A Framework for Efficient Secure Three-Party Computation

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

Faculty of Engineering

BY

Arun Joseph

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2018

Declaration of Originality

I, **Arun Joseph**, with SR No. **04-04-00-10-42-16-1-13762** hereby declare that the material presented in the thesis titled

A Framework for Efficient Secure Three-Party Computation

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2017-2018**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discusions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

© Arun Joseph
June, 2018
All rights reserved

DEDICATED TO

My family and friends

Acknowledgements

In the name of ‘God’, the most beneficent and merciful who gave us strength and knowledge to complete this thesis. This thesis is a part of my ‘M tech in Computer Science’. This has proved to be a great experience. I am greatly indebted to my adviser Dr. Arpita Patra for her sincere guidance and motivation I received to uphold my skills in coding . Her constant contribution in stimulating suggestions and encouragements helped me to coordinate the project successfully. The experience under her supervision has given me an extra confidence to take up other similar works.

I would like to thank Dr. Ashish Choudhary ,professor at IIIT-Bangalore for his collaboration in the project work and also my fellow lab-mates, Ajith and Megha for the support and help in my thesis work. I extend my gratitude to co-authors,labmates and collegues:Divya, Megha, Ajith, Swati, Harsh, Nishat and Pratik for the useful discussions and for making the working environment lively. I acknowledge all IISc faculties that I have come across in two years of my IISc life.

I express my sincere appreciation to my parents ,all my teachers, especially Mrs.Ambika,Mrs. Bimdhu,Sister Roslin,Dr. Ajeesh,Mr. Arun M , for moulding my skills,intelligence , personality , thus shaping to a good and capable human .

To my close friends, Akshay, Minna, Philips, Sreeparvathy, Imthias and Georgin, no words will value your support and love all through my difficult and hectic time in IISc life ; heartfelt thanks to them for making IISc memorable and beautiful part of my life.

Abstract

Secure Multi-Party Computation (MPC) protocols allow a set of mutually-distrusting parties to jointly evaluate a common known function over their inputs, while maintaining correctness of the outputs and the security of their inputs. In the past decade, a plethora of MPC protocols have evolved, with improvements in efficiency and security. Specifically, the introduction of *mixed protocols* opened up the possibility of more efficient practical realizations that use a mixture of arithmetic and boolean circuits to represent common operations in an optimal fashion.

In this work, we extend the well known two-party mixed-protocol Arithmetic Boolean Yao [9] (ABY) framework to the three party setting, obtaining a new protocol which we call BAG (Boolean Arithmetic Garble). The BAG protocol is constructed by the combining efficient schemes based on Boolean sharing, Arithmetic sharing, and Garbled circuit based sharing; and using state-of-the-art conversion techniques.

Similar to the ABY-framework, the BAG-framework also considers semi-honest centralized static adversaries, and tolerates at most two corruptions out of three (dishonest majority). Again following ABY, the BAG-framework has two phases (offline, online) and allows us to pre-compute many of the expensive cryptographic operations, thus reducing the online phase execution time.

Implenting our protocol, we have benchmarked ABY (publicly available [8]) and BAG w.r.t. the atomic operations which are known to give insights into the design of mixed protocols. In comparison to ABY, our framework has an average of 55% overhead for elementary operations and 20% overhead for conversions, which is reasonable considering that it is in the three-party setting.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

1.1 Secure Multiparty Computation 2

1.1.1 Boolean and Arithmetic Circuits 3

1.2 Related Work 3

1.3 Our Contribution 4

1.4 Organization 4

1.5 Preliminaries 4

1.5.1 The Three-Party setting 5

1.5.2 Security against semi-honest adversaries 5

1.5.3 Notation 5

1.5.4 Oblivious Transfer 6

1.5.5 Garbling Scheme 6

2 BAG protocol **7**

2.1 Sharing Types 7

2.1.1 Arithmetic Sharing 7

2.1.2 Boolean Sharing 9

2.1.3 Garbled Sharing 11

2.2 Sharing Conversions . . . 14
2.2.1 Sharing Conversions when $\mathbb{F} = \mathrm{GF}(2^\ell)$ in the Arithmetic World . . . 15
2.2.2 Sharing Conversions when $\mathbb{F} = \mathbb{Z}_{2^\ell}$ in the Arithmetic World . . . 16

3 Implementation Results **19**
3.1 Experiment setup . . . 19
3.1.1 Hardware Details . . . 19
3.1.2 Design and Implementation . . . 20
3.2 Benchmarking . . . 21

4 Conclusion **23**

Bibliography **24**

List of Figures

2.1 OT_1 between P_3 and P_1 17
2.2 OT_2 between P_3 and P_2 17
2.3 OT_3 between P_1 and P_2 18

3.1 Setup time and communication (in Bytes) for a single atomic operation on 64-bit values in a local and cloud scenario, averaged over 1000 operations 20
3.2 Online time and communication (in bytes) for a single atomic operation on 64-bit values in the local and cloud scenarios, averaged over 1000 operations 21

List of Tables

3.1 Comparing online cloud performance of important elementary operations with ABY[9] 22

Chapter 1

Introduction

With the internet as the backbone, and especially after the significant leaps in wireless communication technology and the near-ubiquitous proliferation of the *Internet of Things* idea in the past decade, we now have millions of computational devices, with a great diversity of capabilities and uses, connected to each other across the globe. This new ecosystem has brought with it a plethora of security and privacy challenges hitherto unexplored.

In its beginnings, modern (theoretical) cryptography focused more on the formal study of problems related to the protection of data (encryption), authentication of messages and guarantee of non-repudiation (digital signatures) etc. But the field has since grown to encompass many other problems and techniques of both theoretical interest as well as practical applications. The framework of Secure Multiparty Computation (MPC), introduced by Andrew Chi-Chih Yao [21], is a sub-field of modern cryptography that studies problems related to performing computations on data while preserving privacy.

Although many of the earlier MPC constructions were studied for the sake of theoretical interest without focusing on implementation constraints, the information revolution in the twenty-first century has also shifted the focus towards practical MPC. Some of the notable applications of efficient MPC design techniques include secure auctions, secure elections, privacy-preserving machine learning [17, 18], satellite collision [14] etc.

Over the past decade, research in multi-party computation has drifted towards a balance between theory and practice. As a result, several protocols have been proposed focus on theoretical guarantees and efficient implementation. In this work, we extend the two-party Arithmetic Boolean Yao (ABY) framework [9] to three parties. Our protocol considers a scenario where three parties (P_1, P_2, P_3), who do not trust each other, need to evaluate a known function f on a combination of their private inputs (x_1, x_2, x_3), and obtain their corresponding outputs (y_1, y_2, y_3) without the inputs being shared with each other. Some potential applications of such a protocol include private set intersection and

building classifiers without leaking training data.

1.1 Secure Multiparty Computation

Secure Multiparty Computation considers a set of n mutually-distrustful parties $P_1, \ldots, P_n$, with private inputs $x_1, \ldots, x_n$, who wish to jointly evaluate a known vector-valued function f (with n outputs $y_1, \ldots, y_n$) on $\varphi(x_1, \ldots, x_n)$ (which is a combination of their private inputs). But they also wish to ensure that a bounded adversary who controls an arbitrary collated set of up to t (of n) parties cannot learn any extra information other than the outputs of the corrupted parties (*privacy*) and cannot affect the outputs of the honest parties successfully (*correctness*). We consider a *monolithic* or *centralized* adversary, where the set of t corrupted parties can *collude* with each other. Here, the *adversary* models the distrust among parties, and the possible collusion of a small set of parties with the aim of disrupting the computation. These t parties are said to be 'corrupt' and the remaining $(n - t)$ parties are said to be 'honest'. An MPC protocol should ensure that the following conditions hold:

- **Correctness:** Each honest party P_i outputs the correct function value $y_i = f(\varphi(x_1, \ldots, x_n))$.
- **Privacy:** The adversary cannot learn any information other than what can be efficiently derived from the inputs and outputs of the corrupted parties.

There are also other properties that may be fulfilled by MPC protocols. Some of the notable ones are the following:

- **Independence of Inputs:** The corrupted parties must choose their inputs independently of the honest parties' inputs.
- **Fairness:** The corrupted parties should receive their outputs only if the honest parties also receive their outputs.
- **Guaranteed Output Delivery:** The corrupted parties should not be able to prevent the honest parties from receiving their output.

Multi-Party Computation (MPC) solutions can be categorized based on the adversary type and behavior, the nature of the function to be computed, and the network (synchronous or asynchronous). An adversary can either be centralized or decentralized, and either static or dynamic. In our work, we consider a *centralized* adversary who can *statically* corrupt at most t parties, where t is called the 'threshold'. We consider a *synchronous* network in which the 'rounds' of communication are well-defined due to the presence of a global clock. An adversary can either be semi-honest (follows

the protocol faithfully, but tries to glean extra information) or malicious (can arbitrarily deviate from the protocol). We consider only *semi-honest* adversaries for this work. We also consider the adversary to be computationally bounded (allowed probabilistic polynomial-time computation) for all our protocols.

1.1.1 Boolean and Arithmetic Circuits

In MPC protocols, the function to be computed is represented as a *circuit*, which may either be an *arithmetic* or a *boolean* circuit. When modeling a higher-level computation as a mathematical function, we can choose to represent it either as a *boolean circuit* (with $\{0, 1\}$ inputs and AND, OR, XOR gates) or as an *arithmetic circuit* (with inputs from a field and $+$, $\times$ gates). For particular high-level operations, it may be that one of these representations (arithmetic *vs.* boolean) can be realized much more efficiently than the other. For example, the comparison operation may be realized more efficiently as a boolean circuit, whereas addition may be realized more efficiently as an arithmetic circuit. But it may be difficult for a non-expert (such as a developer) to choose between these representations so as to yield the best performance in practice.

Most of the earlier protocols in MPC supported a particular representation of circuits. The generic solutions for n parties were first proposed in [12] for arithmetic circuits and [6] for boolean circuits.

Later, some protocols attempted to support the combination of these representations in the circuit for the function (to be computed). That is, a part of the circuit may be arithmetic, while another part may be boolean. Such protocols are called *mixed protocols*. An expert could choose to realize a part of her function as an arithmetic circuit and the other part as a boolean circuit, with the mixed protocol using the appropriate conversions for each part. The ABY protocol [9] is a mixed protocol, and our work also follows this regime.

