost of iterating over potential pairings is at most n^2, but since it cannot be determined in advance that which proposer's preference list will be evaluated, so they store the results of these pairing in an ORAM which helps to reduce the overall cost of reading the preferences. This implementation requires n^2 accesses to an ORAM of size n^2.

Since the secure implementation of the algorithm requires the proposer lists to be selected obliviously rather than in-order like in the original algorithm, they design oblivious linked multi-list to be able to iterate through n separate arrays while hiding which component of the array is being iterated. The algorithm also uses the fastest available implementations of square-root ORAM, circuit ORAM and oblivious queue construction from Zahur et al. [ZWR+16]. It modifies these queues to stop allocation of dynamic layers by introducing a public size bound. Their implementation for the 512 x 512 participants matching took around 48 minutes which was 40 times faster than that of Zahur et al. [ZWR+16] which was over 33 hours.

Riazi et al. [RSS+17] proposed the first provably secure and scalable implementation of stable matching by using Yao's garbled circuit protocol along with ORAM. They introduced sub-linear circuit size which can scale to high set sizes and early termination technique (ETT) which provides efficiency/privacy trade-off. Since random access is a very costly operation in GC with linear access cost when multiplexers are used with flip-flops, so they have used ORAM which has logarithmic access cost.

They start by generating a netlist offline with describing the functionality as a Boolean circuit, they have written the stable matching algorithm in a hardware description language (Verilog), and then they fit it into a hardware synthesis tool to get the netlist. Then they convert it into an ordered netlist by sorting it into a topological order to avoid any deadlocks. The ordered netlist is then passed as an input to the GC protocol. The GC protocol (Section 2.4) is then run to produce a stable matched list as an output. The worst case scenario excluding the invalid proposals in the GC is $O(n^2)$. In the sub-linear size circuit, the sub-linearity is concerning the number of participants in each group. They have a sub-module named algorithm combinational circuit which acts as a control flow of the whole circuit and efficiently implements the stable matching algorithm then they have a main selection circuit which finds a free participant (that is not matched yet) in the previous clock-cycle and fir it to the algorithm combinational circuit. They also have a memory module which can be implemented as ORAM. Since in the garbled protocol it is not possible to detect if a match is stable or

not, the ETT protocol helps to detect a stable match between the process by revealing one bit of information between each clock-cycle. It is an optional protocol because it comes at the cost of revealing the total number of proposal needed for a stable match. For a 512 x 512 participants problem this protocol takes 8 hours of execution time without using ETT and around 5 minutes execution time with using ETT. As compared to Doerner et al. [DES16] the standard execution without ETT takes more time, but the execution with ETT is faster.

4 Privacy-Preserving Assignments

In this chapter, we explain our design choices for a privacy-preserving assignments framework. We develop a framework which solves the assignment problem in a privacy-preserving manner. We start by giving an overview of the architecture of our solution and move on to explain the Hungarian algorithm which is used by our framework to solve the assignment problem.

4.1 Overview

We focus on solving the assignment problem which is one of the most fundamental combinatorial optimization problems in the field of computer science and mathematics, without using a trusted third-party in a privacy-preserving manner. We use the Hungarian algorithm [Kuh10] to be able to solve the assignment problem in polynomial time. In this problem, there are n workers and n jobs. There is a cost associated with every worker-task pair, i.e., the cost required for a worker i to perform task j. The idea is to minimize the total cost. Each task should be assigned to one worker, and one worker should only have one assigned task. We utilize secure computation techniques to compute the results and help the workers get their preferred job without revealing their preferences.

The concrete application example that we aim to realize is privacy-preserving assignments of tasks for students based on their priority lists. This means that the student is presented with a list of topics and he prioritizes them according to his preference. In the assignment we make sure that the happiness of the students should be maximized, i.e., every student gets a topic according to his preference, while keeping the overall cost of the assignments minimum. Currently, there are systems in place that solve assignment problems, such as The National Resident Matching Program (NRMP) where applicants are placed into residency and fellowship positions based on their preference list, but they require a trusted third party to compute the results for them. Our system would guarantee that no party learns about the input of other parties and the whole computation is executed by ensuring the privacy of the data. Currently, to achieve the goal of computing privacy-preserving assignments securely one needs to use multiple frameworks, or the intervention of a third-party is required.

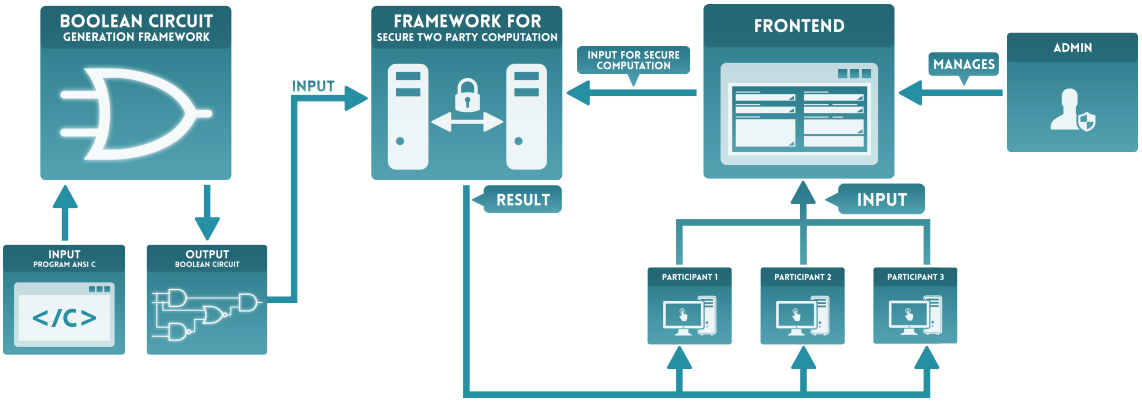

Figure 4.1: Global view for privacy-preserving assignments

Figure 4.1 illustrates our system design. This is a distributed system which incorporates three modules. The first module is the front-end which takes the inputs from the user, encrypts them using the keys of the computing parties on the user's end and sends it to the STC module. The front-end is used to manage all the administrative and user actions. Second is the Boolean circuit generation module which is needed to convert the Hungarian algorithm to a Boolean circuit. The last is an STC module which uses the outputs of the first two modules as an input and produces a result without compromising the privacy.

The front-end server helps with the management of the framework. It calculates the number of participating users; the number serves as an input for the creation of Boolean circuits. It also collects user inputs and then sends them to the STC modules. It also makes it easier for the participants to encrypt their input and send it to the computing parties without the use of any extra utility.

In STC we either represent a function as a Boolean circuit or as an arithmetic circuit. For our framework, we chose Boolean circuits because our computation included non-arithmetic operations such as comparison which if implemented as an arithmetic circuit would be very inefficient. Also concerning depth, Boolean circuits are exponentially more dynamic than arithmetic circuits [GS91].

We use two parties for this computation. Since there is no single party involved in the computation, it increases the trust factor that the computation can never be influenced. Any computing party would never have the input in clear even after the decryption it would hold a XORed input which would not reveal anything about the original input. We assume that both parties would never communicate to view the original input and this is what makes the protocol secure.

4.1.1 System Flow

Consider the above described example of a seminar course where n topics are being offered for n students.

Step 1: Problem setup. First, the initialization phase takes place where the administrator uses our front-end and sets a limit for the maximum number of participants. He then adds all the topics which are being offered. He shares the link with the students so that they can submit their choices.

Step 2: Providing inputs. Next, the input sharing phase starts when a student visits the link, he sees the list of topics being offered along with the fields where he needs to enter his data. He arranges the topics according to his preference, and after filling the form, he submits it. When he clicks submit, the front-end module checks if the inputs are valid. The module generates a random number and XORs it with the preference of the user. The public-keys of the two parties that would be securely computing the assignments are then used to encrypt the data on the client end. The random number gets encrypted with public key of the first computing party, and the XORed input gets encrypted with public key of the second computing party before being sent to the front-end module.

Step 3: Circuit generation. Once we have the value of n from the front-end which is the total number of participants, we can proceed to the circuit generation phase where we generate a Boolean circuit for the Hungarian Algorithm by using the Boolean circuit generation module or use a pre-generated Boolean circuit. Once the deadline for the topic selection is over, the front-end sends the encrypted inputs to both the parties.

Step 4: Computation. In the secure computation phase which is also the final phase, the computation is then triggered by the administrator who has access to the two computing servers and can start the computation. The Boolean circuit is used as an input for the STC module. The computing parties in the STC module already have the inputs from the users, and those inputs would be used as the inputs for the secure computation. The STC module then computes the result and sends them back to the administrator who then sends it to the participants. This way, every student ends up with a topic of his choice without revealing his preferences to anyone else.

4.1.2 Front-end

Our idea is to have our front-end based on a web framework. We have two modules in the front-end: one for the administrator and the other one for the participants. The front-end consists of an administrative panel where the administrator can set a limit for the number of participants he wants to allow. Also, the module allows the administrator to add and remove topics as needed. The other module is of the participants. When a participant visits the portal, a page is displayed to him. He sees the list of available topics for the seminar and then arranges the topics according to his priority and along with his personal information submits

the data to the server. The preferences (inputs) are then XOR-shared before getting encrypted on the participant's machine by using the public-keys of the two computing parties and then sent to the two-parties. The XOR-sharing makes sure that none of the two computing parties can see the inputs in clear (such as one-time-pad encryption), while the public key encryption makes sure that the front-end server does not see both shares of the XOR-sharing, from which he could retrieve the result. The computing parties can decrypt their shares that were encrypted using their public keys since they own the private keys.

4.1.3 Circuit Generation Module

We take an implementation of the Hungarian algorithm [Cde14] in ANSI C and adapt it according to an available circuit generation module. The module would then take the ANSI C program as an input and generate a corresponding Boolean circuit. Since the bound for the loops in the program have to be known before the circuit generation, we need to specify a bound for the loops inside the program to avoid infinite loops in the circuit generation process and for the circuit to be generated correctly. The Boolean circuits could be pre-generated or triggered by the front end.

4.1.4 STC Module

We use an STC module which would help us achieve the privacy-preserving assignments. The module takes as an input the Boolean circuit which is generated by the circuit generation module. It also takes the encrypted inputs from the participants, which were sent to the two computing parties by the front-end. The inputs are first decrypted using the respective private keys and then used in the circuit for computing the assignment securely. Once the module completes the circuit computation, it will then send the output results to the participants without breaching their privacy. The overall security depends on the security of the STC module, i.e., either malicious security or semi-honest security, depends on the STC module we use.

4.2 Hungarian Algorithm

The Hungarian algorithm [Kuh10] which is also known as the Kuhn-Munkres algorithm is a combinatorial optimization problem which solves the assignment problem in polynomial time. The best-known complexity for the Hungarian algorithm is $\mathcal{O}(n^3)$ but the worst case running time of our implementation is $\mathcal{O}(n^4)$. We take inputs as a cost matrix and then apply row operations on them to create zeros, where zero means a possible assignment in our case. Only one assignment is possible in each row and column. Lines are drawn horizontally and vertically to cover zeros with the minimum lines possible. If the lines are equal to the size of the matrix, we have an optimal solution. Figure 4.2 shows the flow of the

algorithm. The algorithm can be divided into seven steps, and all the operations are applied to a cost matrix M' which is a copy of the original cost matrix M. Now we explain each step in detail.

Step 1: Row minimum. In this step, we calculate the minimum value in a row and subtract it from all the elements in the row; this is done for all rows.

