

A Pragmatic Introduction to Secure Multi- Party Computation

Other titles in Foundations and Trends® in Privacy and Security

Contextual Integrity through the Lens of Computer Science

Sebastian Benthall, Seda Gurses and Helen Nissenbaum

ISBN: 978-1-68083-384-3

Methods for Location Privacy: A comparative overview

Kostantinos Chatzikokolakis, Ehab ElSalamouny, Catuscia Palamidessi and Pazii Anna

ISBN: 978-1-68083-366-9

Principles and Implementation Techniques of Software-Based Fault Isolation

Gang Tan

ISBN: 978-1-68083-344-7

Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif

Bruno Blanchet

ISBN: 978-1-68083-206-8

A Pragmatic Introduction to Secure Multi-Party Computation

David Evans

University of Virginia
evans@virginia.edu

Vladimir Kolesnikov

Georgia Institute of Technology
kolesnikov@gatech.edu

Mike Rosulek

Oregon State University
rosulekm@eecs.oregonstate.edu

now

the essence of knowledge

Boston — Delft



Documents

Document Sharing Hub

Foundations and Trends® in Privacy and Security

Published, sold and distributed by:

now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:

now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

D. Evans, V. Kolesnikov and M. Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*. Foundations and Trends® in Privacy and Security, vol. 2, no. 2-3, pp. 70–246, 2018.

ISBN: 978-1-68083-509-0

© 2018 D. Evans, V. Kolesnikov and M. Rosulek

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without prior written permission of the publishers.

Photocopying. In the USA: This journal is registered at the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923. Authorization to photocopy items for internal or personal use, or the internal or personal use of specific clients, is granted by now Publishers Inc for users registered with the Copyright Clearance Center (CCC). The 'services' for users can be found on the internet at: www.copyright.com

For those organizations that have been granted a photocopy license, a separate system of payment has been arranged. Authorization does not extend to other kinds of copying, such as that for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. In the rest of the world: Permission to photocopy must be obtained from the copyright owner. Please apply to now Publishers Inc., PO Box 1024, Hanover, MA 02339, USA; Tel. +1 781 871 0245; www.nowpublishers.com; sales@nowpublishers.com

now Publishers Inc. has an exclusive license to publish this material worldwide. Permission to use this content must be obtained from the copyright license holder. Please apply to now Publishers, PO Box 179, 2600 AD Delft, The Netherlands, www.nowpublishers.com; e-mail: sales@nowpublishers.com

Foundations and Trends® in Privacy and Security

Volume 2, Issue 2-3, 2018

Editorial Board

Editors-in-Chief

Anupam Datta

Carnegie Mellon University, USA

Jeannette Wing

Columbia University, USA

Editors

Martín Abadi

Google and University of California, Santa Cruz

Michael Backes

Saarland University

Dan Boneh

Stanford University, USA

Véronique Cortier

LORIA, CNRS, France

Lorrie Cranor

Carnegie Mellon University

Cédric Fournet

Microsoft Research

Virgil Gligor

Carnegie Mellon University

Jean-Pierre Hubaux

EPFL

Deirdre Mulligan

University of California, Berkeley

Andrew Myers

Cornell University

Helen Nissenbaum

New York University

Michael Reiter

University of North Carolina

Shankar Sastry

University of California, Berkeley

Dawn Song

University of California, Berkeley

Daniel Weitzner

Massachusetts Institute of Technology



Documents

Document Sharing Hub

Editorial Scope

Topics

Foundations and Trends® in Privacy and Security publishes survey and tutorial articles in the following topics:

- Access control
- Accountability
- Anonymity
- Application security
- Artificial intelligence methods in security and privacy
- Authentication
- Big data analytics and privacy
- Cloud security
- Cyber-physical systems security and privacy
- Distributed systems security and privacy
- Embedded systems security and privacy
- Forensics
- Hardware security
- Human factors in security and privacy
- Information flow
- Intrusion detection
- Malware
- Metrics
- Mobile security and privacy
- Language-based security and privacy
- Network security
- Privacy-preserving systems
- Protocol security
- Security and privacy policies
- Security architectures
- System security
- Web security and privacy

Information for Librarians

Foundations and Trends® in Privacy and Security, 2018, Volume 2, 4 issues. ISSN paper version 2474-1558. ISSN online version 2474-1566. Also available as a combined paper and online subscription.