However, it should be noted that a non-expert may find it difficult to choose the appropriate representations for each part of the circuit so as to maximize performance. But mixed protocols potentially allow approaches that automatically generate a "good" circuit given a function.

1.2 Related Work

We outline some of the relevant earlier work for mixed MPC protocols in this section. Even though the MPC protocol was introduced in 1982 by Yao [21] (for boolean circuits), the first mixed protocol that combined Yao's garbled circuits and homomorphic encryption [4] was published in 2007 (to the best of our knowledge). This paper used the mixed protocol technique to evaluate branching programs with applications in remote diagnostics.

The Arithmetic Boolean Yao (ABY) protocol [9] provides an improved framework that allows mixing of multiple protocols, shifts expensive parts of the protocol into a setup phase, and eliminates

the need to use expensive homomorphic encryption operations. Compared to the previous mixed protocol frameworks, ABY provides more flexibility in the design of protocols, more efficient multiplication, and more efficient conversions between the different constituent protocols. The ABY protocol was a pioneering breakthrough, after which many other researchers come up with more efficient implementations of two-party mixed protocols. For example, [19] exploits the secure hardware (SGX) to improve the performance significantly. [5] is a framework built on top of ABY, which automatically generates efficient ABY code (mixing the representations) when given a function - this allows a non-expert developer to easily write secure and peformant MPC programs for two parties.

1.3 Our Contribution

We extend the two-party Arithmetic Boolean Yao (ABY) [9] framework to the three party setting. We use existing state-of-the-art cryptographic techniques to make our protocol efficient. For the arithmetic world and the boolean world, we extend ABY sharing to the three party case. Instead of the Yao world for two parties (as in ABY), we introduce the Garbled world which is based on [3] (and originally from [1]). We call our new three-party protocol the Boolean Arithmetic Garbled (BAG) protocol. We formulate simple and efficient conversions between these newly defined three-party worlds. We implement the basic operations and compare its performance with the ABY protocol. We find that the overhead is reasonable, considering that we move to the three-party setting.

1.4 Organization

In the next section, we define the preliminaries needed for our protocol construction. The following Chapter 2 describes our protocol in detail. In this chapter, we also define the three worlds: arithmetic, boolean, and garbled, and the conversions between them.

Chapter 3 contains the experimental details and the implementation results (including the comparison to ABY).

Finally, chapter 4 summarizes our contributions and states our conclusions as well as possible avenues for future work.

1.5 Preliminaries

In this section, we formally define our protocol setting and state the basic security definitions required for understanding our protocols. We also introduce the notations used in our work and briefly describe the well-known notions of oblivious transfer and garbling schemes, which are the main building blocks in this work.

1.5.1 The Three-Party setting

We make two assumptions about the parties, of which there are three: the first is that the parties are connected pair-wise by private, secure and authenticated channels; the second assumption is that each party is modeled as a Probabilistic Polynomial Time (PPT) Turing machine. We require these assumptions for ensuring the security of our constructions.

1.5.2 Security against semi-honest adversaries

We use the *semi-honest* (passive) adversary model, where we assume a computationally bounded PPT adversary who faithfully follows the protocol, but tries to learn additional information from the messages seen during the protocol execution. In contrast, the stronger *malicious* adversaries are allowed to deviate arbitrarily from the protocol to disrupt it or learn information. Although more restrictive than the malicious adversary model, the semi-honest adversary model also finds many applications: for example, to protect against passive insider attacks by administrators or government agencies, or in situations where the parties can be trusted to not actively misbehave. The semi-honest model enables the development of highly efficient secure computation protocols and is therefore widely used to model privacy-preserving applications which require efficient realizations. Our work concentrates on the design and implementation of an efficient mixed-protocol for the three-party setting, in the semi-honest adversary model.

1.5.3 Notation

We summarize the general notations used in our protocol in this section. Throughout this work, we refer to the protocol in [9] as ABY (Arithmetic, Boolean, Yao) and our new protocol as BAG (Boolean, Arithmetic, Garbled). We denote a set of $n = 3$ parties as $\mathcal{P} = \{P_1, P_2, P_3\}$. We write $x \oplus y$ for bitwise XOR and $x \wedge y$ for bitwise AND. We use the subscript operator $x[i]$ to refer to the i-th element of a list x. In particular, if x is a sequence of bits, $x[i]$ is the $(i+1)$-th bit of x (from right-to-left) and $x[0]$ is the least-significant bit of x.

We denote the cryptographic security parameter by κ. A negligible function in κ is denoted by $\texttt{negl}(\kappa)$. A function $\texttt{negl}(\cdot)$ is *negligible* if for every polynomial $p(\cdot)$ there exists a value N such that for all $m > N$ it holds that $\texttt{negl}(m) < \frac{1}{p(m)}$.

We denote a shared variable x as $\langle x \rangle^t$. The superscript $t \in \{B, A, G\}$ indicates the *type* of sharing, where B denotes Boolean sharing, A denotes Arithmetic sharing, and G denotes Garbled sharing. The semantics of the different sharing types and operations are defined in Section 2.1 and Section 2.2. We refer to the individual share of $\langle x \rangle^t$ that is held by party P_i as $\langle x \rangle^t_i$.

1.5.4 Oblivious Transfer

Oblivious Transfer (OT) is perhaps the most fundamental primitive in the theory of cryptographic MPC protocols. It is a two-party protocol between a sender S and a receiver R. The sender holds an array of inputs and the receiver holds an index indicating its intended pick from the sender's array. OT allows the sender to send the receiver's selected input while preserving the secrecy of (i) the sender's other inputs and (ii) the choice of the receiver.

Of particular interest to the cryptographic community is the following variant of OT: In a 1-out-of-2 OT [10], S holds two inputs x_0, x_1, and R holds a *choice bit* b. The output to R is x_b and neither party learns any additional information. In this work, we use 1-out-of-4 OT [7, 20] in the process of generating boolean multiplication triples.

1.5.5 Garbling Scheme

Garbling Schemes, traditionally used as a technique in secure protocols, were formalized as a primitive by Bellare et al. [2] and were assigned well-defined notions of security, namely *correctness*, *privacy*, *obliviousness*, and *authenticity*. This terminology has largely been adopted by works that followed in this domain [15, 22, 13].

A garbling scheme $\mathcal{G}$ is characterized by a tuple of PPT algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ described below. With the exception of Gb, they are all deterministic.

- $\mathsf{Gb}\left(1^\kappa, C\right)$ is invoked on a circuit C in order to produce a 'garbled circuit' $\mathbf{C}$, 'input encoding information' e, and 'output decoding information' d.
- $\mathsf{En}\left(x, e\right)$ encodes a clear input x with encoding information e in order to produce an encoded input $\mathbf{X}$.
- $\mathsf{Ev}\left(\mathbf{C}, \mathbf{X}\right)$ evaluates $\mathbf{C}$ on $\mathbf{X}$ to produce an encoded output $\mathbf{Y}$.
- $\mathsf{De}\left(\mathbf{Y}, d\right)$ translates $\mathbf{Y}$ into a clear output y as per decoding information d.

In the context of garbled circuits, in addition to *correctness* and *privacy*, the two security properties of *obliviousness* and *authenticity* are also important. Obliviousness captures the notion that when the decoding information is withheld, the garbled circuit evaluation leaks no information about *any* underlying clear values; be they of the input, intermediate, or output wires of the circuit. Authenticity enforces that the evaluator can only learn the output label that corresponds to the value of the function.

Chapter 2

BAG protocol

In this chapter, we describe our new three-party protocol in detail. As stated earlier, BAG stands for Boolean, Arithmetic, Garbled. We will define these three sharing schemes in detail and the conversions between them. Our protocol is in the dishonest majority setting and tolerates a semi-honest centralized static adversary. 'Dishonest majority' means that we allow at most two out of the three parties (a majority) to be corrupt. We assume a synchronous network, so that all communication should be complete at the end of each round.

2.1 Sharing Types

Here, we describe the sharing types that our BAG protocol use, namely the **B**oolean sharing, the **A**rithmetic sharing, and the **G**arbled sharing. For each sharing type, we describe the semantics of the sharing, standard operations and the state of the art in the respective subsections.

2.1.1 Arithmetic Sharing

For the Arithmetic sharing, we assume that each ℓ-bit value is either represented as an element of the ring $\mathbb{Z}_{2^\ell}$ or as as an element of finite field GF(2^ℓ) and shared additively. Later we will show that depending upon the underlying algebraic structure over which we perform the arithmetic operations, the cost of converting from one representation to another will vary. More specifically, if we perform the arithmetic operations over GF(2^ℓ), then the conversion from Boolean sharing to arithmetic sharing and vice-versa will be completely free, as the addition over GF(2^ℓ) is the same as XOR operation. On the other hand, the conversions are not free if we perform arithmetic operations over $\mathbb{Z}_{2^\ell}$, where all operations are done modulo $\mathbb{Z}_{2^\ell}$. The choice of using GF(2^ℓ) or $\mathbb{Z}_{2^\ell}$ depends upon the ease with which the underlying computation can be represented as a circuit over these algebraic structures. In the following discussion, we use the notation $\mathbb{F}$, which could be either GF(2^ℓ) or $\mathbb{Z}_{2^\ell}$; accordingly the operations "$+$" and "$\cdot$" should be interpreted.

1) Sharing Semantics: Arithmetic sharing is based on additively sharing private values between the parties as follows.

Shared values. An ℓ-bit value x is arithmetic shared as $\langle x \rangle^A = \langle x \rangle_1^A + \langle x \rangle_2^A + \langle x \rangle_3^A$, with $\langle x \rangle_1^A, \langle x \rangle_2^A, \langle x \rangle_3^A \in \mathbb{F}$.

Sharing. $\mathsf{Shr}_i^A(x)$: Party P_i considers the ℓ-bit input x as an element of $\mathbb{F}$ and chooses $r_1, r_2, r_3 \in_R \mathbb{F}$, such that $r_1 + r_2 + r_3 = x$. Party P_i sends r_j to each $P_j \in \mathcal{P}$, who sets $\langle x \rangle_j^A = r_j$.

Reconstuction. $\mathsf{Rec}_i^A(x)$: Each $P_j \in \mathcal{P}$ sends its share $\langle x \rangle_j^A$ to P_i, who computes $x = \langle x \rangle_1^A + \langle x \rangle_2^A + \langle x \rangle_3^A$

2) Operations: We assume that the parties want to evaluate an Arithmetic circuit over $\mathbb{F}$, consisting of linear (addition) and non-linear (multiplication) gates over $\mathbb{F}$, which are evaluated over Arithmetic-shared values in the following fashion.