Step 2: Column minimum. In this step, we calculate the minimum value in a column and subtract that value from each element in the column; this is done for all columns.

Step 3: Check for optimal solution. In this step, we check if we already have an optimal solution, i.e., the number of selected zeros is equal to the size of the matrix. If we have an optimal solution we move to step 7, else we move to step 4.

Step 4: Mark selected zeros. In this step, we mark zeros which are not yet marked and their rows and columns are not yet selected and then we move to step 6 to create additional zeros. We move to step 5 if no further entries can be marked.

Step 5: Create augmenting path. In this step, we mark all rows for assignment and move to step 3 to check for an optimal solution.

Step 6: Create additional zeros. In this step, we select the smallest element that is not covered by any line. We subtract it from all uncovered elements and add it to all the elements covered twice.

Step 7: Display optimal solution. In this step, the optimal solution is displayed on the original cost matrix.

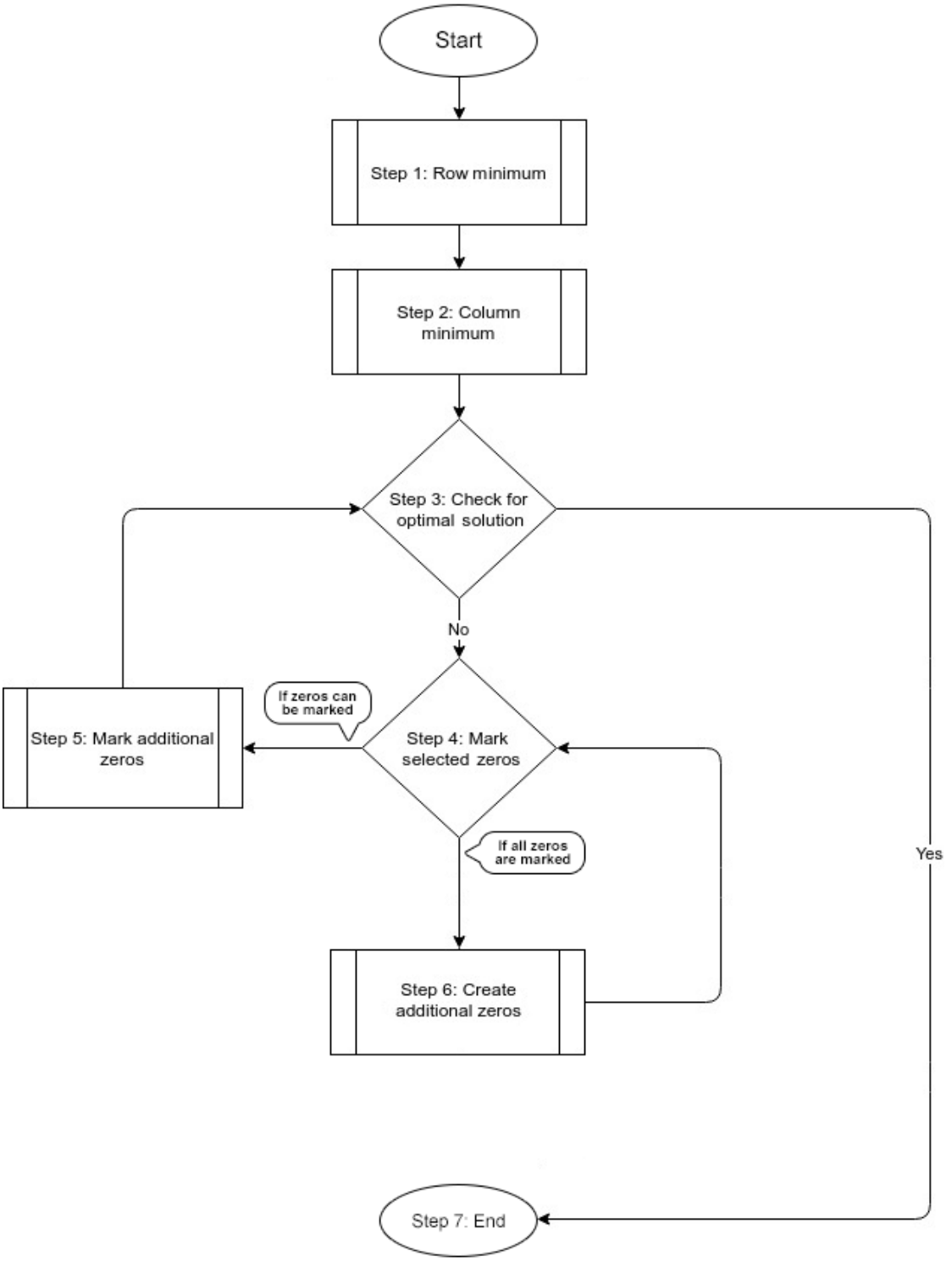

Figure 4.2: Hungarian Algorithm

4.2.1 Example

Suppose we have three workers $\{w_1, w_2, w_3\}$ and three tasks $\{t_1, t_2, t_3\}$. Given below is a cost matrix M. Each pair (w_i, t_j) shows the cost of worker i to perform a task j where i and j are in $\{1, 2, 3\}$.

$$M = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 77 & 13 & 88 \\ 31 & 11 & 94 \\ 15 & 33 & 71 \end{pmatrix} \end{array}$$

In step one, we start with calculating row minimum, which is the lowest value in each row. Once we have the row minimum, we subtract it from each row to get zero values.

$$M' = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 64 & 0 & 75 \\ 20 & 0 & 83 \\ 0 & 18 & 56 \end{pmatrix} \end{array} \begin{array}{l} \\ (-13) \\ (-11) \\ (-15) \end{array}$$

In step two, we do the same with columns, i.e., we calculate the lowest value in each column and subtract the whole column with that value.

$$M' = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 64 & 0 & 19 \\ 20 & 0 & 27 \\ 0 & 18 & 0 \end{pmatrix} \\ \begin{array}{ccc} (-0) & (-0) & (-56) \end{array} \end{array}$$

In step three, we look for the minimum number of lines that need to be drawn in order to cover all zeros. We tick a column or a row in order to achieve this. If the number of lines needed to cover the zeros is equal to the size of the matrix then we have an optimal solution. If the lines are less than the size of the matrix then we move to the next step.

$$M' = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 64 & 0 & 19 \\ 20 & 0 & 27 \\ 0 & 18 & 0 \end{pmatrix} \\ \begin{array}{ccc} & \checkmark & \end{array} \end{array} \begin{array}{l} \\ \\ \\ \checkmark \\ \\ \end{array}$$

Now we combine step four, five and six for the next three steps. Since the number of lines required to cover the zeros was two which is less than the size of the matrix we create additional zeros to reach an optimal solution. We check for the smallest element which is not ticked and subtract it from all the unticked elements and add it to all the elements that are ticked twice, i.e., an element whose both row and column are ticked. The smallest element,

in this case, is 19, so we subtract 19 from all the unticked elements of the matrix and add it to all the elements that are ticked twice.

$$M' = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 45 & 0 & 0 \\ 1 & 0 & 8 \\ 0 & 37 & 0 \end{pmatrix} \end{array}$$

We move back to the step 3 to find an optimal solution by ticking the rows and columns. Here the number of lines required to cover all zeros is 3 which is equal to the size of the matrix which means we have an optimal solution.

$$M' = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 45 & 0 & 0 \\ 1 & 0 & 8 \\ 0 & 37 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ \checkmark \\ \checkmark \\ \checkmark \end{array}$$

We select the zeros which would be the cost of an assignment. We need to select zeros in such a way that it should be the only selected zero in the whole row and column which means that one worker is assigned to one task and no task is assigned to multiple workers. First, we move to the row of worker w_2 because it has a single zero entry which means it is the only assignment possible. After we mark the task t_2 as assigned to worker w_2 we can ignore all zero values in the task t_2 column because the task cannot be assigned to someone else. We move to worker w_1 where we have two zero values which means two possible assignments. Since the task t_2, is already assigned, so we move to the next zero which is task t_3 which gets assigned to worker w_1. The last assignment is for worker w_3, we check the first zero value and select it as the second zero value is in the column of task t_3 which is already assigned to worker w_1.

$$M' = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 45 & 0 & \boxed{0} \\ 1 & \boxed{0} & 8 \\ \boxed{0} & 37 & 0 \end{pmatrix} \end{array}$$

In the final step, we replace the matrix M' with the original cost matrix M. The position of the assignments remain the same. The cost to perform the tasks $\{t_1, t_2, t_3\}$ is $\{15, 11, 88\}$, which, as we can observe, is the optimal assignment.

$$M = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 77 & 13 & \boxed{88} \\ 31 & \boxed{11} & 94 \\ \boxed{15} & 33 & 71 \end{pmatrix} \end{array}$$

5 Implementation

This section covers all the details about the implementation of our framework. We used PHP, Javascript, HTML, and CSS for the front-end framework. For the circuit generation we used CBMC - GC (Section 2.6.2), and for the STC part, we used ABY [DSZ15]. This section contains all the technical details of our system and it also covers the complexity analysis of our implementation of the Hungarian algorithm. Later in the section, we describe how we have used ABY to run our computation securely.

5.1 Technical Details of the Front-end

The front-end uses PHP for its server-side and HTML, CSS for the design. We use RSA for the encryption of user data. The large key size in RSA is essential for increasing the security of the algorithm. We encrypt the user input in RSA-2048 encryption by making use of a Javascript framework called jsencrypt [Tra13] because to ensure security it is not recommended to implement an existing cryptographIC algorithm, and the use of libraries is always encouraged. The library jsencrypt is a wrapper which takes public and private key pairs generated by OpenSSL in PEM format and translates it to the variables required by the jsbn library [Wu09]. Jsenrcypt heavily relies on jsbn which is written by Tom Wu. We have divided the front-end functionalities into two types which are further discussed below.

5.1.1 User Panel

In this section, we discuss the functions of the users. The user will always visit this home page where all the available topics for the seminar would be displayed. The second step is where the user will fill the form according to his preference. In case, if the user assigns the same priority to more than one topic, an error would be displayed which can be seen in Figure 5.1. If the submission is successful, then a message would be displayed to the user as shown in Figure 5.2. If the number of allowed participants are already filled, then an error would be displayed to the user.

The home page fetches all the topics stored in our system and displays them to the user. After the form is filled by the user, a validation runs which ensures that no two topics are assigned the same priority. After validation, the preference of the user is encrypted on the browser using RSA-2048 encryption and is shared with the computing parties using secret

sharing. A random number is generated for every choice and is XORed by the priority. The new number and the random number are then encrypted separately on the user's machine by using the public key's of two computing parties A and B respectively to avoid the system from getting inputs in plain. The encrypted preference list along with the user data is then sent to the server. A success message or an error message is returned depending if the limit for the maximum users is reached or not. The secret sharing and encryption helps us to never transfer data in plain and also ensures that no party would have the full-data access preserve the privacy of the user.