Contents

1	Introduction	2
1.1	Outsourced Computation	3
1.2	Multi-Party Computation	4
1.3	MPC Applications	5
1.4	Overview	11
2	Defining Multi-Party Computation	12
2.1	Notations and Conventions	12
2.2	Basic Primitives	14
2.3	Security of Multi-Party Computation	16
2.4	Specific Functionalities of Interest	25
2.5	Further Reading	28
3	Fundamental MPC Protocols	29
3.1	Yao's Garbled Circuits Protocol	30
3.2	Goldreich-Micali-Wigderson (GMW) Protocol	34
3.3	BGW Protocol	39
3.4	MPC From Preprocessed Multiplication Triples	41
3.5	Constant-Round Multi-Party Computation: BMR	43
3.6	Information-Theoretic Garbled Circuits	45

3.7	Oblivious Transfer	51
3.8	Custom Protocols	56
3.9	Further Reading	60
4	Implementation Techniques	62
4.1	Less Expensive Garbling	63
4.2	Optimizing Circuits	71
4.3	Protocol Execution	76
4.4	Programming Tools	80
4.5	Further Reading	83
5	Oblivious Data Structures	84
5.1	Tailored Oblivious Data Structures	85
5.2	RAM-Based MPC	89
5.3	Tree-Based RAM-MPC	90
5.4	Square-Root RAM-MPC	93
5.5	Floram	94
5.6	Further Reading	98
6	Malicious Security	99
6.1	Cut-and-Choose	99
6.2	Input Recovery Technique	104
6.3	Batched Cut-and-Choose	106
6.4	Gate-level Cut-and-Choose: LEGO	107
6.5	Zero-Knowledge Proofs	109
6.6	Authenticated Secret Sharing: BDOZ and SPDZ	113
6.7	Authenticated Garbling	118
6.8	Further Reading	121
7	Alternative Threat Models	123
7.1	Honest Majority	124
7.2	Asymmetric Trust	128
7.3	Covert Security	130
7.4	Publicly Verifiable Covert (PVC) Security	134

7.5 Reducing Communication in Cut-and-Choose Protocols	138
7.6 Trading Off Leakage for Efficiency	139
7.7 Further Reading	142
8 Conclusion	145
Acknowledgements	149
References	151

A Pragmatic Introduction to Secure Multi-Party Computation

David Evans¹, Vladimir Kolesnikov² and Mike Rosulek³

¹*University of Virginia; evans@virginia.edu*

²*Georgia Institute of Technology; kolesnikov@gatech.edu*

³*Oregon State University; rosulekm@eecs.oregonstate.edu*

ABSTRACT

Secure multi-party computation (MPC) has evolved from a theoretical curiosity in the 1980s to a tool for building real systems today. Over the past decade, MPC has been one of the most active research areas in both theoretical and applied cryptography. This book introduces several important MPC protocols, and surveys methods for improving the efficiency of privacy-preserving applications built using MPC. Besides giving a broad overview of the field and the insights of the main constructions, we overview the most currently active areas of MPC research and aim to give readers insights into what problems are practically solvable using MPC today and how different threat models and assumptions impact the practicality of different approaches.

David Evans, Vladimir Kolesnikov and Mike Rosulek (2018), “A Pragmatic Introduction to Secure Multi-Party Computation”, Foundations and Trends® in Privacy and Security: Vol. 2, No. 2-3, pp 70–246. DOI: 10.1561/33000000019.

1

Introduction

Secure multi-party computation (MPC) enable a group to jointly perform a computation without disclosing any participant's private inputs. The participants agree on a function to compute, and then can use an MPC protocol to jointly compute the output of that function on their secret inputs without revealing them. Since its introduction by Andrew Yao in the 1980s, multi-party computation has developed from a theoretical curiosity to an important tool for building large-scale privacy-preserving applications.

This book provides an introduction to multi-party computation for practitioners interested in building privacy-preserving applications and researchers who want to work in the area. We provide an introduction to the foundations of MPC and describe the current state of the art. Our goal is to enable readers to understand what is possible today, and what may be possible in the future, and to provide a starting point for building applications using MPC and for developing MPC protocols, implementations, tools, and applications. As such, we focus on practical aspects, and do not provide formal proofs.

The term *secure computation* is used to broadly encompass all methods for performing computation on data while keeping that data secret. A computation method may also allow participants to confirm the result is indeed the output of the function on the provided inputs, which is known as *verifiable computation*.

There are two main types of secure and verifiable computation: *outsourced computation* and *multi-party computation*. Our focus is on multi-party computation, but first we briefly describe outsourced computation to distinguish it from multi-party computation.

1.1 Outsourced Computation

In an outsourced computation, one party owns the data and wants to be able to obtain the result of computation on that data. The second party receives and stores the data in an encrypted form, performs computation on the encrypted data, and provides the encrypted results to the data owner, without learning anything about the input data, intermediate values, or final result. The data owner can then decrypt the returned results to obtain the output.

Homomorphic encryption allows operations on encrypted data, and is a natural primitive to implement outsourced computation. With *partially-homomorphic encryption* schemes, only certain operations can be performed. Several efficient partially-homomorphic encryption schemes are known (Paillier, 1999; Naccache and Stern, 1998; Boneh *et al.*, 2005). Systems built on them are limited to specialized problems that can be framed in terms of the supported operations.