Addition. $\langle z \rangle^A = \langle x \rangle^A + \langle y \rangle^A$: Given $\langle x \rangle_i^A$ and $\langle y \rangle_i^A$, each party $P_i \in \mathcal{P}$ locally computes $\langle z \rangle_i^A = \langle x \rangle_i^A + \langle y \rangle_i^A$.

Multiplication. $\langle z \rangle^A = \langle x \rangle^A \cdot \langle y \rangle^A$: Given $\langle x \rangle_i^A$ and $\langle y \rangle_i^A$ for $i = 1, \ldots, 3$, the multiplication gate is evaluated using a precomputed, Arithmetic-shared, random multiplication-triple $(\langle a \rangle^A, \langle b \rangle^A, \langle c \rangle^A)$ with $c = a \cdot b$ as follows: each P_i computes $\langle e \rangle_i^A = \langle a \rangle_i^A + \langle x \rangle_i^A$ and $\langle f \rangle_i^A = \langle b \rangle_i^A + \langle y \rangle_i^A$, followed by performing $\mathsf{Rec}^A(e)$ and $\mathsf{Rec}^A(f)$. Now if $\mathbb{F} = \mathrm{GF}(2^\ell)$, then each P_i then sets $\langle z \rangle_i^A = (e \cdot f) + \left(f \cdot \langle a \rangle_i^A\right) + \left(e \cdot \langle b \rangle_i^A\right) + \langle c \rangle_i^A$. On the other hand, if $\mathbb{F} = \mathbb{Z}_{2^\ell}$, then P_1 sets $\langle z \rangle_1^A = (e \cdot f) + \left(f \cdot \langle a \rangle_1^A\right) + \left(e \cdot \langle b \rangle_1^A\right) + \langle c \rangle_1^A$, while $P_i \in \{P_2, P_3\}$ sets $\langle z \rangle_i^A = \left(f \cdot \langle a \rangle_i^A\right) + \left(e \cdot \langle b \rangle_i^A\right) + \langle c \rangle_i^A$.

(4) Generating Arithmetic-shared Random Multiplication Triples: In the offline phase, we generate Arithmetic sharings $(\langle a \rangle^A, \langle b \rangle^A, \langle c \rangle^A)$ of random multiplication triples (a, b, c), with $c = a \cdot b$, using the method of [11], which is further based on C-OTs. We explain the idea with respect to the generation of one multiplication triple; the same method can be executed in parallel to simultaneously generate several multiplication triples. Moreover, we explain the method assuming $\mathbb{F} = \mathrm{GF}(2^\ell)$. The parties first generate $\langle a \rangle^A$ and $\langle b \rangle^A$, by each party $P_i \in \mathcal{P}$ sampling $a_i, b_i \in_R \mathrm{GF}(2^\ell)$ and setting $\langle a \rangle_i^A = a_i$ and $\langle b \rangle_i^A = b_i$, with $a \stackrel{def}{=} a_1 + a_2 + a_3$ and $b \stackrel{def}{=} b_1 + b_2 + b_3$. As at least one honest party in $\mathcal{P}$ is honest who samples a uniformly random a_i, b_i, it holds that the resultant a and b are truly random. To generate $\langle a \cdot b \rangle^A$, we note that $a \cdot b = (a_1 + a_2 + a_3) \cdot (b_1 + b_2 + b_3)$. Each party P_i can locally com-

pute the term $a_i \cdot b_i$ in the above expression. The challenge is to securely compute the "cross terms" $a_i \cdot b_j$, with $i \neq j$, where P_i holds a_i and P_j holds b_j. For this we use the protocol of [11], which allows two parties, say S and R, with private inputs β and α respectively from GF(2^ℓ), to securely compute additive shares $\langle \alpha \cdot \beta \rangle_S^A$ and $\langle \alpha \cdot \beta \rangle_R^A$ respectively, such that $\langle \alpha \cdot \beta \rangle_S^A + \langle \alpha \cdot \beta \rangle_R^A = \alpha \cdot \beta$ holds. In our context, each pair of parties (P_i, P_j) with $P_i \neq P_j$, executes an instance of the protocol of [11] (by playing the role of S and R) to generate the additive sharing of the mixed term $a_i \cdot b_j$. The details follow.

Let $(\alpha_0, \ldots, \alpha_{\ell-1}) \in \{0,1\}^\ell$ be the "bit-decomposition" of the element $\alpha \in$ GF(2^ℓ). This implies that in GF(2^ℓ), the element $\alpha = \alpha_0 \cdot X^0 + \alpha_1 \cdot X + \ldots + \alpha_{\ell-1} \cdot X^{\ell-1}$. Hence the product $\alpha \cdot \beta$ can be written as $(\alpha_0 \cdot X^0 + \alpha_1 \cdot X + \ldots + \alpha_{\ell-1} \cdot X^{\ell-1}) \cdot \beta$, which can be further expressed as $\alpha_0 \cdot \beta \cdot X^0 + \ldots + \alpha_{\ell-1} \cdot \beta \cdot X^{\ell-1}$. The idea of the secure multiplication protocol of [11] is to securely generate additive-sharing of each of the terms $\alpha_i \cdot \beta \cdot X^i$, for $i = 0, \ldots, \ell-1$, using ℓ instances of C-OT. More specifically, in the ith instance of C-OT, S acts as the sender with pair of co-related input bit-strings $(s_i, \beta \cdot X^i + s_i)$, where $s_i \in_R$ GF(2^ℓ), while R acts as the receiver with α_i as the choice bit. It follows from the property of C-OT, that at the end, R receives $s_i + \alpha_i \cdot \beta \cdot X^i$, which it can set as its additive share of $\alpha_i \cdot \beta \cdot X^i$, while S can set $-s_i$ as its additive share of $\alpha_i \cdot \beta \cdot X^i$. It now follows easily that $\sum_{i=0}^{\ell-1} s_i + \alpha_i \cdot \beta \cdot X^i$ and $\sum_{i=0}^{\ell-1} -s_i$ constitutes the additive shares $\langle \alpha \cdot \beta \rangle_R^A$ and $\langle \alpha \cdot \beta \rangle_S^A$ for R and S respectively.

If $\mathbb{F} = \mathbb{Z}_{2^\ell}$ instead of GF(2^ℓ), then in the above method, $X^0, \ldots, X^{\ell-1}$ are replaced by $2^0, \ldots, 2^{\ell-1}$ respectively. This is because if $x \in$ GF(2^ℓ) and has bit-decomposition $(x_0, \ldots, x_{\ell-1})$, then $x = x_0 \cdot 2^0 + x_1 \cdot 2^1 + \ldots + x_{\ell-1} \cdot 2^{\ell-1}$.

Complexity: To generate $\langle a \cdot b \rangle^A$, total six instances of the multiplication protocol of [11] need to be executed, as there are six cross terms, in the expansion of $a \cdot b$. Each instance of the multiplication protocol needs ℓ instances of C-OT, dealing with ℓ-bit strings. So in total, 6ℓ instances of C-OT are required to generate one $\langle\rangle^A$-shared multiplication triple.

2.1.2 Boolean Sharing

The Boolean sharing uses an XOR-based secret sharing scheme to share a Boolean variable. We evaluate functions represented as Boolean circuits using the GMW protocol [12]. In the following, we first define the sharing semantics, describe how operations are performed, and give an overview of related work.

1) Sharing Semantics: To simplify the presentation, we assume single bit values; for ℓ-bit values, each operation is performed ℓ times in parallel.

Shared values. Let x be a Boolean value. Then a Boolean-sharing $\langle x\rangle^B$ of x consists of the Boolean shares $\langle x\rangle_1^B, \langle x\rangle_2^B, \langle x\rangle_3^B \in \{0,1\}$, such that $\langle x\rangle_1^B \oplus \langle x\rangle_2^B \oplus \langle x\rangle_3^B = x$ holds.

Sharing. $\mathsf{Shr}_i^B(x)$: Party P_i chooses $r_1, r_2, r_3 \in_R \{0,1\}$, such that $r_1 \oplus r_2 \oplus r_3 = x$ and sends r_j to each $P_j \in \mathcal{P}$, who sets $\langle x\rangle_j^B = r_j$.

Reconstuction. $\mathsf{Rec}_i^B(x)$: Each $P_j \in \mathcal{P}$ sends its share $\langle x\rangle_j^B$ to P_i, who computes $x = \langle x\rangle_1^B \oplus \langle x\rangle_2^B \oplus \langle x\rangle_3^B$.

2) Operations: We imagine that the parties are given a Boolean circuit for evaluation, consisting of XOR and AND gates, which are evaluated over Boolean-shared values in the following fashion.

XOR. $\langle z\rangle^B = \langle x\rangle^B \oplus \langle y\rangle^B$: Given $\langle x\rangle_i^B$ and $\langle y\rangle_i^B$, each party $P_i \in \mathcal{P}$ locally computes $\langle z\rangle_i^B = \langle x\rangle_i^B \oplus \langle y\rangle_i^B$.

AND. $\langle z\rangle^B = \langle x\rangle^B \wedge \langle y\rangle^B$: Given $\langle x\rangle_i^B$ and $\langle y\rangle_i^B$ for $i = 1,2,3$, the AND gate is evaluated using a precomputed, Boolean-shared, random multiplication-triple $(\langle a\rangle^B, \langle b\rangle^B, \langle c\rangle^B)$, with $c = a \wedge b$, as follows: each P_i computes $\langle e\rangle_i^B = \langle a\rangle_i^B \oplus \langle x\rangle_i^B$ and $\langle f\rangle_i^B = \langle b\rangle_i^B \oplus \langle y\rangle_i^B$, followed by performing $\mathsf{Rec}^B(e)$ and $\mathsf{Rec}^B(f)$. Each P_i then sets $\langle z\rangle_i^B = (e \cdot f) \oplus \left(f \cdot \langle a\rangle_i^B\right) \oplus \left(e \cdot \langle b\rangle_i^B\right) \oplus \langle c\rangle_i^B$.

3) State-of-the-Art: The first implementation of the GMW protocol for multiple parties and with security in the semi honest model was given in [7]. GMW protocol achieves good performance in low-latency networks.