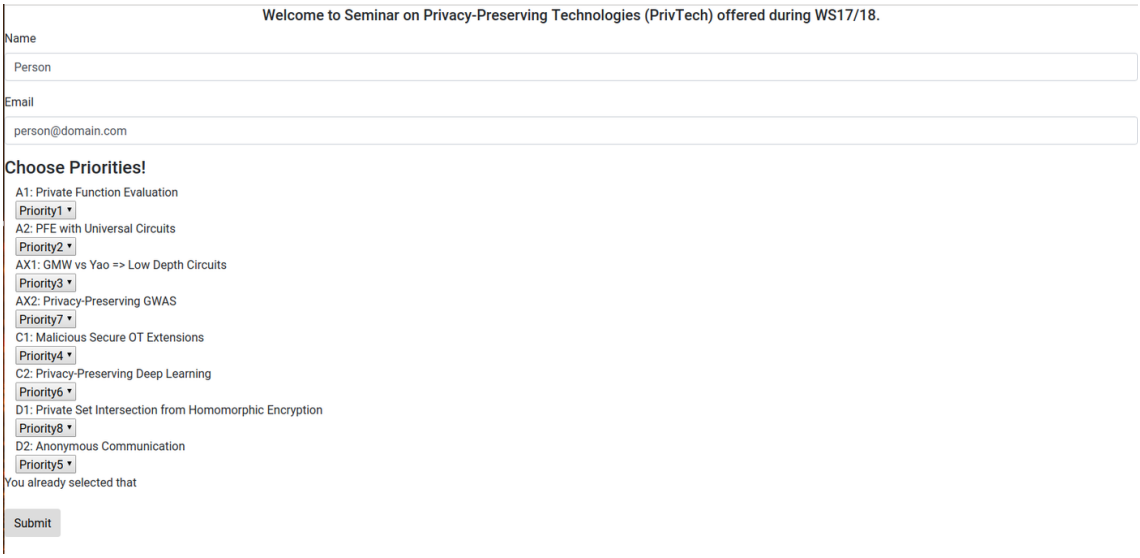

Figure 5.1: Duplicate Priority Detection

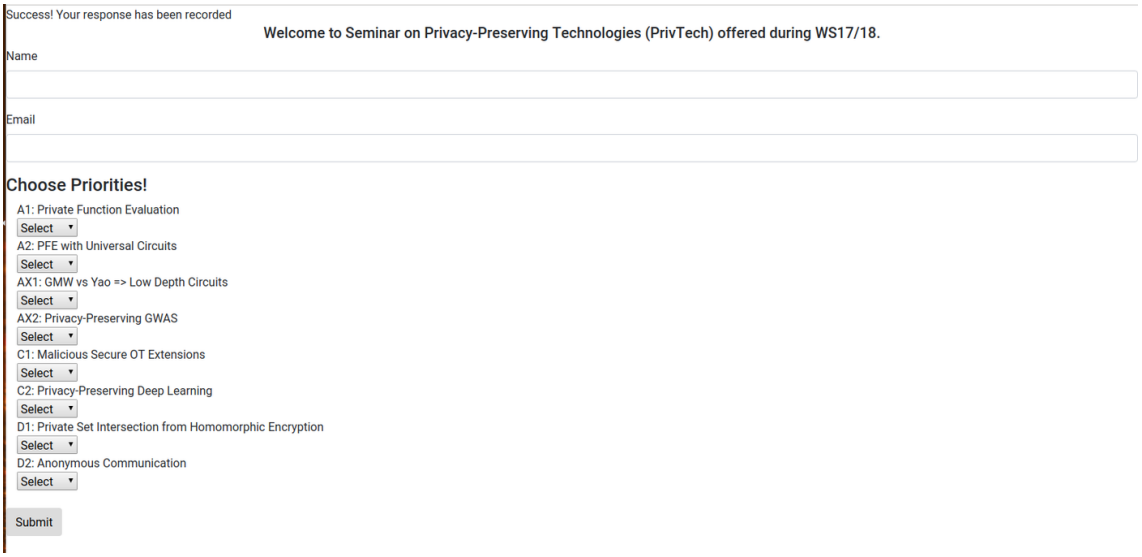

Figure 5.2: Successful Submission

5.1.2 Administrator Panel

In this section, we discuss the administrative functionalities. Figure 5.3 shows the entry point of the administrator. The administrator needs a password to log in to the system which is communicated via direct communication but could be easily modified to a symmetric key encryption for password sharing. As manually hashing the password and verifying it can be hectic and not secure, the password is stored as a hash in the system by using *password_hash* which generates a unique salted hash and is verified using *password_verify* which are the built-in functions provided by PHP. These functions allow hashing and securely verifying the passwords. After logging in, the administrator has access to four functionalities which are pre-defined in the system. He can configure the limit for the number of users, which is a number that gets set in the system and is checked against to allow/deny participants. Add and remove topics for the seminar and set public-keys for the computing parties in which the old keys get replaced by the specified keys, this is illustrated in Figure 5.4. In Figure 5.5 and Figure 5.6 we can see how an administrator can change the number of allowed users and the message that is displayed on an update. Figure 5.7 shows the functionality of adding topics to the system. All the added topics are written to the system. In Figure 5.8, we can see the list of available topics and select the topic we no longer need. After a topic is removed from the system, the list is updated, and the removed topic is displayed, as shown in Figure 5.9. Figure 5.10 and Figure 5.11 illustrate the mechanism of changing keys of the computing parties. The computing parties can always change for different runs of the system, i.e., if the system is deployed for another course. This functionality makes it easier for the administrators to update the information with one click.

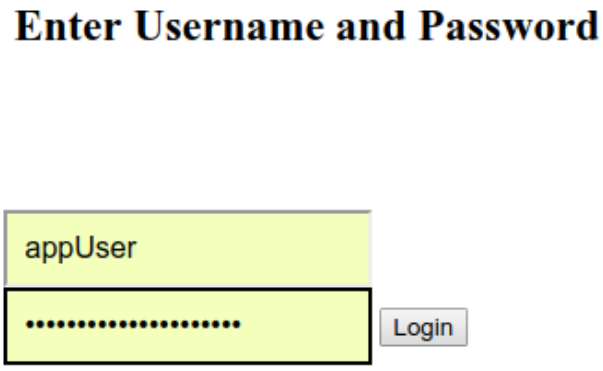

Figure 5.3: Administrator Login

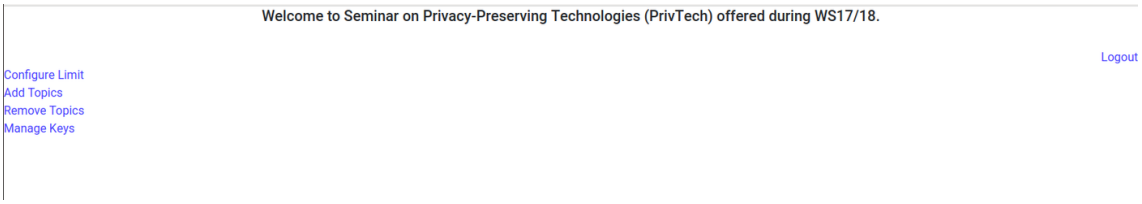

Figure 5.4: Administrator Panel

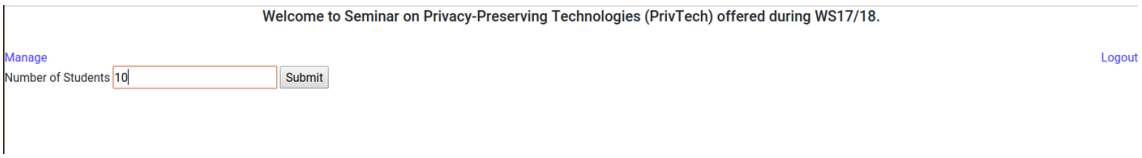

Figure 5.5: Set number of users

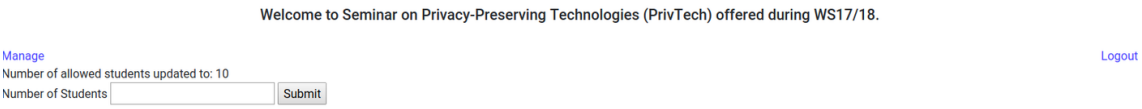

Figure 5.6: Updated number of users

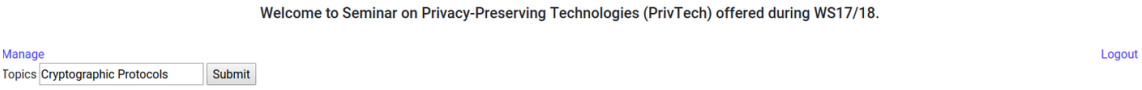

Figure 5.7: Add topics

Figure 5.8: Remove topics

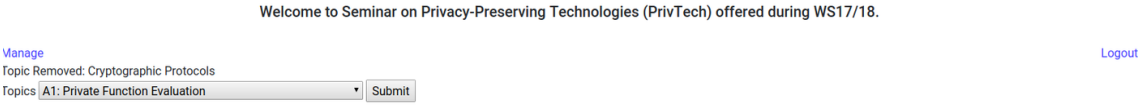

Figure 5.9: Topic removed

Figure 5.10: Update Public Keys

Figure 5.11: Public Keys Updated

5.2 Circuit Generation in CBMC - GC

We have optimized an existing implementation of the Hungarian algorithm in C [Cde14]. The original program was created for a rectangular matrix, and we have changed it for our use case which is a square matrix. We have optimized it for size by using *unsigned char* which is 8 bits instead of *int* 32 bits. In step 1 and step 6 we have reduced the size of *minVal* from *INFINITY* to *SIZE+1* which in our case would never exceed this value because we are using priorities as a cost so the maximum value will always be *SIZE*. This saves us from using extra wires which would then be used in the comparison. We are using priorities instead of costs so this implementation can support 126 users at a time. If we want to make it general concerning cost or to increase users, we can change the data type of *minVal* to *int* and can use the maximum value of an integer. We use CBMC - GC to convert our C program to a Boolean circuit. Since CBMC - GC allows a C program to be converted to a Boolean circuit, so the code presented in this section is very C-like. However, the framework has its particular input and output style and some limitations that are discussed further in this section. To reduce the circuit size even further we wanted to convert the single-bit values to type *Bool* but currently *boolean* types are not supported by CBMC - GC. We now discuss the steps that are required to create a Boolean circuit by using CBMC - GC.