(4) Generating Boolean-shared Random Multiplication Triples: In the offline phase, we generate Boolean sharings $(\langle a\rangle^B, \langle b\rangle^B, \langle c\rangle^B)$ of random multiplication triples (a,b,c), with $c = a \wedge b$, using the method of [7, 20], which is further based on 1-out-of-4 OTs. We explain the idea with respect to the generation of one multiplication triple; the same method can be executed in parallel to simultaneously generate several multiplication triples.

The parties first generate $\langle a\rangle^B$ and $\langle b\rangle^B$, where $a, b \in_R \{0,1\}$. For this, each party $P_i \in \mathcal{P}$ samples $a_i, b_i \in_R \{0,1\}$ and set $\langle a\rangle_i^B = a_i$ and $\langle b\rangle_i^B = b_i$ and a and b are defined as $a \stackrel{def}{=} a_1 \oplus a_2 \oplus a_3$ and $b \stackrel{def}{=} b_1 \oplus b_2 \oplus b_3$. Now notice that $a \wedge b = (a_1 \wedge b_1) \oplus (a_2 \wedge b_2) \oplus (a_3 \wedge b_3) \oplus ((a_1 \wedge b_2) \oplus (a_2 \wedge b_1)) \oplus ((a_1 \wedge b_3) \oplus (a_3 \wedge b_1)) \oplus ((a_2 \wedge b_3) \oplus (a_3 \wedge b_2))$. Each $P_i \in \mathcal{P}$ can locally compute the term $a_i \wedge b_i$. The challenge is to securely compute the cross-terms $((a_i \wedge b_j) \oplus (a_j \wedge b_i))$, for each $i, j \in \{1,2,3\}$ with $i < j$, where P_i holds a_i, b_i and P_j holds a_j, b_j. For this, P_i and P_j engages in an instance of 1-out-of-4 OT, to securely generate a Boolean-sharing of $((a_i \wedge b_j) \oplus (a_j \wedge b_i))$ between them. In a more

detail, P_j selects $c_j^{\{i,j\}} \in_R \{0,1\}$ and acts as a sender in an instance of 1-out-of-4 OT, with messages $c_j^{\{i,j\}}, c_j^{\{i,j\}} \oplus a_j, c_j^{\{i,j\}} \oplus b_j$ and $c_j^{\{i,j\}} \oplus a_j \oplus b_j$, while party P_i acts as the receiver with (a_i, b_i) as the selection bits. Let $c_i^{\{i,j\}}$ denote the OT output of P_i; it follows that $c_i^{\{i,j\}} = c_j^{\{i,j\}} \oplus ((a_i \wedge b_j) \oplus (a_j \wedge b_i))$. Now to generate $\langle a \wedge b \rangle^B$, each party P_i sets $\langle a \wedge b \rangle_i^B = (a_i \wedge b_i) \bigoplus_{j \neq i} c_i^{\{i,j\}}$.

Complexity: To generate one Boolean-shared multiplication triple, we require three instances of 1-out-of-4 OT or 6 instances of 1-out-of-2 OT, as there are three cross-terms involved. To generate l-bit multiplication triple, $6l$ instances of 1-out-of-2 OTs required.

2.1.3 Garbled Sharing

The BMR protocol [1] is the generalization of the Yao's protocol to the multiparty case. In our context, given a Boolean circuit, all the three parties jointly garble the circuit, which is evaluated by a designated evaluator, say P_3. In a more detail, for each wire w in the circuit, each party P_i locally associates a pair of random "labels" (keys) $k_{w,0}^i$ and $k_{w,1}^i$, corresponding to the bit value 0 and 1 respectively over w. The concatenated keys $k_{w,0} \stackrel{def}{=} k_{w,0}^1 || \ldots || k_{w,0}^n$ and $k_{w,1} \stackrel{def}{=} k_{w,1}^1 || \ldots || k_{w,1}^n$ are considered as the "0-superkey" and "1-superkey", corresponding to the bit value 0 and 1 respectively over w. The parties then jointly garble each gate, using an encryption function Garble, where the output-wire labels of each gate-output is encrypted separately under each single party's gate-input wire labels. For example, consider a gate g with gate function $g(\cdot,\cdot)$, with input wires u, v and output wire w. And let $k_{u,0}, k_{u,1}, k_{v,0}, k_{v,1}, k_{w,0}$ and $k_{w,1}$ be the superkeys associated with these wires, with each party P_i holding the key pairs $(k_{u,0}^i, k_{u,1}^i)$, $(k_{v,0}^i, k_{v,1}^i)$ and $(k_{w,0}^i, k_{w,1}^i)$ respectively. Then for every $a, b \in \{0,1\}$, the jth gate-output label $k_{w,g(a,b)}^j$ is encrypted as:

$$F^2_{k_{u,a}^1, k_{v,b}^1}(g||j) \oplus F^2_{k_{u,a}^2, k_{v,b}^2}(g||j) \oplus \ldots \oplus F^2_{k_{u,a}^n, k_{v,b}^n}(g||j) \oplus k_{w,g(a,b)}^j$$

Here F^2 denotes a *double-key PRF* [3], which takes two keys and maintains security, as long as at least one key is secret.

To hide which values of a and b are dealt with, a slight modification of the above idea is used. Specifically, a random "permutation bit" λ_u is associated with each wire u, with each party holding a share λ_u^i, such that $\lambda_u = \oplus_{i=1}^n \lambda_u^i$. Then if the actual bits on the gate-input wires u and v are a and b, then the circuit-evaluator sees masked values $\hat{a} = a \oplus \lambda_u$ and $\hat{b} = b \oplus \lambda_v$ and the superkeys $k_{u,\hat{a}}$ and $k_{v,\hat{b}}$. The superkeys "guide" the evaluator regarding which entry in the garbled table to decrypt. Moreover, the garbled gate is computed in such a way that after decrypting the appropriate entry in the table using $k_{u,\hat{a}}$ and $k_{v,\hat{b}}$, the evaluator obtains the superkey $k_{w,\hat{c}}$, where $\hat{c} = g(a,b) \oplus \lambda_w$, using which the evaluator obtains the masked gate-output $\hat{c}$, which enables the evaluator to proceed to evaluate the next gate. This maintains the invariant that for each wire w in the circuit, the evaluator sees masked

value $x_w \oplus \lambda_w$ and the corresponding superkey $k_{w,x_w \oplus \lambda_w}$, where x_w is the actual bit value over w. We call the gate-evaluation process as Eval. Intuitively, even if two out of the three parties are corrupted (possibly including the evaluator P_3), nothing additional is revealed about the actual bit x_w over wire w, even though the superkey $k_{w,x_w \oplus \lambda_w}$ is available with the evaluator for wire w; this is because the actual value of λ_w is hidden from the corrupted parties. Once the circuit-output gate is evaluated, the clear value of the circuit output is reconstructed, using the procedure discussed below.

In the following description, we use the free-XOR technique [16], which is generalized for the multi-party case in [3]. In this technique, for every wire w and for every party P_i, the key-pair $(k^i_{w,0}, k^i_{w,1})$ is related by a global offset R^i, randomly and privately chosen by P_i. More specifically, $k^i_{w,1} = k^i_{w,0} \oplus R^i$ holds. Moreover, if w is the output of an XOR gate with u, v being the gate-input wires, then $k^i_{w,0}$ itself is selected as $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and the permute-bit share λ^i_w is selected as $\lambda^i_w = \lambda^i_u \oplus \lambda^i_v$.

1) Sharing Semantics: In the following, to simplify the presentation, we assume single bit values; for ℓ-bit values each operation is performed ℓ times in parallel.

Shared values. Let $x \in \{0,1\}$. Then a garbled sharing $\langle x \rangle^G$ of x is defined as $\langle x \rangle^G_1 = \lambda^1_x$, $\langle x \rangle^G_2 = \lambda^2_x$ and $\langle x \rangle^G_3 = (\lambda^3_x, \hat{x}, k_{x,\hat{x}})$, with $\lambda^1_x, \lambda^2_x, \lambda^3_x \in \{0,1\}$, such that $\lambda_x = \lambda^1_x \oplus \lambda^2_x \oplus \lambda^3_x$, $\hat{x} = x \oplus \lambda_x$ and $k_{x,\hat{x}} = k^1_{x,\hat{x}} || k^2_{x,\hat{x}} || k^3_{x,\hat{x}}$. Here $(k^j_{x,0}, k^j_{x,1})$ denote the pair of keys *implicitly* available with party $P_j \in \mathcal{P}$, corresponding to the two possible values of x; for simplicity we do not include them explicitly as part of state information, while defining the $\langle\rangle^G$-sharing. In addition, each P_j also holds the offset R^j, which is common for P_j across all the $\langle\rangle^G$-shared values.

Sharing. $\mathsf{Shr}^G_i(x)$: To enable P_i share a bit $x \in \{0,1\}$, the parties do the following: each party $P_j \in \mathcal{P}$ samples $(k^j_{x,0}, k^j_{x,1}, \lambda^j_x)$, where $k^j_{x,0} \in_R \{0,1\}^\kappa$, $\lambda^j_x \in_R \{0,1\}$ and $k^j_{x,1} = k^j_{x,0} \oplus R^j$. Each P_j then sends λ^j_x to P_i, who computes $\lambda_x = \lambda^1_x \oplus \lambda^2_x \oplus \lambda^3_x$ and broadcasts $\hat{x} = x \oplus \lambda_x$. On receiving the masked bit $\hat{x}$, each P_j sends $k^j_{x,\hat{x}}$ to party P_3, who computes $k_{x,\hat{x}} = k^1_{x,\hat{x}} || k^2_{x,\hat{x}} || k^3_{x,\hat{x}}$. For $j = 1, 2$, party P_j sets $\langle x \rangle^G_j = \lambda^j_x$, while P_3 sets $\langle x \rangle^G_3 = (\lambda^3_x, \hat{x}, k_{x,\hat{x}})$.

Reconstuction. $\mathsf{Rec}^G_i(x)$: Let $x \in \{0,1\}$ be $\langle\rangle^G$-shared, with P_1, P_2 and P_3 holding $\langle x \rangle^G_1 = \lambda^1_x$, $\langle x \rangle^G_2 = \lambda^2_x$ and $\langle x \rangle^G_3 = (\lambda^3_x, \hat{x}, k_{x,\hat{x}})$ respectively. To enable party P_i reconstruct x, each party $P_j \in \mathcal{P}$ sends λ^j_x to P_i. In addition, P_3 also sends $\hat{x}$ to P_i, who then outputs $x = \hat{x} \oplus \lambda^1_x \oplus \lambda^2_x \oplus \lambda^3_x$.