5.2.1 Initilization

Listing 5.1 shows the structure of the data and the globally defined variables. The structure refers to a matrix *M*. The size of the matrix is *SIZE*, we have defined it as 4 here for example but it can be changed as per need. In the struct, *C* holds the actual cost matrix. The *row_marked* and *col_marked* variables refer to the rows and columns that have been selected because of a selected zero and *marked* being the zero that has been selected. The *row_covered*, *col_covered*, *lonePrime* holds the value of ticked rows, ticked columns and the index of the newly marked value as described in step 5. The *step* variable saves the current step. The final assignment is saved to the *result*.

```
1  #define SIZE 4
2
3  typedef struct {
4  unsigned char C[SIZE][SIZE];
5  unsigned char row_marked[(SIZE)];
6  unsigned char col_marked[(SIZE)];
7  unsigned char marked[(SIZE)][(SIZE)];
8  unsigned char row_cover[(SIZE)];
9  unsigned char col_cover[(SIZE)];
10 unsigned char step;
11 unsigned char lonePrime[2];
12 unsigned char result[(SIZE)];
13 } Munkres;
14
15
16 unsigned char A[SIZE][SIZE] = {0};
```

```
17 unsigned char done = (SIZE);
18 Munkres M={0};
```

Listing 5.1: Data structures

5.2.2 Main Program

Listing 5.2 shows the main body of the program which is also the entry point of our program. CBMC - GC has a specific format of inputs and outputs. In our case, we are using variables as input so it is *INPUT_A_0* for the first input of computing party A and *INPUT_B_0* for the first input of computing party B. The variables can be numbered according to the need, we have 16 inputs from each party, so we number them from 0−15. The output can either be returned as an object or a variable. We are returning an object, so we do not need to define an output variable explicitly. The default function which gets executed is *mpc_main* however in CBMC - GC we can also define another entry point while running the framework. If the entry point of the program is another function, we need to specify it while executing the framework with the argument *–function function_name*. However, we have defined the default function in this case, in the *mpc_main* function inputs values from party A and B are being XORed before being assigned to the Matrix M. This step brings the priorities to the original state before being computed. We then move to the next step which is compute assignments by calling the function *get_munkres* which is described in Section 5.2.3.

```
1  Munkres mpc_main (unsigned char INPUT_A_0, unsigned char INPUT_A_1,
       unsigned char INPUT_A_2, unsigned char INPUT_A_3, unsigned char
       INPUT_A_4, unsigned char INPUT_A_5, unsigned char INPUT_A_6, unsigned
       char INPUT_A_7, unsigned char INPUT_A_8, unsigned char INPUT_A_9,
       unsigned char INPUT_A_10, unsigned char INPUT_A_11, unsigned char
       INPUT_A_12, unsigned char INPUT_A_13, unsigned char INPUT_A_14,
       unsigned char INPUT_A_15, unsigned char INPUT_B_0, unsigned char
       INPUT_B_1,  unsigned char INPUT_B_2, unsigned char INPUT_B_3, unsigned
        char INPUT_B_4, unsigned char INPUT_B_5, unsigned char INPUT_B_6,
       unsigned char INPUT_B_7, unsigned char INPUT_B_8, unsigned char
       INPUT_B_9, unsigned char INPUT_B_10, unsigned char INPUT_B_11,
       unsigned char INPUT_B_12, unsigned char INPUT_B_13, unsigned char
       INPUT_B_14, unsigned char INPUT_B_15) {
2
3  //XOR input values from two parties to obtain original priorities for the
        matrix and call the computing function.
4     unsigned char i;
5     unsigned char A[NP][NP] = { 0 };
6     A[0][0] = bitXor(INPUT_A_0, INPUT_B_0);
7     A[0][1] = bitXor(INPUT_A_1, INPUT_B_1);
8     A[0][2] = bitXor(INPUT_A_2, INPUT_B_2);
9     A[0][3] = bitXor(INPUT_A_3, INPUT_B_3);
10    A[1][0] = bitXor(INPUT_A_4, INPUT_B_4);
11    A[1][1] = bitXor(INPUT_A_5, INPUT_B_5);
12    A[1][2] = bitXor(INPUT_A_6, INPUT_B_6);
13    A[1][3] = bitXor(INPUT_A_7, INPUT_B_7);
```

```
14    A[2][0] = bitXor(INPUT_A_8, INPUT_B_8);
15    A[2][1] = bitXor(INPUT_A_9, INPUT_B_9);
16    A[2][2] = bitXor(INPUT_A_10, INPUT_B_10);
17    A[2][3] = bitXor(INPUT_A_11, INPUT_B_11);
18    A[3][0] = bitXor(INPUT_A_12, INPUT_B_12);
19    A[3][1] = bitXor(INPUT_A_13, INPUT_B_13);
20    A[3][2] = bitXor(INPUT_A_14, INPUT_B_14);
21    A[3][3] = bitXor(INPUT_A_15, INPUT_B_15);
22
23    Munkres M={0};
24
25    for (i = 0; i < SIZE; i++) {
26     unsigned char j;
27     for (j = 0; j < SIZE; j++) {
28      M.C[i][j] = A[i][j];
29     }
30    }
31    return get_munkres(A, M);
32  }
```

Listing 5.2: Main Program

5.2.3 Compute Assignments

Listing 5.3 shows the main function which is responsible for all the computation. In this function, we initialize the matrix and then call different steps depending on the state of the matrix as previously described in Section 4.2. The original code used switch cases. We replaced them with if-else statements as we were not able to use the switch implementation with CBMC-GC as they were not supported. After each step, the values of the matrix get updated and are passed to the next step. This is important because the same matrix is shared between all steps, this way we do not need to use extra variables to store the values, which would increase the size of the circuit and also make it difficult to manage the code. The while loop is executed $\mathcal{O}(n)$ times because it would at most run $SIZE$ times and since all the steps are being called inside the while loop with step 4 having the highest complexity $\mathcal{O}(n^3)$ as we later analyze in Section 5.2.7, so the asymptotic complexity is $\mathcal{O}(n^4)$.

```
1   Munkres get_munkres(unsigned char A[SIZE][SIZE], Munkres M) {
2    unsigned char done = SIZE;
3    unsigned char i;
4    unsigned char j;
5    unsigned char theEndIsNear = 0;
6
7    //Initialize the attributes of matrix  and run the computation until an
         optimal solution is found.
8    for (i = 0; i < SIZE; i++) {
9     M.row_cover[i] = 0;
10    M.row_marked[i] = 0;
11    for (j = 0; j < SIZE; j++) {
12     M.marked[i][j] = 0;
```

```
13      M.col_cover[j] = 0;
14      M.col_marked[j] = 0;
15      }
16    }
17
18    M.step = 1;
19    M = step1(M);
20    M = step2(M);
21    while (theEndIsNear != 1) {
22      if (M.step == 3) {
23        M = step3(done, M);
24      } if (M.step == 4) {
25        M = step4(M);
26      } if (M.step == 5) {
27        M = step5(M);
28      } if (M.step == 6) {
29        M = step6(M);
30      } if (M.step == 7) {
31        M = step7(M);
32        theEndIsNear = 1;
33      }
34    }
35  }
```

Listing 5.3: Function to compute assignments

5.2.4 Step 1: Row and Column Minimum

In Listing 5.4, we subtract the minimum value of each row from every element of that row. The same is repeated for the columns. The aforementioned gives us at least one zero in each row and column, after which we move to step 2. We changed the type of *minVal* from *double* to *unsigned char* which reduced the bits from 64 to 8. We have also changed the value of *minVal* from *INFINITY* to *SIZE+1* since we are using priorities and not costs. In this case, the largest number required for the first comparison would be 1 more than the total number of available topics. This is also a size optimization for the circuit, and the complexity of this step is $\mathcal{O}(n^2)$ because of the nested for-loops used to iterate over the matrix. The original code reduced rows only which was not always correct because there were cases when there were no zeros after row reduction, so we added column reduction in this step to ensure that we have at least one zero in each row and column. This is also an optimization because if it is not done, then we need to run an additional step (step 6) later to create zeros which increases program complexity.

```
1  Munkres step1(Munkres M) {
2    unsigned char i;
3    unsigned char j;
4    //For each row, calculate the smallest element and subtract it from
         every element in the row.
5    for (i = 0; i < (SIZE); i++) {
```

```
6       unsigned char minVal = SIZE+1;
7       for (j = 0; j < (SIZE); j++) {
8        if (M.C[i][j] < minVal) {
9         minVal = M.C[i][j];
10       }
11      }
12      for (j = 0; j < (SIZE); j++) {
13       M.C[i][j] = M.C[i][j] - minVal;
14      }
15   }
16   //For each column, calculate the smallest element and subtract it from
         every element in the column.
17   for (i = 0; i < (SIZE); i++) {
18      unsigned char minVal = SIZE+1;
19      for (j = 0; j < (SIZE); j++) {
20       if (M.C[i][j] < minVal) {
21        minVal = M.C[j][i];
22       }
23   }
24      for (j = 0; j < (SIZE); j++) {
25       M.C[j][i] = M.C[j][i] - minVal;
26      }
27   }
28   M.step = 2;
29 }
```

Listing 5.4: Row and Column Minimum

5.2.5 Step 2: Mark Rows and Columns

In Listing 5.5, we check for all the zero values in the matrix which are not yet marked, which means the possible assignments that have not been selected yet. If the row and column of a zero value are not yet marked, we set the value of marked as *true* and also mark the row and column which had the assignment, and proceed to step 3. The complexity of this step is $\mathcal{O}(n^2)$ because it uses nested loops to star zeros in the matrix and mark the corresponding row and column.

```
1  Munkres step2(Munkres M) {
2   unsigned char i;
3   unsigned char j;
4   //Find a zero in the matrix and mark it along with its row and column,
        repeat for all elements in the matrix.
5   for (i = 0; i < (SIZE); i++) {
6       for (j = 0; j < (SIZE); j++) {
7         if (M.C[i][j] == 0 && !M.marked[i][j]) {
8          if (!M.row_marked[i] && !M.col_marked[j]) {
9           M.marked[i][j] = 1;
10          M.row_marked[i] = 1;
11          M.col_marked[j] = 1;
12          break;
```

```
13          }
14         }
15       }
16  }
17  M.step = 3;
18 }
```

Listing 5.5: Mark Rows and Columns

5.2.6 Step 3: Check for Optimal Solution

In Listing 5.6, for every marked value in step 2 we mark its column as covered. Then we check if we have covered all columns with the minimum number of marked lines, i.e., we have an assignment in all columns; if yes, then we proceed to the last step, i.e., step 7 else we move to step 4 for further processing. The complexity of this step is $\mathcal{O}(n^2)$ which is inevitable because we use nested loops to cover the columns of all selected zeros and we have to iterate over all values in the matrix.

```
1  Munkres step3(Munkres M) {
2   unsigned char i;
3   unsigned char j;
4   unsigned char cont = 0;
5   //Cover each column containing a marked zero.
6   for (i = 0; i < (SIZE); i++) {
7      for (j = 0; j < (SIZE); j++) {
8       if (M.marked[i][j] == 1) {
9        M.col_cover[j] = 1;
10       cont++;
11      }
12     }
13  }
14  //If all columns are covered then it means we have an optimal solution
        and proceed to step 7, otherwise proceed to step 4.
15  if (done == cont) {
16     M.step = 7;
17     } else {
18     M.step = 4;
19  }
20 }
```

Listing 5.6: Check for Optimal Solution

5.2.7 Step 4: Mark Selected Zeros

In Listing 5.7, we check if for all the selected zeros their corresponding rows and columns have been marked. If they are marked, then we proceed to step 6 to create additional zeros. If we only have the newly created zeros from step 6, we move to step 5 for further processing.

The complexity of this step is $\mathcal{O}(n^3)$ because here we are building augmented path for the matrix by gaining a constant number of alternating elements in each run. Since this step is repeated until we have no uncovered zeros so this step adds the complexity of $\mathcal{O}(n)$ to the nested for loops having a complexity of (n^2) making it $\mathcal{O}(n^3)$.

```
Munkres step4(Munkres M) {
 unsigned char end = 0;
 int markedIndex = 0;
 //Repeat until there are no zeros left that are not covered.
 while (end == 0) {
    unsigned char i;
    unsigned char j;
    for (i = 0; i < (SIZE); i++) {
      unsigned char j;
      for (j = 0; j < (SIZE); j++) {
       markedIndex = -1;
       if (M.marked[i][j] == 1) {
        markedIndex = j;
        break;
       }
      }
     //Find a zero which is not covered and prime it.
     for (j = 0; j < (SIZE); j++) {
      end = 1;
      if (M.C[i][j] == 0 && M.row_cover[i] != 1
      && M.col_cover[j] != 1) {
       M.marked[i][j] = 2;
       //If there is no marked zero in this row go to step 5.
       if (markedIndex == -1) {
        M.lonePrime[0] = i;
        M.lonePrime[1] = j;
        M.step = 5;

        return M;

       } else {
       //Mark this row as covered and mark the column containing the
          primed zero as not covered
        end = 0;
        M.row_cover[i] = 1;
        M.col_cover[markedIndex] = 0;
       }
      }
     }
    }
 }
 //Proceed to step 6 to create additional zeros.
 M.step = 6;
 return M;
}
```

Listing 5.7: Mark selected zeros

5.2.8 Step 5: Create Augmenting Path

In Listing 5.8, we check the additional rows marked by step 4 and convert it into the acceptable form to detect the final assignment. All the newly created zeros marked by step 4 are then detected here, and all zeros in the matrix are marked with 1 along with the corresponding rows. We then go back to step 3 to check for an optimal solution. We have optimized this step by removing unnecessary loops which were not adding anything to the functionality but were just there to increase the running time, and we also removed the *memcpy* function that was used to copy matrix into memory and was not providing any benefit in the circuit level. The complexity of this step is $\mathcal{O}(n^2)$ since we are just using for loops with $\mathcal{O}(n)$ to mark the primed zeros from step 4 and a while loop with $\mathcal{O}(n)$ complexity to repeat it until all primed zeros are marked with no starred zero in their column making the total complexity $\mathcal{O}(n^2)$.

```
1  Munkres step5(Munkres M) {
2   unsigned char mat[(SIZE)][(SIZE)]={0};
3   // Copy all marked value to mat variable for further processing.
4   for (unsigned char i = 0; i < (SIZE); i++) {
5      for (unsigned char j = 0; j < (SIZE); j++) {
6       mat[i][j]=M.marked[i][j];
7      }
8   }
9   unsigned char path[(SIZE)][2]={0};
10  unsigned char count = 0;
11
12  path[count][0] = M.lonePrime[0];
13  path[count][1] = M.lonePrime[1];
14  unsigned char i;
15  unsigned char done = 0;
16
17  //Create a series of alternating primed and starred zeros. For every
        uncovered primed zero found in step 4, we might have a starred zero
        in its column. We will always have one primed zero in its row. Keep
        itterating until we reach the point where no primed zero has a
        starred zero in its column.
18  while (done == 0) {
19     unsigned char rowEnd = 0;
20     for (i = 0; i < (SIZE); i++) {
21      if (mat[i][path[count][1]] == 1) {
22       count++;
23       path[count][0] = i;
24       path[count][1] = path[count - 1][1];
25       mat[i][path[count][1]] = 0;
26       rowEnd = 1;
27       break;
28      }
29     }
30     if (rowEnd == 0) {
31     done = 1;
32     }
33     if (done != 1) {
```

```
34      for (i = 0; i < (SIZE); i++) {
35       if (mat[path[count][0]][i] == 2) {
36        count++;
37        path[count][0] = path[count - 1][0];
38        path[count][1] = i;
39        mat[i][path[count][1]] = 0;
40        break;
41       }
42      }
43     }
44  }
45  count++;
46  //Unmark all the marked zeros and mark all the primed zeros and remove
        the primes.
47  for (i = 0; i < count; i++) {
48     if (M.marked[path[i][0]][path[i][1]] == 1) {
49      M.marked[path[i][0]][path[i][1]] = 0;
50      M.row_marked[path[i][0]] = 0;
51     } else {
52      if (M.marked[path[i][0]][path[i][1]] == 2) {
53       M.marked[path[i][0]][path[i][1]] = 1;
54       M.row_marked[path[i][0]] = 1;
55      }
56     }
57  }
58  //Uncover all rows and column and move to step 3
59  for (i = 0; i < (SIZE); i++) {
60     M.row_cover[i] = 0;
61     M.col_cover[i] = 0;
62  }
63  M.step = 3;
64 }
```

Listing 5.8: Mark additional zeros

5.2.9 Step 6: Create Additional Zeros

In Listing 5.9, to create additional zeros we check for the minimum value that is uncovered and subtract it from all the uncovered elements, and add it to all the elements that are covered twice and go back to step 4. Same as step 1 we have optimized *minVal* and data types in this step. The complexity of this step is $\mathcal{O}(n^2)$ because we are using nested for loops to iterate over the matrix to create zeros, which is a three step procedure and cannot be combined in a single nested loop.

```
1  Munkres step6(Munkres M) {
2   unsigned char i;
3   unsigned char j;
4   unsigned char minVal = SIZE+1;
5   // Find the smallest element that is not covered.
6   for (i = 0; i < (SIZE); i++) {
```

```
7       for (j = 0; j < (SIZE); j++) {
8        if (M.C[i][j] < minVal && M.row_cover[i] != 1
9        && M.col_cover[j] != 1) {
10        minVal = M.C[i][j];
11       }
12      }
13   }
14   //Add the smallest elements to all the elements that are covered twice.
15   for (i = 0; i < (SIZE); i++) {
16      if (M.row_cover[i] == 1) {
17       for (j = 0; j < (SIZE); j++) {
18        M.C[i][j] = M.C[i][j] + minVal;
19       }
20      }
21   }
22   //Subtract it from all the uncovered elements.
23   for (i = 0; i < (SIZE); i++) {
24      if (M.col_cover[i] != 1) {
25       for (j = 0; j < (SIZE); j++) {
26        M.C[j][i] = M.C[j][i] - minVal;
27       }
28      }
29   }
30   // Go back to step 4.
31   M.step = 4;
32 }
```

Listing 5.9: Create additional zeros

5.2.10 Step 7: Display Optimal Solution

After having found the optimal solution in step 3, we proceed to the step shown in Listing 5.10. We now save the indices of the assignments to map them to the original input matrix. After this step, the assignments are returned as an output of the program. We have introduced an attribute *result* which was not part of the original code, we assign the final values to this array. The complexity of this step is $\mathcal{O}(n^2)$ because of the nested for loops used to iterate over the matrix and record indices of the selected priorities.

```
1  Munkres step7(Munkres M) {
2   //Save the selected values to the result variable.
3   unsigned char count=0;
4   for (unsigned char i = 0; i < SIZE; i++) {
5      for (unsigned char j = 0; j < (SIZE); j++) {
6       if (M.marked[i][j]==1) {
7        M.result[count]=A[i][j];
8        count++;
9       }
10     }
11  }
12    return M;
```

```
13 }
```

Listing 5.10: Display Optimal Solution

5.2.11 Limitations

We faced some issues in implementing our problem in CBMC-GC. The first issue was with pointers; programmers heavily rely on pointers in C Language while the current version of CBMC-GC has limited support for pointers. Also, for more extensive circuits, the compilation times were quite large because of loop unrolling and the high complexity.

We solved these problems by generating scalable circuits which is explained in Section 5.2.12. The complexity of this algorithm can be made better, i.e., $\mathcal{O}(n^3)$ by implementing step 4 to step 6 together by using another algorithm. We do not build the augmenting path in a single step, when the value is returned from step 6 we do not reset the partially completed augmented path to keep track of the marked values. For instance, this can be implemented by using maximum weight bipartite matching algorithm [Kuh10]. The matching M would take at most n phases as it is increased by 1 in each phase of the algorithm. Every step other than M would take $\mathcal{O}(n^2)$ which would make the total running time to $\mathcal{O}(n^3)$.

5.2.12 Generating Scalable Circuits

Generating huge circuits in CBMC - GC is still a problem and to overcome this we made a utility that makes our system supported for larger number of participants. We generate circuits separately for each step and convert them to Bristol format. In Bristol format, in the first line of the circuit file, the number of gates and the number of wires are defined. In the second line, there are two numbers n_1 and n_2 which define the number of wires in the inputs to the function given by the circuit, the inputs are assumed to be two always but in case if there is a single input then the size of the second input is set to zero. These numbers are followed by n_3 which define the number of wires in the output. The wires are ordered in such a way that n_1 wires correspond to the first input value, and n_2 wires correspond to the second input values while the n_3 wires correspond to the output of the circuit. The gates are then listed in the format having the number of input wires, the number of output wires, list of input wires, list of output wires and the gate operation, e.g., 2 1 154 150 155 XOR [Til]. We then convert that Bristol circuit to ABY format. The first two lines contain the inputs bits which are followed by the gate identifier input and output wires and the last line contains the output bits. In this we convert the INV gates to XORs so the final circuit file has only AND, XOR and OR gates.

We generate five circuit files because of the high complexity of step4 and also because we need to run it n times due to the while loop, the first one contains step1 and step2, the second one contains step3, the third has step4, the fourth contains step5 and step6 and the last deals with the final assignments. We modified the code parts to optimize the process, and in the

last step, we just return an array of the final assignments to reduce the number of gates being used, unlike other steps where the whole object containing both matrices and the supporting data is being returned. Once we have all files ready we move to the combining part. We created a utility that combines different circuit files to one file in ABY format. Since each step is dependent on the input from the previous step, we modified the code to initialize the structure with values from the previous step.

We created another utility to combine these files. The utility reads two circuit files at a time. The first circuit file is read and written in a new file, and its output bits are taken as the input bits for the second circuit file. The input bits of the second circuit file is mapped 1-1 with the output bits of the first file. The second file is then read, and all the gates and wires are added to the new file after being updated using the mapping that we made. If the wire value does not exist in the mapping so the max bit value from the output of the first file is added to the wire value and all values are then updated including the output bits and are then written to the new file. This process drastically reduces the time to generate the circuits. However, the sizes of these circuits would be much larger than the other optimized combined circuits because of the redundant wires and the optimizations by CBMC - GC as described in Section 2.6.4.

5.3 Secure Assignments

We use ABY described in Section 2.7 for the secure-computation part of our framework. We use an example within ABY that is an adapter that can read a function represented as a Boolean circuit. The involved parties then share the private input values for this function for the secure-computation to be executed. For the evaluation, the parties use Boolean sharing and Yao's Protocol. First, we convert the Boolean circuit generated by CBMC - GC into bristol format, which is a feature provided by CBMC - GC itself. We then use a utility to convert this circuit to ABY format. The utility reads the bristol file and generates a graph with the circuit gates and wires and then writes it to a different file in a topologically ordered manner in that format. Once we have the circuit, we then use it in ABY to perform the computation securely. For large circuits, we use the method explained in Section 5.2.12 to generate this file.

The inputs are already with the computing parties, so they are used with the generated circuits, and then the outputs are generated. We have created a utility that converts the inputs to 8 bits. The input length is calculated from the Boolean circuit file, and then input vectors are created based on that. We then read the inputs into a vector, the file of the respective computing party is picked based on the role provided. We use role 0 for party A and 1 for party B which can be called server and client in the ABY context. After reading the inputs, we convert them to bit vector where all inputs are stored bitwise. We then use `PutSIMDINGate` gates to load the inputs to the share and `PutOUTGate` is used to store the result of a share. `ExecCircuit()` method is then executed to start the circuit evaluation. After the evaluation,

results are then sent to party B (client) which are then converted from bit vectors to decimal values.

6 Evaluation

In this section, we present how the circuit scales concerning size and runtime both for evaluation and generation. The runtime and communication time taken by circuits in ABY using Yao's GC protocol and the GMW protocol. Also, the difference between two of our implementations of the circuits and how well they scale.

6.1 Benchmarking the Circuit Generation

Since XOR gates can be evaluated for free as described in Section 2.4.3.3, we focus on reducing the number of AND gates used to lower the cost. We show how AND gates grow with the change in size and also compare the more scalable circuit generation with the non-scalable version.

6.1.1 Environment

The following measurements have been made on a machine with 16GB memory and Intel® Core™ i5-4300U CPU @ 1.90GHz as the processor. We did five runs each for recording the measurements.