2) Operations: We assume that the parties are given a publicly known Boolean circuit, consisting of XOR and AND gates, which is evaluated over $\langle\rangle^G$-shared values as follows: let x and y be $\langle\rangle^G$-shared, with party P_1 holding $\langle x \rangle^G_1 = \lambda^1_x$ and $\langle y \rangle^G_1 = \lambda^1_y$, party P_2 holding $\langle x \rangle^G_2 = \lambda^2_x$ and $\langle y \rangle^G_2 = \lambda^2_y$

and party P_3 holding $\langle x \rangle_3^G = (\lambda_x^3, \hat{x}, k_{x,\hat{x}})$ and $\langle y \rangle_3^G = (\lambda_y^3, \hat{y}, k_{y,\hat{y}})$.

XOR. $\langle z \rangle^G = \langle x \rangle^G \oplus \langle y \rangle^G$: each party $P_j \in \mathcal{P}$ locally sets $\langle z \rangle_j^G = \langle x \rangle_j^G \oplus \langle y \rangle_j^G$.

AND. $\langle z \rangle^G = \langle x \rangle^G \wedge \langle y \rangle^G$ is computed as follows: corresponding to z, each party $P_j \in \mathcal{P}$ samples $(k_{z,0}^j, k_{z,1}^j, \lambda_z^j)$, where $k_{z,0}^j \in_R \{0,1\}^\kappa$, $\lambda_z^j \in_R \{0,1\}$ and $k_{z,1}^j = k_{z,0}^j \oplus R^j$. The parties then jointly construct a garbled table for the AND gate by executing the protocol Garble [3], where the input for P_j is $\langle x \rangle_j^G, \langle y \rangle_j^G$ and $(k_{z,0}^j, k_{z,1}^j, \lambda_z^j)$ and the table is reconstructed towards P_3. For $j = 1, 2$, party P_j sets $\langle z \rangle_j^G = \lambda_z^j$, while party P_3 evaluates the garbled table using Eval, with inputs $\langle x \rangle_3^G, \langle y \rangle_3^G$ and λ_z^3, to obtain $\langle z \rangle_3^G$. We next provide the details of Eval and Garble.

(3) Protocol Garble: We recall the protocol Garble for constructing garbled gates from [3]. Since we are using the free-XOR technique, the parties need to garble only AND gates. Let g be an AND gate, with input wires x, y and output wire z. The input for the party $P_i \in \mathcal{P}$ is $(k_{x,0}^i, k_{x,1}^i, k_{y,0}^i, k_{y,1}^i, k_{z,0}^i, k_{z,1}^i, \lambda_x^i, \lambda_y^i, \lambda_z^i, R^i)$, where $k_{x,1}^i = k_{x,0}^i \oplus R^i$, $k_{y,1}^i = k_{y,0}^i \oplus R^i$ and $k_{z,1}^i = k_{z,0}^i \oplus R^i$. Moreover, $k_{x,0}^i, k_{y,0}^i$ and λ_x^i, λ_y^i are selected in a special way, if wires x or y is an output of an XOR gate, to enable free evaluation of XOR gate, during Eval. Using Garble, the parties jointly create a garble table $\tilde{g} = (\tilde{g}_{00}, \tilde{g}_{01}, \tilde{g}_{10}, \tilde{g}_{11})$ for P_3, where for every $\hat{x}, \hat{y} \in \{0,1\}$, the garbled entry $\tilde{g}_{\hat{x}\hat{y}} = (\tilde{g}_{\hat{x}\hat{y}}^1 || \tilde{g}_{\hat{x}\hat{y}}^2 || \tilde{g}_{\hat{x}\hat{y}}^3)$. Moreover, for $j \in \{1,2,3\}$, the value $\tilde{g}_{\hat{x}\hat{y}}^j$ is of the form:

$$\tilde{g}_{\hat{x}\hat{y}}^j = \left(\bigoplus_{i=1}^{3} F^2_{k^i_{x,\hat{x}}, k^i_{y,\hat{y}}}(g||j) \right) \oplus k_{z,0}^j \oplus \left(R^j \wedge ((\lambda_u \oplus \hat{x}) \wedge (\lambda_y \oplus \hat{y}) \oplus \lambda_z) \right), \tag{2.1}$$

where $\lambda_x \overset{def}{=} \lambda_x^1 \oplus \lambda_x^2 \oplus \lambda_x^3$, $\lambda_y \overset{def}{=} \lambda_y^1 \oplus \lambda_y^2 \oplus \lambda_y^3$ and $\lambda_z \overset{def}{=} \lambda_z^1 \oplus \lambda_z^2 \oplus \lambda_z^3$. Before going into the details of how the garbled table $\tilde{g}$ is jointly constructed, let us first understand the computation of $\tilde{g}_{\hat{x}\hat{y}}^j$. We first recall that the BMR protocol maintains the invariant that during the circuit evaluation (namely protocol Eval), for wires x and y, the evaluator P_3 sees the masked value $\hat{x} = x \oplus \lambda_x$ and $\hat{y} = y \oplus \lambda_y$ and holds the superkey $k_{x,\hat{x}} = k_{x,\hat{x}}^1 || k_{x,\hat{x}}^2 || k_{x,\hat{x}}^3$ and $k_{y,\hat{y}} = k_{y,\hat{y}}^1 || k_{y,\hat{y}}^2 || k_{y,\hat{y}}^3$. (Notice that this does not reveal anything to the evaluator about the actual value of x and y, as λ_x and λ_y are unknown.) Now the actual value of z will be $(\hat{x} \oplus \lambda_x) \wedge (\hat{y} \oplus \lambda_y)$ and we need to ensure that $\tilde{g}_{\hat{x}\hat{y}}^j$ should be such that, later by decrypting $\tilde{g}_{\hat{x}\hat{y}}^j$ using the superkeys $k_{x,\hat{x}}$ and $k_{y,\hat{y}}$ (using the protocol Eval), the evaluator P_3 should obtain the key $k_{z,\hat{z}}^j$, where $\hat{z} = ((\hat{x} \oplus \lambda_x) \wedge (\hat{y} \oplus \lambda_y)) \oplus \lambda_z$; this will further ensure that by decrypting the entire entry $\tilde{g}_{\hat{x}\hat{y}}$ using the superkeys $k_{x,\hat{x}}$ and $k_{y,\hat{y}}$, evaluator P_3 obtains $k_{z,\hat{z}}^1$, $k_{z,\hat{z}}^2$ and $k_{z,\hat{z}}^3$ and hence the superkey $k_{z,\hat{z}} = k_{z,\hat{z}}^1 || k_{z,\hat{z}}^2 || k_{z,\hat{z}}^3$, which further helps to maintain the required invariant for the next gates.

Now let us consider the value $\tilde{g}_{\hat{x}\hat{y}}^j$, as computed in Equation 2.1. There are two cases: if $\lambda_z = 0$

(and hence $\hat{z} = z = (\hat{x} \oplus \lambda_x) \wedge (\hat{y} \oplus \lambda_y)$); then depending upon whether z is 0 or 1, the value $\tilde{g}^j_{\hat{x}\hat{y}}$ will be either an encryption of $k^j_{z,0}$ or $k^j_{z,1}$. On the other hand, consider the case when $\lambda_z = 1$ (and hence $\hat{z} = z \oplus 1$); in this case, the result will be reversed. Hence, the value $\tilde{g}^j_{\hat{x}\hat{y}}$, as computed in Equation 2.1, ensures to maintain the desired variant.

In order to enable P_3 construct the garbled value $\tilde{g}^j_{\hat{x}\hat{y}}$, the parties execute the protocol Garble, which is an MPC protocol to allow P_3 to securely compute Equation 2.1. Intuitively, in Garble, the parties generate secret-shares of $R^j \wedge ((\lambda_u \oplus \hat{x}) \wedge (\lambda_y \oplus \hat{y}) \oplus \lambda_z)$ (by using an MPC protocol), from which they further generate secret-shares of $\tilde{g}^j_{\hat{x}\hat{y}}$, which are sent to P_3, who can then reconstruct $\tilde{g}^j_{\hat{x}\hat{y}}$. The key insight is that the circuit for computing $R^j \wedge ((\lambda_u \oplus \hat{x}) \wedge (\lambda_y \oplus \hat{y}) \oplus \lambda_z)$ has a *constant* depth and hence can be efficiently computed using *any* dis-honest majority MPC. In [3], a highly optimized and efficient protocol has been proposed to securely compute the garbled table.

4) Protocol Eval: Let g be an AND gate, with input wires x, y and output wire z, which is garbled as per the protocol Garble. And let $\tilde{g} = (\tilde{g}_{00}, \tilde{g}_{01}, \tilde{g}_{10}, \tilde{g}_{11})$ be the garbled gate, obtained by P_3 at the end of Garble. Moreover, P_3 has the superkeys $k_{x,\hat{x}}, k_{y,\hat{y}}$ and the masked values $\hat{x}, \hat{y}$, where $k_{x,\hat{x}} = k^1_{x,\hat{x}}||k^2_{x,\hat{x}}||k^3_{x,\hat{x}}$ and $k_{y,\hat{y}} = k^1_{y,\hat{y}}||k^2_{y,\hat{y}}||k^3_{y,\hat{y}}$. Moreover, corresponding to z, party P_3 also has λ^3_z and $(k^3_{z,0}, k^3_{z,1})$ available with it (recall that during Garble, these values are picked by P_3 itself for garbling the gate g). Now using the procedure Eval, the evaluator P_3 evaluates the gate $\tilde{g}$ as follows: based on the value of $\hat{x}$ and $\hat{y}$, party P_3 picks the entry $\tilde{g}^j_{\hat{x}\hat{y}}$ for decryption. Let $\tilde{g}_{\hat{x}\hat{y}} = (\tilde{g}^1_{\hat{x}\hat{y}}||\tilde{g}^2_{\hat{x}\hat{y}}||\tilde{g}^3_{\hat{x}\hat{y}})$. Then using $k_{x,\hat{x}}$ and $k_{y,\hat{y}}$, party P_3 computes the following for every $j \in \{1, 2, 3\}$:

$$k^j_{z,\hat{z}} = \tilde{g}_{\hat{x}\hat{y}} \oplus \left(\bigoplus_{i=1}^{n} F^2_{k^i_{x,\hat{x}}, k^i_{y,\hat{y}}}(g||j) \right).$$

Party P_3 then compares $k^3_{z,\hat{z}}$ with $k^3_{z,0}$ and $k^3_{z,1}$, and accordingly sets $\hat{z} = 0$ or 1. Finally, P_3 outputs λ^3_z, $\hat{z}$ and $k_{z,\hat{z}} = k^1_{z,\hat{z}}||k^2_{z,\hat{z}}||k^3_{z,\hat{z}}$ as its outcome for Eval. For the correctness and security of Eval, see [3].

2.2 Sharing Conversions

We now discuss how to convert between different sharings. For this, we consider two cases, depending upon whether for the arithmetic sharing, $\mathbb{F} = \text{GF}(2^\ell)$ or $\mathbb{F} = \mathbb{Z}_{2^\ell}$. We first consider the case when $\mathbb{F} = \text{GF}(2^\ell)$, as in this case, most of the conversions have al most zero cost during the online phase when the actual conversion needs to happen.

2.2.1 Sharing Conversions when $\mathbb{F} = \mathrm{GF}(2^\ell)$ in the Arithmetic World

Garbled to Boolean Sharing (G2B): Converting a $\langle\rangle^G$-shared bit to its $\langle\rangle^B$-sharing is free and involves only local computation. More specifically, let $x \in \{0,1\}$ be $\langle\rangle^G$-shared, with P_1 holding $\langle x\rangle_1^G = \lambda_x^1$, party P_2 holding $\langle x\rangle_2^G = \lambda_x^2$ and party P_3 holding $\langle x\rangle_3^G = (\lambda_x^3, \hat{x}, k_{x,\hat{x}})$. To obtain a Boolean-sharing $\langle x\rangle^B$ of x, we notice that P_1 can set $\langle x\rangle_1^B = \langle x\rangle_1^G$, party P_2 can set $\langle x\rangle_2^B = \langle x\rangle_2^G$, while party P_3 can set $\langle x\rangle_3^B = \hat{x} \oplus \lambda_x^3$. The correctness simply follows from the fact that for the $\langle x\rangle^G$-sharing, the condition $\hat{x} = x \oplus \lambda_x^1 \oplus \lambda_x^2 \oplus \lambda_x^3$ holds.

Boolean to Garbled Sharing (B2G): Before presenting our B2G conversion, we introduce a notation. Let $b \in \{0,1\}$. We say that b is BG-shared, represented as $(\langle b\rangle^B, \langle b\rangle^G)$, if the bit b is both $\langle\rangle^B$-shared as well as $\langle\rangle^G$-shared. In our B2G conversion, we require a BG-sharing of a random and private $m \in \{0,1\}$, for each conversion. Such BG-sharings can be precomputed in the offline phase as follows: to compute one random BG-sharing, the parties first compute $\langle m\rangle^G$, for a random and private $m \in \{0,1\}$. For this, each party $P_i \in \mathcal{P}$ selects a random bit $m^{(i)} \in \{0,1\}$ and the parties execute $\mathsf{Shr}_i^G(m^{(i)})$ to enable P_i garble-share $m^{(i)}$. Let $\langle m^{(i)}\rangle^G$ be the resultant garbled-sharing. The parties then set $\langle m\rangle^G = \langle m^{(1)}\rangle^G \oplus \langle m^{(2)}\rangle^G \oplus \langle m^{(3)}\rangle^G$. As our G2B conversion is free, the parties set $\langle m\rangle^B = \mathrm{G2B}(\langle m\rangle^G)$, as a result of which the parties obtain $(\langle m\rangle^B, \langle(\rangle^G m))$. We next explain our B2G conversion.

Let $x \in \{0,1\}$ be $\langle\rangle^B$-shared. To obtain a $\langle\rangle^G$-sharing of x, the parties deploy a $(\langle m\rangle^B, \langle m\rangle^G)$-shared random $m \in \{0,1\}$, generated in the offline phase. The idea is to first make masked x public, using m as the mask, followed by taking a default $\langle\rangle^G$-sharing of the masked x. Since the mask m is also available in $\langle\rangle^G$-shared fashion, the effect of mask can be now locally removed from the $\langle\rangle^G$-sharing of the masked x, resulting in a $\langle\rangle^G$-sharing of x. More formally, the parties first compute $\langle x \oplus m\rangle^B = \langle x\rangle^B \oplus \langle m\rangle^B$, followed by executing $\mathsf{Rec}^B(x \oplus m)$, to publicly reconstruct $x \oplus m$. The parties then take a default $\langle\rangle^G$-sharing $\langle x \oplus m\rangle^G$ of [1] $x \oplus m$, followed by computing $\langle x\rangle^G = \langle x \oplus m\rangle^G \oplus \langle m\rangle^G$. The correctness is obvious and privacy follows from the fact that m is a random and private value. The online complexity of this conversion is equivalent to the cost of public reconstruction of an ℓ-bits $\langle\rangle^B$-sharing.

Arithmetic to Boolean (A2B) and Boolean to Arithmetic Sharing (B2A): Since in $\mathrm{GF}(2^\ell)$ the XOR operation is same as addition, both A2B as well as B2A conversions are free. More specifically, consider an ℓ-bit value $x \in \mathrm{GF}(2^\ell)$, which is $\langle\rangle^A$-shared, with $\langle x\rangle_1^A = x_1, \langle x\rangle_2^A = x_2$ and $\langle x\rangle_3^A = x_3$,

[1] A default $\langle\rangle^G$-sharing of a publicly known bit $b \in \{0,1\}$ can be computed as follows: the parties publicly set $\lambda_b^1 = \lambda_b^2 = \lambda_b^3 = 0$ (which implies that $\lambda_b = \bigoplus_{i=1}^3 \lambda_b^i = 0$ and hence the masked bit $\hat{b} = b \oplus \lambda_b = b$). The parties also publicly set $k_{b,\hat{b}}^1 = k_{b,\hat{b}}^2 = k_{b,\hat{b}}^3 = \{0,1\}^\kappa$. Party P_1 sets $\langle b\rangle_1^G = \lambda_b^1$, party P_2 sets $\langle b\rangle_2^G = \lambda_b^2$ and party P_3 sets $\langle b\rangle_3^G = (\lambda_b^3, \hat{b}, k_{b,\hat{b}})$, where $k_{b,\hat{b}} = k_{b,\hat{b}}^1 || k_{b,\hat{b}}^2 || k_{b,\hat{b}}^3$. In addition, each P_i sets $k_{b,1\oplus\hat{b}}^i = R^i$.

such that $x = x_1 + x_2 + x_3$ holds in GF(2^ℓ). Now as per the definition of addition over GF(2^ℓ), it holds that $x^B = x_1^B \oplus x_2^B \oplus x_3^B$, where x^B, x_1^B, x_2^B and x_3^B are ℓ-bit bit-decomposition of x, x_1, x_2 and x_3 respectively. So parties can set $\langle x \rangle_1^B = x_1^B$, $\langle x \rangle_2^B = x_2^B$ and $\langle x \rangle_3^B = x_3^B$ and hence the A2B conversion is completely free.

Due to the same argument, even B2A conversion is free. Namely, let $x \in \{0,1\}^\ell$ be an ℓ-bit string, which is $\langle \rangle^B$-shared, with $\langle x \rangle_1^B = x_1, \langle x \rangle_2^B = x_2$ and $\langle x \rangle_3^B = x_3$, with $x_1, x_2, x_3 \in \{0,1\}^\ell$, such that $x = x_1 \oplus x_2 \oplus x_3$ holds. Now as per the definition of addition over GF(2^ℓ), it holds that $x^A = x_1^A + x_2^A + x_3^A$, where x^A, x_1^A, x_2^A and x_3^A are the encodings of the bit-strings x, x_1, x_2 and x_3 respectively, as elements of GF(2^ℓ). So parties can set $\langle x \rangle_1^A = x_1^A$, $\langle x \rangle_2^A = x_2^A$ and $\langle x \rangle_3^A = x_3^A$ and hence the B2A conversion is completely free.

Garbled to Arithmetic Sharing (G2A): In our framework, where both G2B as well as B2A conversion are free, the parties can freely convert $\langle \rangle^G$-shared ℓ-bit string x to its $\langle \rangle^A$-sharing as $\langle x \rangle^A$ = B2A(G2B($\langle x \rangle^G$)).

Arithmetic to Garbled Sharing (A2G): To convert an arithmetic sharing $\langle x \rangle^A$ to its garbled-sharing, we first apply the A2B conversion, which is free. The intermediate Boolean-sharing is then converted to a garbled-sharing, using our B2G conversion. Hence $\langle x \rangle^G$ = B2G(A2B($\langle x \rangle^A$)).

2.2.2 Sharing Conversions when $\mathbb{F} = \mathbb{Z}_{2^\ell}$ in the Arithmetic World

Garbled to Boolean Sharing (G2B) and Boolean to Garbled Sharing (B2G): The conversions here are exactly the same as in the case of $\mathbb{F} =$ GF(2^ℓ).

Arithmetic to Garbled Sharing (A2G): To convert an arithmetic sharing over $\mathbb{Z}_{2^\ell}$ to its garbled sharing, we use the same idea as used in [9]. More specifically, let $x \in \mathbb{Z}_{2^\ell}$, which is arithmetic-shared as $\langle x \rangle^A$, with each P_i holding its share $\langle x \rangle_i^A = x_i \in \mathbb{Z}_{2^\ell}$. To compute a garbled sharing $\langle x \rangle^G$ of x, each party P_i garbled-share its share $\langle x \rangle_i^A$ as $\langle x_i \rangle^G = \mathsf{Shr}_i^G(x_i)$. Let Adder be an ℓ-bit Boolean adder, whose inputs are ℓ-bit x_1, x_2, x_3 and output is $(x_1 + x_2 + x_3 \mod 2^\ell) = x$. The parties in $\mathcal{P}$ then jointly evaluate evaluate Adder and obtain $\langle x \rangle^G = \mathsf{Adder}(\langle x_1 \rangle^G, \langle x_2 \rangle^G, \langle x_3 \rangle^G)$.

Arithmetic to Boolean Sharing (A2B): Let $\langle x \rangle^A$ be an arithmetic sharing over $\mathbb{Z}_{2^\ell}$. As the G2B conversion is for free, we simply convert $\langle x \rangle^A$ to a corresponding Boolean sharing as $\langle x \rangle^B$ = G2B(A2G($\langle x \rangle^A$)).