6.1.2 Benchmarks for Circuit Size

Table 6.1 shows the increase in the number of AND gates with the increasing size of the circuit. The graphical representation of the Table 6.1 is shown in Figure 6.1 where we can see a growth in the number of AND gates for all five circuits with the increase in size of the circuit. Circuit 3 grows more than the rest because of its high complexity $\mathcal{O}(n^3)$ than the rest of the circuits having complexity $\mathcal{O}(n^2)$, and circuit 4 has a large number of gates because we combined Step 5 and Step 6 of the algorithm in this circuit which has many operations, so more gates are used in these steps which results in more significant circuit sizes. The rest of the circuits have fewer operations and therefore require less number of gates.

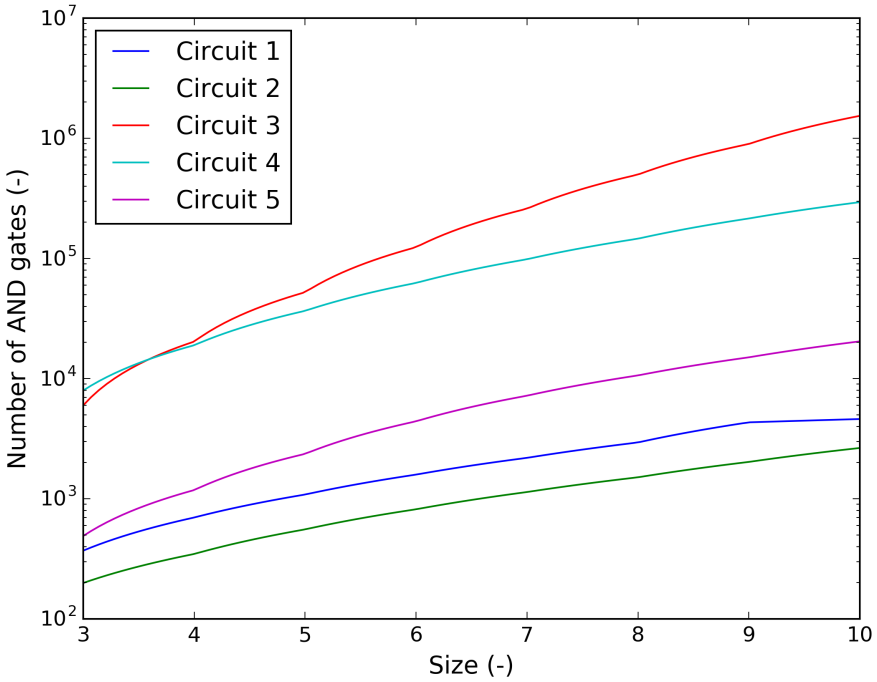

Figure 6.1: The number of AND gates in the different circuits for varying number of participants and topics in the assignment problem

Size	Circuit 1	Circuit 2	Circuit 3	Circuit 4	Circuit 5
3	370	597	17 967	23 892	490
4	698	1 388	81 280	75 788	1 181
5	1 084	2 780	261 625	183 170	2 363
6	1 588	4 902	743 568	375 024	4 410
7	2 189	7 959	1 814 414	688 807	7 202
8	2 940	12 064	3 985 120	1 167 368	10 623
9	4 324	18 225	8 068 365	1 934 586	15 026
10	4 606	26 450	15 366 070	2 938 300	20 401

Table 6.1: The number of AND gates in the circuits

Circuit 1 includes step 1 and 2 from the algorithm (Section 4.2) as described in Section 5.2.4 and Section 5.2.5, Circuit 2 contains step 3 which is described in Section 5.2.6, Circuit 3 contains step 4 which is described in Section 5.2.7, Circuit 4 contains step 5 and 6 as which are described in Section 5.2.8 and Section 5.2.9 and Circuit 5 contains the step for final assignment as described in Section 5.2.10.

6.1.3 Benchmarks for Runtime

We first generated circuits directly, and as it was taking large amount of time, i.e., for $n = 6$ more than 14 hours to be generated, so we shifted to a scalable approach by generating parts of circuits which speeded up the process. This can be seen in Figure 6.2 which shows a comparison between circuit sizes and the time required to generate them. The total time required to generate all circuits for size 10 is around 67 minutes which makes the scalable version more than 12x faster. Since circuit 3 is the one with highest running time, we can estimate the running time for bigger circuit size by looking at the current data. Generating circuit 3 for size 11 and 12, takes 90 and 120 minutes respectively. Another thing to be noticed here is the decrease in the runtime of circuit 1 for even sizes $(6, 8, 10)$. Circuit 3 grows the most with respect to runtime because of the large number of gates involved and the complexity of $\mathcal{O}(n^3)$.The other circuits show similar characteristics as they all have same asymptotic complexity $\mathcal{O}(n^2)$.

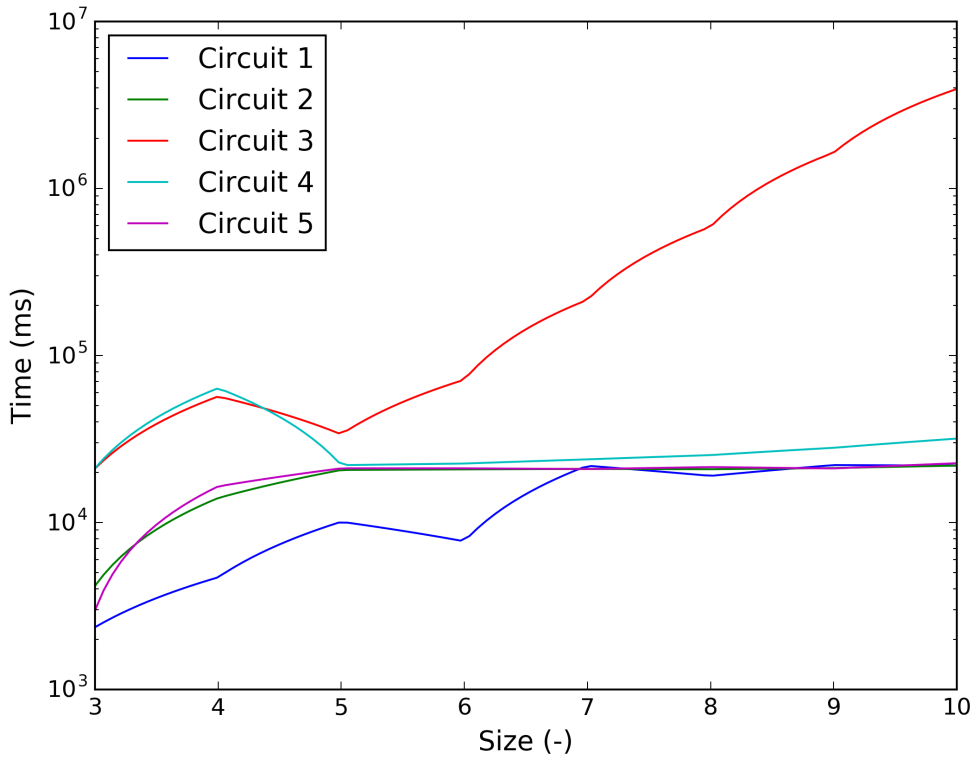

Figure 6.2: Time taken to generate circuits

6.1.4 Size Comparison

Figure 6.3 shows the comparison between our two implementations of the Hungarian algorithm concerning the number of gates as descried in Section 5.2.12. We combine these circuits in a way that the first circuit is combined with second circuit and then the resulting circuit is combined with the next circuit. Since the steps $3-6$ (Section 4.2) run in a while loop in the algorithm which runs up to n times making the total complexity of the algorithm to $\mathcal{O}(n^4)$, we need to combine circuit 2, 3 and 4 *size* times before being finally combined with the last step circuit 5.

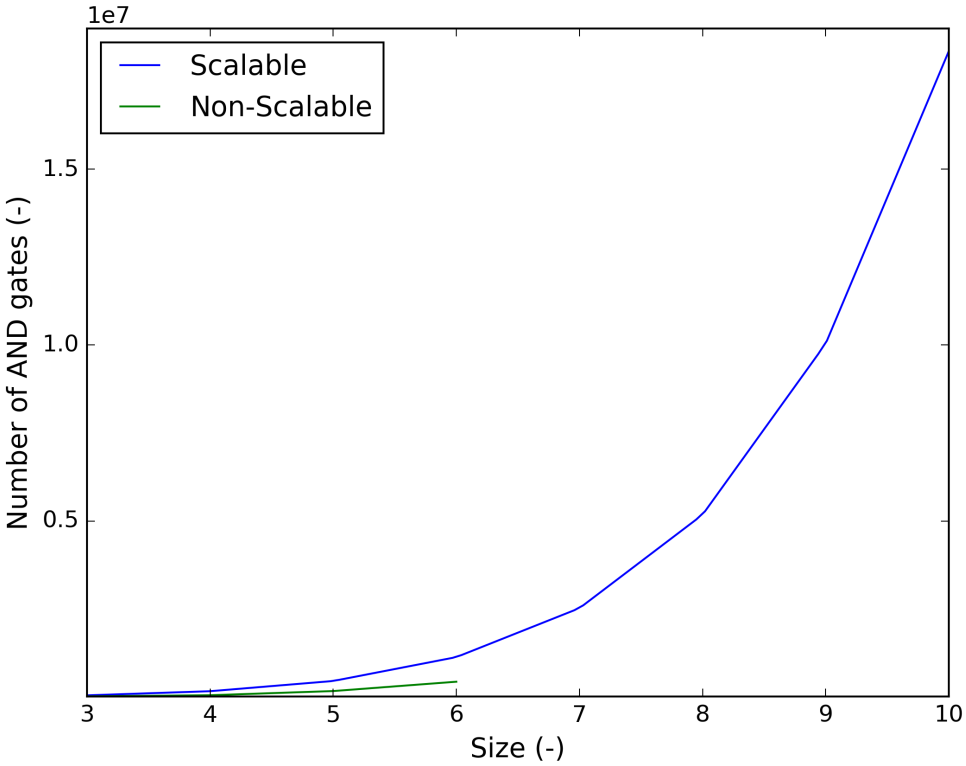

Figure 6.3: Comparison between Scalable and Non-scalable implementation

In Figure 6.3, the x-axis is the size of the circuit and on the y-axis are the number of AND gates used by the circuit. As depicted in the graph in Figure 6.3, the non-scalable version created smaller circuits because of the additional optimizations (Section 2.6.4) that run on the whole circuit in CBMC -GC. The number of AND gates can be seen in Table 6.2 which shows that the non-scalable circuits are up to 3 times smaller. However, generating these circuits took longer time and we were not able to generate more because of the $\mathcal{O}(n^4)$ complexity of the algorithm. For the scalable version, the curve that we see shows $y = Ax^4$ where A is a

constant value by which the gates are increasing. The estimated value for A is 1420 with a deviation of 25% which we have calculated from the data in Table 6.2.