Boolean to Arithmetic Sharing (B2A) : Let $x \in \{0,1\}^\ell$ be an ℓ-bit value, which is Boolean-shared as $\langle x \rangle^B$, with each party $P_i \in \mathcal{P}$ holding its share $\langle x \rangle_i^B = x_i \in \{0,1\}^\ell$, such that $x = x_1 \oplus x_2 \oplus x_3$. Let $(x_{\ell-1}, \ldots, x_0)$, $(x_{1,\ell-1}, \ldots, x_{1,0})$, $(x_{2,\ell-1}, \ldots, x_{2,0})$ and $(x_{3,\ell-1}, \ldots, x_{3,0})$ be the bit-decomposition of x, x_1, x_2 and x_3 respectively. Then in the arithmetic world, we have $x = \sum_{i=0}^{\ell-1} 2^i \cdot (x_{1,i} \oplus x_{2,i} \oplus x_{3,i})$. The goal is to generate an arithmetic-sharing $\langle x \rangle^A$ of x. For this, we extend the idea used in [9] for the two party case; however extending the idea to the three party case is non-trivial. The idea behind the

conversion is to generate arithmetic-sharing of each of the ℓ terms $2^i \cdot (x_{1,i} \oplus x_{2,i} \oplus x_{3,i})$, using instances of OT. For this, we present a generic sub-protocol BA, where the inputs for P_1, P_2 and P_3 are a, b and c respectively, with $a, b, c \in \{0, 1\}$. In addition, there is a public constant $\alpha \in \{0, \ldots, \ell - 1\}$. The protocol securely generates arithmetic sharing $\langle 2^\alpha \cdot (a \oplus b \oplus c) \rangle^A$. Namely, it outputs shares s_1, s_2 and s_3 for P_1, P_2 and P_3 respectively, such that $s_1 + s_2 + s_3 = 2^\alpha \cdot (a \oplus b \oplus c)$. The protocol involves three instances of 1-out-of-2 OT, whose details are as follows:

OT_1**:** Here P_3 acts as the sender, with input pair $(2^\alpha \cdot c - r_3, 2^\alpha \cdot (1 \oplus c) - r_3)$, where r_3 is sampled randomly from $\mathbb{Z}_{2^\ell}$, while party P_1 participates as the receiver, with a as its choice bit. It follows from the property of OT that at the end, P_1 obtains the output $y_1 = 2^\alpha \cdot (a \oplus c) - r_3$.

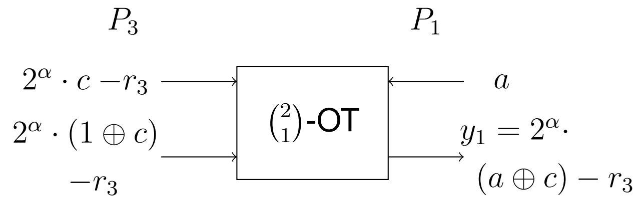

Figure 2.1: OT_1 between P_3 and P_1

OT_2**:** Here P_3 acts as the sender, with input pair $(r_2, r_2 + r_3)$, where r_2 is sampled randomly from $\mathbb{Z}_{2^\ell}$, while party P_2 participates as the receiver, with b as its choice bit. It follows from the property of OT that at the end, P_2 obtains the output $y_{2,1} = 2 \cdot b \cdot r_3 + r_2$, which can be re-written as $(1 - (-1)^b) \cdot r_3 + r_2$.

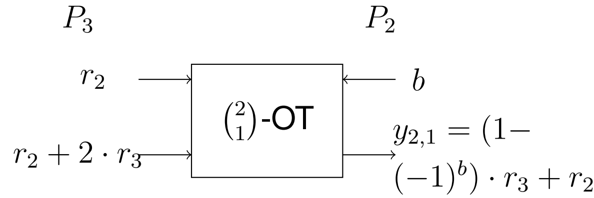

Figure 2.2: OT_2 between P_3 and P_2

OT_3**:** Here P_1 acts as the sender, with input pair $(y_1 - r_1, 2^\alpha - y_1 - r_1) = (2^\alpha \cdot (a \oplus c) - r_3 - r_1, 2^\alpha \cdot [1 \oplus (a \oplus c)] - (r_1 - r_3))$, while P_2 participates as the receiver, with b as its choice bit. It follows from the property of OT that at the end, P_2 obtains the output $y_{2,2} = 2^\alpha \cdot (a \oplus b \oplus c) - r_1 - (-1)^b \cdot r_3$.

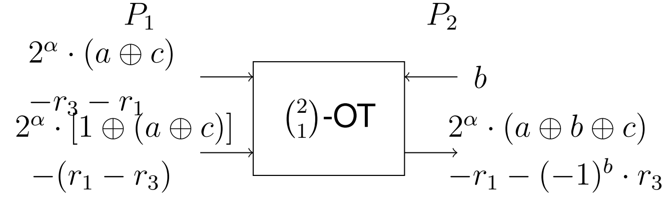

Figure 2.3: OT_3 between P_1 and P_2

At the end, party P_1 sets its share $s_1 = r_1$ and P_3 sets its share $s_3 = r_2 + r_3$, while party P_2 sets its share $s_2 = y_{2,2} - y_{2,1} = 2^\alpha \cdot (a \oplus b \oplus c) - r_1 - r_2 - r_3$. It is easy to verify that $s_1 + s_2 + s_3 = 2^\alpha \cdot (a \oplus b \oplus c)$ holds. The security follows from the properties of OT.

Now to obtain an arithmetic sharing $\langle x \rangle^A$ from $\langle x \rangle^B$, the parties execute ℓ instances $\mathsf{BA}_0, \ldots, \mathsf{BA}_{\ell-1}$ of the protocol BA, where for the instance BA_i with $i \in \{0, \ldots, \ell - 1\}$, the parties set $\alpha = i$ and $a = x_{1,i}, b = x_{2,i}$ and $c = x_{3,i}$. Let $s_{1,i}, s_{2,i}$ and $s_{3,i}$ be the outputs for P_1, P_2 and P_3 respectively during the instance BA_i. Party P_1 then sets $\langle x \rangle_1^A = \sum_{i=0}^{\ell-1} s_{1,i}$, party P_2 sets $\langle x \rangle_2^A = \sum_{i=0}^{\ell-1} s_{2,i}$ and party P_3 sets $\langle x \rangle_3^A = \sum_{i=0}^{\ell-1} s_{3,i}$.

Garbled to Arithmetic Sharing (G2A): Let $\langle x \rangle^G$ be a grabled-sharing of $x \in \{0,1\}^\ell$. As the G2B conversion is for free, we simply convert $\langle x \rangle^G$ to a corresponding arithmetic-sharing as $\langle x \rangle^A$ = B2A(G2B($\langle x \rangle^G$)).

Chapter 3

Implementation Results

In this chapter, we detail the implementation and benchmarking of our BAG protocol. We run our experiments in both local and cloud-based environments. The benchmarking values on cloud execution may better reflect real-world scenarios. We calculated the online and offline execution time and data communication (time and quantity) in both cloud and local settings. Our implementation considers $\mathbb{F} = \text{GF}(2^\ell)$ in the arithmetic world. We find that local computation and communication time is arbitrarily varying depending on the system load, so in this chapter, we mainly focus on the cloud experimental results.

3.1 Experiment setup

For the performance evaluation of our work, we use two deployment scenarios: a local setting (with a low-latency, high-bandwidth network) and an intercontinental cloud setting (with a high-latency network). These two scenarios cover two extremes in the design space w.r.t. latency that affects the performance. In the cloud setting, we put our three parties in three continents.

3.1.1 Hardware Details

Local setting. In the local setting, we run the benchmarks on a desktop machine equipped with a 32GB RAM; an Intel Core i7-7700-4690 octa-core CPU with 3.6 GHz processing speed. The hardware supports AES-NI instructions. We instantiate each party in different terminals for execution.
Cloud setting. In the cloud setting, we run the benchmarks on Microsoft Azure Cloud Services with machines located in West USA, East Asia, and India. We used machines with 1.75GB RAM and single core processor. The bandwidth is limited to 10Mbps. Before running our experiments, we measured the round-trip delays between India–West USA, India–East Asia, and East Asia–West USA. These values average to 0.34 s, 0.14 s and 0.18 s respectively.

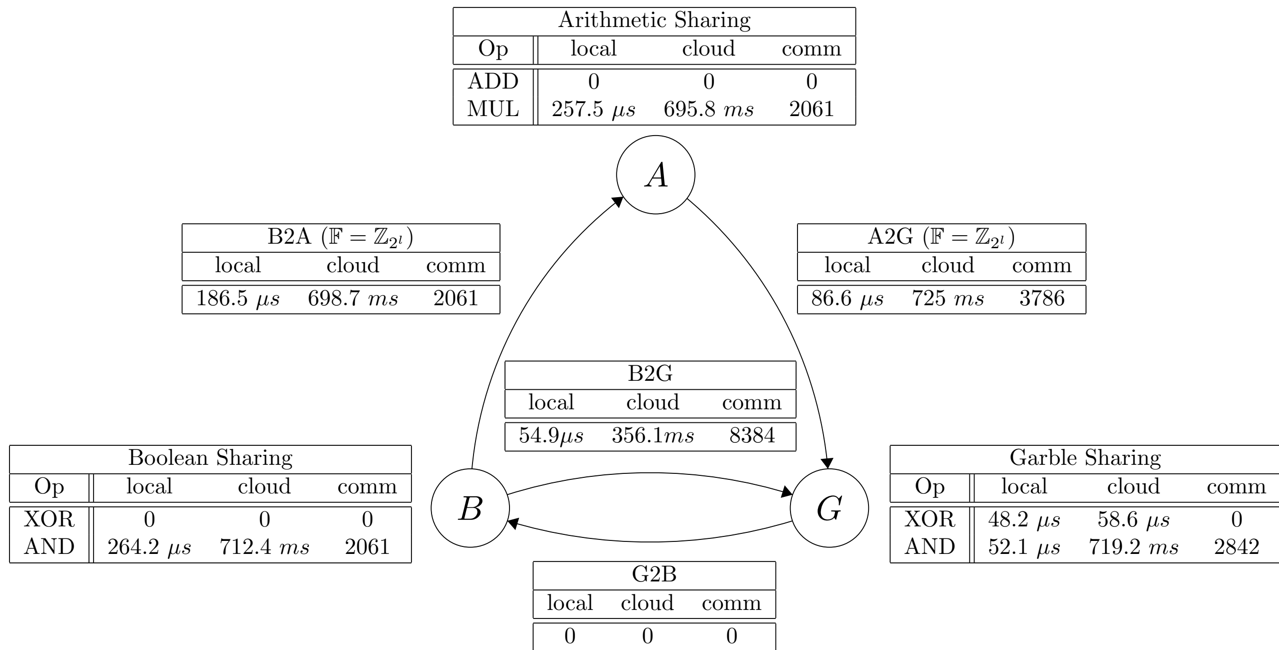

Figure 3.1: Setup time and communication (in Bytes) for a single atomic operation on 64-bit values in a local and cloud scenario, averaged over 1000 operations

3.1.2 Design and Implementation

Similar to ABY, our main design-goal is to achieve an efficient *online phase*, and we build on top of the C++ ABY [8] implementation. The operating system used for our experiments is Ubuntu 18.04. Our code follows the standards of C++11. We used the `openssl` library, which operates with AES-NI support from the Intel hardware.