Size	3	4	5	6	7	8	9	10
Scalable	43 316	160 335	451 022	1 129 492	2 520 571	5 178 115	10 040 526	18 355 827
Non-Scalable	14 275	46 422	164 594	430 568				

Table 6.2: Number of AND gates in scalable and non-scalable version

6.2 Benchmarking the Secure Computation

We evaluate our circuits in ABY by using two different sharing mechanisms Boolean sharing using GMW as described in Section 2.5 and Yao's sharing using Yao's garbled circuits as described in Section 2.4. We take into consideration the two phases, the setup phase and the online phase along with the total runtime information. In Yao's GC, in the setup phase, the server generates a garbled circuit and sends it to the client. In the online phase, the inputs of the two parties are converted into corresponding garbled inputs and then sent from the server to the client using OTs; the total runtime information contains all the phases including function evaluation and decrypting the outputs. In GMW, the setup phase deals with the generation of MTs via OTs, the online phase deals with the sharing of inputs, circuit evaluation and combining the output shares, and the total runtime information contains the setup phase, the online phase and the circuit construction (Section 2.5.2). The platform is the same as described in Section 6.1.1 and we report the average time from 5 runs. In Figure 6.4, we can see the communication numbers for sent and received data where we can see a constant growth in the communication numbers which is because of the increasing input and circuit size. Figure 6.5 shows the runtime in milliseconds which can be seen growing in all phases for all circuit sizes. In Yao's GC protocol, the communication in the setup phase includes the garbled circuit, that is the reason for it being so high, and runtime also depends on the circuit which is being encrypted. Also in the online phase, the runtime depends on the circuit that is being evaluated.

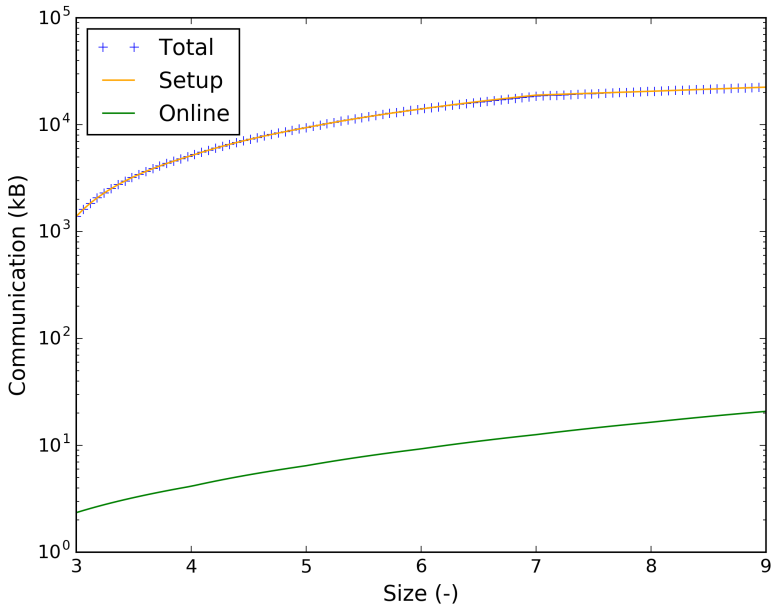

Figure 6.4: Communication using Yao's GC

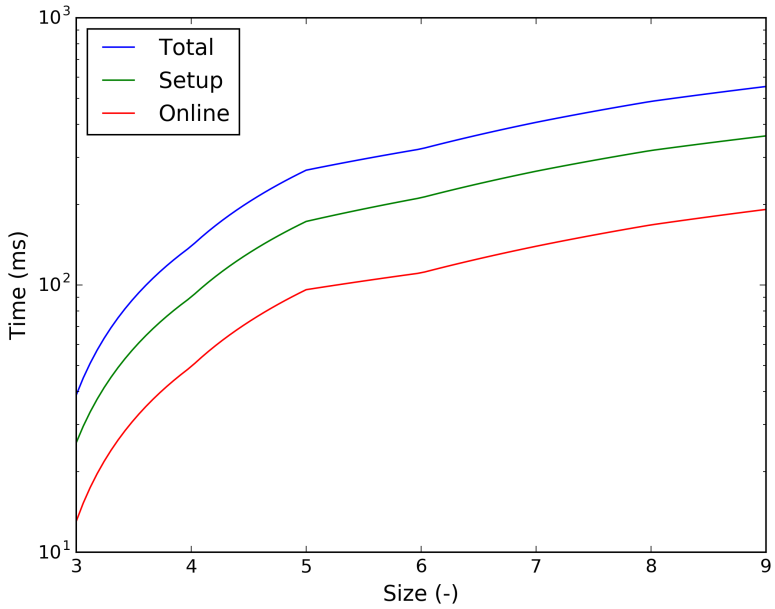

Figure 6.5: Time taken for secure computation using Yao's GC

Figure 6.6 shows the increase in communication numbers for the sizes. Figure 6.7 shows the runtime of the circuits using Boolean sharing. We can see an increase in runtimes for all the phases.

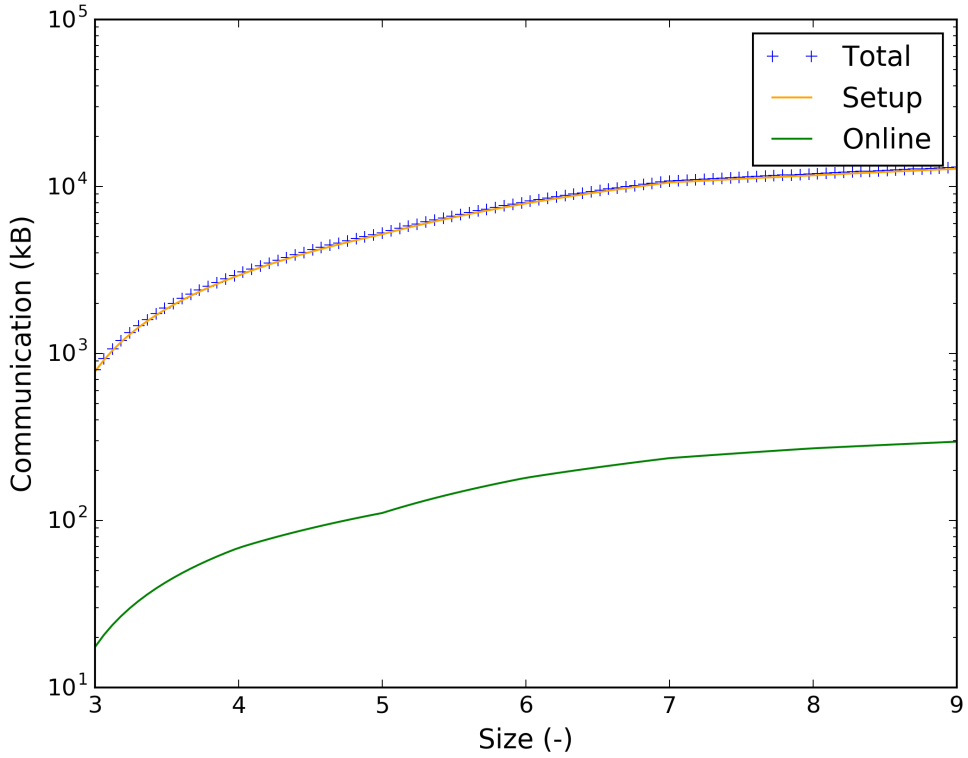

Figure 6.6: Communication using GMW

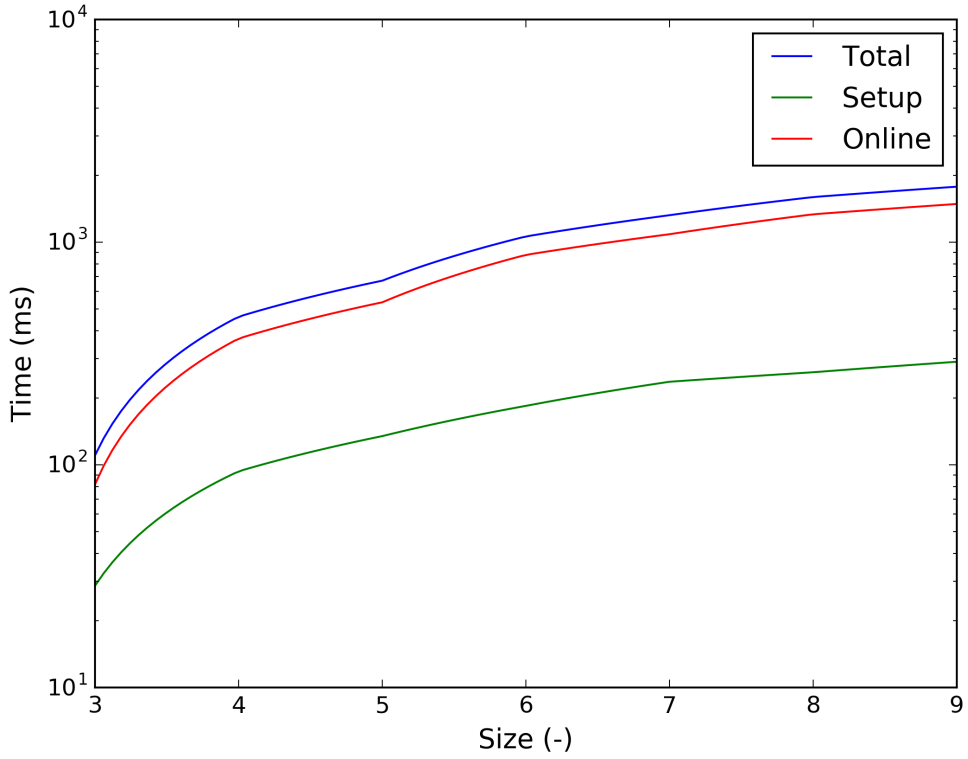

Figure 6.7: Time taken for secure computation using GMW

The graphs depicted in Figure 6.5, Figure 6.7, Figure 6.4, and Figure 6.6 show that Yao's sharing had lesser runtime while Boolean sharing had lesser communication number. The reason for this is the nature of Yao's GC protocol because in Yao's GC one party does most of the work while in GMW the work is divided between both parties evenly. Also, in GMW each party can compute their part of the result locally without communication effort.

7 Conclusion

In this chapter we summarize our work and give an overview of our results and the possible future work.

7.1 Summary

Privacy is a very popular topic these days, and we realized that there exists no system that solves the assignment problem while preserving the privacy of the participants. To not rely on any third party for such computations we designed a framework which can be used for privacy-preserving assignments in any environment.

We created a frontend that was used to collect inputs from users in an encrypted manner so that the inputs are never available in plain and can never be guessed by anyone. We used secret sharing along with encryption to store the input and share parts of it with the computing parties. We then use CBMC -GC to convert our implementation of the Hungarian algorithm to Boolean circuits which can then be used for secure computation in ABY. We used ABY to compute the output securely.

Our implementation shows expected results which means that almost everything is increasing concerning the size of the circuit. However, our implementation of the algorithm has a complexity of $\mathcal{O}(n^4)$ which can be further reduced to $\mathcal{O}(n^3)$ which would speed up the process of scalable circuit generation and would also reduce the overall circuit size. This would also affect the runtime and communication numbers in ABY.

7.2 Future Work

There can be several extensions to be done in this thesis. One of which is implementing the Hungarian Algorithm [Kuh10] in $\mathcal{O}(n^3)$ time which will further reduce the time and circuit size which will be a huge optimization and would be directly compatible with our framework as explained in Section 5.2.11. This framework can also be made compatible with other assignment problems by following the same or different techniques. Also, similar problems can be identified, and the same approach can be used to construct another privacy-preserving framework. One interesting extension would be to have an option to select the algorithm on runtime depending on the problem and size.

List of Figures

List of Tables

Bibliography

[AB09] S. ARORA, B. BARAK. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009 (cit. on p. 5).

[ALSZ15] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries"**. In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 673–701 (cit. on p. 6).

[Bea91] D. BEAVER. **"Efficient Multiparty Protocols Using Circuit Randomization"**. In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference*. Vol. 576. Lecture Notes in Computer Science. Springer, 1991, pp. 420–432 (cit. on p. 10).