We simulated the modified BMR execution for our Garbled world. For both the Boolean and the Garbled world, we benchmark XOR and AND operations, and for the arithmetic world, we benchmark addition (`ADD`) and multiplication (`MUL`). We implemented four conversions between the worlds, namely Arithmetic to Garbled (A2G), Boolean to Arithmetic (B2A), Boolean to Garbled (B2G), and Garbled to Boolean (G2B). All the other conversions can be done by combining these four.

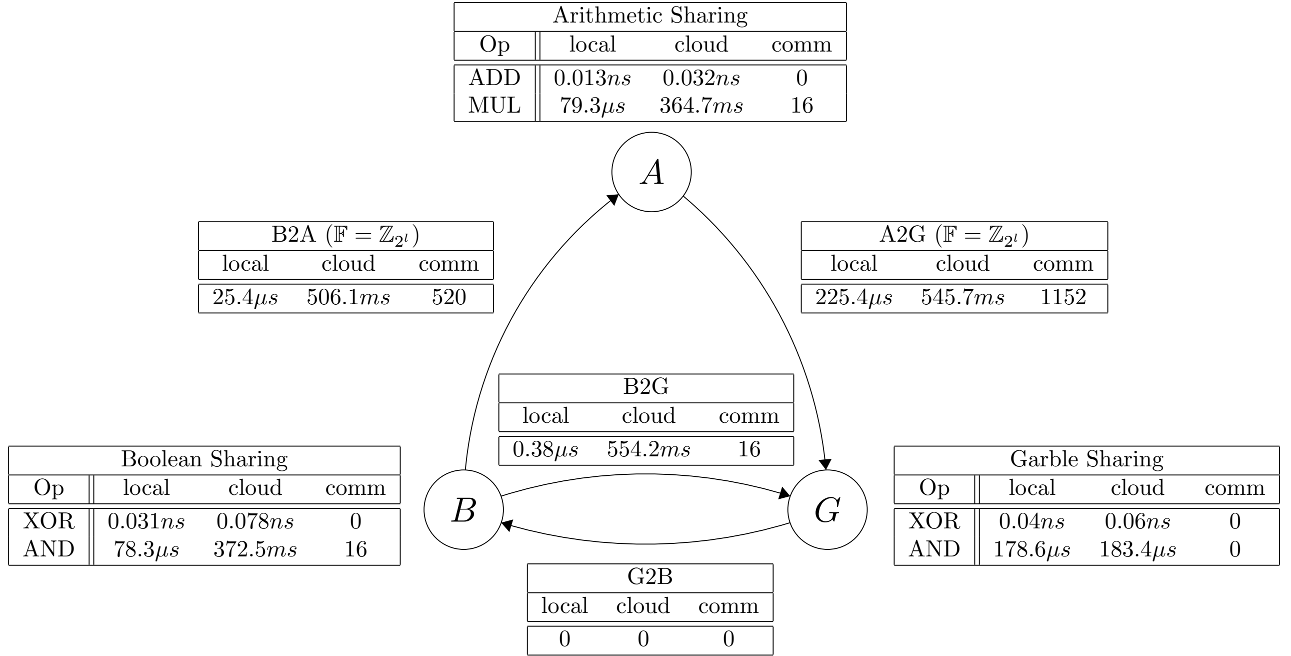

Figure 3.2: Online time and communication (in bytes) for a single atomic operation on 64-bit values in the local and cloud scenarios, averaged over 1000 operations

3.2 Benchmarking

For benchmarking the BAG protocol, we ran each of the elementary operations 1000 times and averaged for both the online and offline phase. Figure 3.1 shows the setup time and communication for a single atomic operation for $l = 64$-bit operands. Figure 3.2 shows the online time and communication for the same. We are mainly interested in the online cloud timing which is representative of the real-world execution time. We find that the local execution time is not stable across multiple runs, so we are not comparing it with ABY. Table 3.1 compares the cloud performance of online execution of our BAG protocol with that of ABY.

Our BAG protocol for three-parties has a reasonably similar performance compared to two-party ABY, with an average overhead of 55% for elementary operations and 20% for conversions; which is really good considering that we have three parties. For the arithmetic world multiplication, we need to communicate 16 bytes because our operands are 64 bits and we have three parties. Similarly for Boolean AND, we need just 16 bytes which is equivalent to ABY. For ADD and XOR, both protocols need only local computation which is negligible.

Our B2A conversion is much more complex because of the three parties; 3 OTs are performed for each bit. The other conversions also have more overhead in the three-party setting compared to two-party ABY. Because our three parties are on different continents, with a delay of up to $0.34s$ on

Operation	Cloud(ms)		Comm($bytes$)	
	ABY	BAG	ABY	BAG
Arith ADD	0	0	0	0
Arith MUL	237.4	364.7	4	16
Bool XOR	0	0	0	0
Bool AND	237.4	372.5	16	16
Garble XOR	0.007	0	0	0
Garble AND	0.016	0.183	0	0
B2A	419.1	506.1	66	520
B2G	478.9	554.2	516	16
A2G	434.6	545.7	1028	1152
G2B	0	0	0	0

Table 3.1: Comparing online cloud performance of important elementary operations with ABY[9]

average, our typical single round will cost $0.34s$ if there is communication between the India and west-USA parties in that round.

We moved most of the expensive operations to the setup phase. We need OTs for both arithmetic and boolean triple generation. We used the exact same trick of ABY to generate the base OTs only once, and then perform OT extension to create required number of OTs. Then we performed multiplication triples generation using these OTs. We can observe from Figure 3.1 that for the cloud, most of the operation setup needs around $700ms$. This is because of the two rounds. One round can cause a delay up to $0.34s$ according to our cloud measurements.

Chapter 4

Conclusion

Mixed protocols for Multi-Party Computation (MPC) are of vital importance in improving the efficiency and performance of practical realizations. In this work, we present the BAG protocol, an extension of the two-party ABY framework to the three party setting. In our protocol, we used state-of-the-art techniques and the best practices in secure computation. For use in our protocol, we constructed faster conversion techniques between the three sharings (namely Boolean, Arithmetic, Garbled) for the three party setting.

We implemented our BAG protocol and evaluated its performance on various elementary operations in real world settings, both local and cloud-based. We also compared the performance of our three-party BAG protocol with the two-party ABY protocol, and found that it is comparable.

We now state some possible directions for future work.

Extending to n parties. The ABY protocol is designed for two parties and our BAG protocol works for three parties. Extending our protocols to work in the n party setting would be a potential future direction. Conceptually, all the three worlds — Boolean, Arithmetic and Garbled — can be easily extended to the n party setting. But the challenge is in finding appropriate conversions that work in the general setting.

Extension to Malicious Adversaries. Another potential direction for future work is to extend our work to tolerate malicious adversaries that can arbitrarily deviate from the protocol, instead of semi-honest adversaries. Secure MPC protocols tolerant of malicious adversaries are more complicated and finding conversions between the three worlds will be the main challenge.

Implementation. In this work we implemented the BAG protocol for benchmarking purposes. Another future direction is to implement the protocol as a reusable library, so that arbitrary protocols can be implemented on top of the BAG protocol.

Bibliography

[1] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513, 1990. 4, 11

[2] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 784–796, 2012. 6

[3] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 578–590, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978347. URL `http://doi.acm.org/10.1145/2976749.2978347`. 4, 11, 12, 13, 14

[4] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 498–507, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315307. URL `http://doi.acm.org/10.1145/1315245.1315307`. 3

[5] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable, efficient, and scalable secure two-party computation for machine learning. Cryptology ePrint Archive, Report 2017/1109, 2017. `https://eprint.iacr.org/2017/1109`. 4

[6] David Chaum, Ivan Damgård, and Jeroen Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 87–119, 1987. 3

[7] S. Choi, K.W. Hwang, Jonathan Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. *Topics in Cryptology–CT-RSA 2012*, pages 416 – 432, 2012/// 2012. doi: 10.1007/978-3-642-27954-6_26. 6, 10

[8] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby framework implementation. GitHub. https://github.com/encryptogroup/ABY. ii, 20

[9] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby - a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015. ii, vi, 1, 3, 4, 5, 16, 22

[10] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985. ISSN 0001-0782. doi: 10.1145/3812.3818. URL http://doi.acm.org/10.1145/3812.3818. 6

[11] Niv Gilboa. Two party rsa key generation. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 116–129, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66347-9. URL http://dl.acm.org/citation.cfm?id=646764.703977. 8, 9

[12] Oded Goldreich, Silvio Micali, and Avi 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, 1987, New York, New York, USA*, pages 218–229, 1987. 3, 9

[13] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 567–578, 2015. 6

[14] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 169–187, Cham, 2016. Springer International Publishing. ISBN 978-3-319-44618-9. 1

[15] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 955–966, 2013. 6

[16] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 486–498, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70583-3. 12

[17] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '00, pages 36–54, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67907-3. URL `http://dl.acm.org/citation.cfm?id=646765.704129`. 1

[18] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 334–348, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.30. URL `http://dx.doi.org/10.1109/SP.2013.30`. 1

[19] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. pages 707–721, 05 2018. 4

[20] Thomas Schneider and Michael Zohner. Gmw vs. yao? efficient secure two-party computation with low depth circuits. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security*, pages 275–292, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39884-1. 6, 10

[21] A. C. Yao. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982. 1, 3

[22] Samee Zahur, Mike Rosulek, and David 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, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 220–250, 2015. 6