[BFH+17] N. BÜSCHER, M. FRANZ, A. HOLZER, H. VEITH, S. KATZENBEISSER. **"On compiling Boolean circuits optimized for secure multi-party computation"**. In: *Formal Methods in System Design* 51.2 (2017), pp. 308–331 (cit. on pp. 11 sqq., 17 sq.).

[BHR12] M. BELLARE, V. T. HOANG, P. ROGAWAY. **"Foundations of garbled circuits"**. In: *the ACM Conference on Computer and Communications Security, CCS'12*. ACM, 2012, pp. 784–796. URL: http://doi.acm.org/10.1145/2382196.2382279 (cit. on p. 7).

[BHWK16] N. BÜSCHER, A. HOLZER, A. WEBER, S. KATZENBEISSER. **"Compiling Low Depth Circuits for Practical Secure Computation"**. In: *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security*. Vol. 9879. Lecture Notes in Computer Science. Springer, 2016, pp. 80–98 (cit. on pp. 16 sq.).

[BMR90] D. BEAVER, S. MICALI, P. ROGAWAY. **"The Round Complexity of Secure Protocols (Extended Abstract)"**. In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*. ACM, 1990, pp. 503–513 (cit. on p. 8).

[BSA13] M. BLANTON, A. STEELE, M. ALIASGARI. **"Data-oblivious graph algorithms for secure computation and outsourcing"**. In: *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13*. ACM, 2013, pp. 207–218 (cit. on p. 23).

[Cde14] CDELPRADO. *Munkres*. https://github.com/cdelprado/munkres. 2014 (cit. on pp. 29, 39).

[CG18] C. CADWALLADR, E. GRAHAM-HARRISON. **"Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach"**. In: *The Guardian* (03/18/2018). URL: `https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election` (visited on 03/26/2018) (cit. on p. 1).

[CHK+12] S. G. CHOI, K. HWANG, J. KATZ, T. MALKIN, D. RUBENSTEIN. **"Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces"**. In: *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012*. Vol. 7178. Lecture Notes in Computer Science. Springer, 2012, pp. 416–432 (cit. on p. 10).

[CKL04] E. M. CLARKE, D. KROENING, F. LERDA. **"A Tool for Checking ANSI-C Programs"**. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176 (cit. on p. 12).

[DES16] J. DOERNER, D. EVANS, A. SHELAT. **"Secure Stable Matching at Scale"**. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1602–1613 (cit. on pp. 22, 24 sq.).

[DSZ15] D. DEMMLER, T. SCHNEIDER, M. ZOHNER. **"ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation"**. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*. The Internet Society, 2015 (cit. on pp. c, 1 sq., 18 sq., 34).

[EGL85] S. EVEN, O. GOLDREICH, A. LEMPEL. **"A Randomized Protocol for Signing Contracts"**. In: *Commun. ACM* 28.6 (1985), pp. 637–647 (cit. on p. 6).

[FGM07] M. K. FRANKLIN, M. A. GONDREE, P. MOHASSEL. **"Improved Efficiency for Private Stable Matching"**. In: *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*. Vol. 4377. Lecture Notes in Computer Science. Springer, 2007, pp. 163–177 (cit. on p. 23).

[FH95] C. W. FRASER, D. R. HANSON. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley, 1995 (cit. on p. 12).

[FHK+14] M. FRANZ, A. HOLZER, S. KATZENBEISSER, C. SCHALLHART, H. VEITH. **"CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations"**. In: *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*. Vol. 8409. Lecture Notes in Computer Science. Springer, 2014, pp. 244–249 (cit. on p. 13).

[GMW87] O. GOLDREICH, S. MICALI, A. WIGDERSON. **"How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority"**. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. ACM, 1987, pp. 218–229 (cit. on pp. c, 9).

[Gol06] P. GOLLE. **"A Private Stable Matching Algorithm"**. In: *Financial Cryptography and Data Security, 10th International Conference, FC 2006*. Vol. 4107. Lecture Notes in Computer Science. Springer, 2006, pp. 65–80 (cit. on p. 23).

[GS13] D. GALE, L. S. SHAPLEY. **"College Admissions and the Stability of Marriage"**. In: *The American Mathematical Monthly* 120.5 (2013), pp. 386–391 (cit. on pp. 22 sqq.).

[GS91] J. v. z. GATHEN, G. SEROUSSI. **"Boolean Circuits Versus Arithmetic Circuits"**. In: *Inf. Comput.* 91.1 (1991), pp. 142–154 (cit. on p. 27).

[Har03] D. HARRIS. **"A taxonomy of parallel prefix networks"**. In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*. Vol. 2. 2003, 2213–2217 Vol.2 (cit. on p. 17).

[HFKV12] A. HOLZER, M. FRANZ, S. KATZENBEISSER, H. VEITH. **"Secure two-party computations in ANSI C"**. In: *the ACM Conference on Computer and Communications Security, CCS'12*. ACM, 2012, pp. 772–783 (cit. on pp. c, 1 sq., 4, 12, 18).

[IKNP03] Y. ISHAI, J. KILIAN, K. NISSIM, E. PETRANK. **"Extending Oblivious Transfers Efficiently"**. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 145–161 (cit. on p. 6).

[KMV91] S. KHULLER, S. G. MITCHELL, V. V. VAZIRANI. **"On-Line Algorithms for Weighted Bipartite Matching and Stable Marriages"**. In: *Automata, Languages and Programming, 18th International Colloquium, ICALP91*. Vol. 510. Lecture Notes in Computer Science. Springer, 1991, pp. 728–738 (cit. on p. 22).

[KS08] V. KOLESNIKOV, T. SCHNEIDER. **"Improved Garbled Circuit: Free XOR Gates and Applications"**. In: *Automata, Languages and Programming, 35th International Colloquium, Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*. Vol. 5126. Lecture Notes in Computer Science. Springer, 2008, pp. 486–498 (cit. on pp. 8, 17).

[KS14] M. KELLER, P. SCHOLL. **"Efficient, Oblivious Data Structures for MPC"**. In: *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security*. Vol. 8874. Lecture Notes in Computer Science. Springer, 2014, pp. 506–525 (cit. on p. 23).

[KSMB13] B. KREUTER, A. SHELAT, B. MOOD, K. R. B. BUTLER. **"PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation"**. In: *Proceedings of the 22th USENIX Security Symposium*. USENIX Association, 2013, pp. 321–336 (cit. on pp. 12, 18).

[KSS12] B. KREUTER, A. SHELAT, C. SHEN. **"Billion-Gate Secure Computation with Malicious Adversaries"**. In: *Proceedings of the 21th USENIX Security Symposium*. USENIX Association, 2012, pp. 285–300 (cit. on pp. 12, 18).

[KT14] D. KROENING, M. TAUTSCHNIG. **"CBMC - C Bounded Model Checker - (Competition Contribution)"**. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 389–391 (cit. on p. 13).

[Kue04] A. KUEHLMANN. **"Dynamic transition relation simplification for bounded property checking"**. In: *2004 International Conference on Computer-Aided Design, ICCAD 2004*. IEEE, 2004, pp. 50–57 (cit. on p. 15).

[Kuh10] H. W. KUHN. "The Hungarian Method for the Assignment Problem". In: *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 2010, pp. 29–47 (cit. on pp. 26, 29, 49, 60).

[LWN+15] C. LIU, X. S. WANG, K. NAYAK, Y. HUANG, E. SHI. **"ObliVM: A Programming Framework for Secure Computation"**. In: *2015 IEEE Symposium on Security and Privacy, SP 2015*. IEEE, 2015, pp. 359–376 (cit. on pp. 12, 18).

[MCB06] A. MISHCHENKO, S. CHATTERJEE, R. K. BRAYTON. **"DAG-aware AIG rewriting a fresh look at combinational logic synthesis"**. In: *Proceedings of the 43rd Design Automation Conference, DAC 2006*. ACM, 2006, pp. 532–535 (cit. on p. 15).

[MGC+16] B. MOOD, D. GUPTA, H. CARTER, K. R. B. BUTLER, P. TRAYNOR. **"Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation"**. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016*. IEEE, 2016, pp. 112–127 (cit. on pp. 12, 18).

[NN01] M. NAOR, K. NISSIM. **"Communication preserving protocols for secure function evaluation"**. In: *Proceedings on 33rd Annual ACM Symposium on Theory of Computing*. ACM, 2001, pp. 590–599 (cit. on p. 23).

[NPS99] M. NAOR, B. PINKAS, R. SUMNER. **"Privacy preserving auctions and mechanism design"**. In: *EC*. 1999, pp. 129–139 (cit. on p. 8).

[Rab05] M. O. RABIN. **"How To Exchange Secrets with Oblivious Transfer"**. In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 187 (cit. on p. 6).

[RSS+17] M. S. RIAZI, E. M. SONGHORI, A. SADEGHI, T. SCHNEIDER, F. KOUSHANFAR. **"Toward Practical Secure Stable Matching"**. In: *PoPETs* 2017.1 (2017), pp. 62–78 (cit. on pp. 22, 24).

[Sch12] T. SCHNEIDER. *Engineering Secure Two-Party Computation Protocols - Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer, 2012 (cit. on p. 5).

[SG] B. L. SYNTHESIS, V. GROUP. *ABC: A System for Sequential Synthesis and Verification, Release 30916,* URL: `http://people.eecs.berkeley.edu/~alanmi/abc/` (cit. on p. 15).

[SHS+15] E. M. SONGHORI, S. U. HUSSAIN, A. SADEGHI, T. SCHNEIDER, F. KOUSHANFAR. **"TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits"**. In: *2015 IEEE Symposium on Security and Privacy, SP 2015*. IEEE, 2015, pp. 411–428 (cit. on pp. 12, 18).

[SZ13] T. SCHNEIDER, M. ZOHNER. **"GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits"**. In: *Financial Cryptography and Data Security - 17th International Conference, FC 2013*. Vol. 7859. Lecture Notes in Computer Science. Springer, 2013, pp. 275–292 (cit. on p. 10).

[Til] S. TILLICH. *Circuits of Basic Functions Suitable For MPC and FHE*. `https://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/` (cit. on p. 49).

[Tra13] TRAVIST. *jsencrypt*. `https://github.com/travist/jsencrypt`. 2013 (cit. on p. 34).

[Wu09] T. WU. *RSA and ECC in JavaScript*. `http://www-cs-students.stanford.edu/~tjw/jsbn`. 2009 (cit. on p. 34).

[Yak17] S. YAKOUBOV. **"A Gentle Introduction to Yao's Garbled Circuits"**. In: 2017 (cit. on p. 8).

[Yao86] A. C. YAO. **"How to Generate and Exchange Secrets (Extended Abstract)"**. In: *27th Annual Symposium on Foundations of Computer Science*. IEEE, 1986, pp. 162–167 (cit. on pp. c, 1, 6, 8).

[ZE15] S. ZAHUR, D. EVANS. **"Obliv-C: A Language for Extensible Data-Oblivious Computation"**. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1153 (cit. on pp. 12, 18).

[ZRE15] S. ZAHUR, M. ROSULEK, D. EVANS. **"Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates"**. In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Vol. 9057. Lecture Notes in Computer Science. Springer, 2015, pp. 220–250 (cit. on pp. 8 sq.).

[ZWR+16] S. ZAHUR, X. S. WANG, M. RAYKOVA, A. GASCÓN, J. DOERNER, D. EVANS, J. KATZ. **"Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation"**. In: *IEEE Symposium on Security and Privacy, SP 2016*. IEEE, 2016, pp. 218–234 (cit. on pp. 23 sq.).