To provide *fully homomorphic encryption* (FHE), it is necessary to support a Turing-complete set of operations (e.g., both addition and multiplication) so that any function can be computed. Although the goal of FHE was envisioned by Rivest *et al.* (1978), it took more than 30 years before the first FHE scheme was proposed by Gentry (2009), building on lattice-based cryptography. Although there has been much recent interest in implementing FHE schemes Gentry and Halevi (2011), Halevi and Shoup (2014), and Chillotti *et al.* (2016), building secure, deployable, scalable systems using FHE remains an elusive goal.

In their basic forms, FHE and MPC address different aspects of MPC, and as such shouldn't be directly compared. They do, however, provide similar functionalities, and there are ways to adapt FHE to use multiple keys that enables multi-party computation using FHE (Asharov *et al.*, 2012; López-Alt *et al.*, 2012; Mukherjee and Wichs, 2016). FHE offers an asymptotic communication improvement in comparison with MPC, but at the expense of computational efficiency. State-of-the-art FHE implementations (Chillotti *et al.*, 2017) are thousands of times slower than two-party and multi-party

secure computation in typical applications and settings considered in literature. Ultimately, the relative performance of FHE and MPC depends on the relative costs of computation and bandwidth. For high-bandwidth settings, such as where devices connected within a data center, MPC vastly outperforms FHE. As FHE techniques improve, and the relative cost of bandwidth over computation increases, FHE-based techniques may eventually become competitive with MPC for many applications.

We do not specifically consider outsourcing computation or FHE further in this book, but note that some of the techniques developed to improve multi-party computation also apply to FHE and outsourcing. Shan *et al.* (2017) provide a survey of work in the area of outsourcing.

1.2 Multi-Party Computation

The goal of secure multi-party computation (MPC) is to enable a group of independent data owners who do not trust each other or any common third party to jointly compute a function that depends on all of their private inputs. MPC differs from outsourced computation in that all of the protocol participants are data owners who participate in executing a protocol. Chapter 2 provides a more formal definition of MPC, and introduces the most commonly considered threat models.

Brief history of MPC. The idea of secure computation was introduced by Andrew Yao in the early 1980s (Yao, 1982). That paper introduced a general notion of secure computation, in which m parties want to jointly compute a function $f(x_1, x_2, \dots, x_m)$ where x_i is the i^{th} party's private input. In a series of talks over the next few years (but not included in any formal publication), Yao introduced the Garbled Circuits Protocol which we describe in detail in Section 3.1. This protocol remains the basis for many of the most efficient MPC implementations.

Secure computation was primarily of only theoretical interest for the next twenty years; it was not until the 2000s that algorithmic improvements and computing costs had reached a point where it became realistic to think about building practical systems using general-purpose multi-party computation. Fairplay (Malkhi *et al.*, 2004) was the first notable implementation of a general-purpose secure computation system. Fairplay demonstrated the possibility that

a privacy-preserving program could be expressed in a high level language and compiled to executables that could be run by the data-owning participants as a multi-party protocol. However, its scalability and performance limited its use to toy programs — the largest application reported in the Fairplay paper was computing the median two sorted arrays where each party’s input is ten 16-bit numbers in sorted order, involving execution of 4383 gates and taking over 7 seconds to execute (with both parties connected over a LAN). Since then, the speed of MPC protocols has improved by more than five orders of magnitude due to a combination of cryptographic, protocol, network and hardware improvements. This enabled MPC applications to scale to a wide range of interesting and important applications.

Generic and specialized MPC. Yao’s garbled circuits protocol is a *generic* protocol—it can be used to compute any discrete function that can be represented as a fixed-size circuit. One important sub-area of MPC focuses on specific functionalities, such as private set intersection (PSI). For specific functionalities, there may be custom protocols that are much more efficient than the best generic protocols. Specific functionalities can be interesting in their own right, but also can be natural building blocks for use in other applications. We focus mostly on generic MPC protocols, but include discussion of private set intersection (Section 3.8.1) as a particularly useful functionality.

1.3 MPC Applications

MPC enables privacy-preserving applications where multiple mutually distrusting data owners cooperate to compute a function. Here, we highlight a few illustrative examples of privacy-preserving applications that can be built using MPC. This list is far from exhaustive, and is meant merely to give an idea of the range and scale of MPC applications.

Yao’s Millionaires Problem. The toy problem that was used to introduce secure computation is not meant as a useful application. Yao (1982) introduces it simply: “Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other’s wealth.” That is, the goal is to compute the Boolean result of $x_1 \leq x_2$ where x_1 is the first party’s private input and x_2 is the second party’s private input.

Although its a toy problem, Yao's Millionaires Problem can still be useful for illustrating issues in MPC applications.

Secure auctions. The need for privacy in auctions is well understood. Indeed, it is crucial for all participants, both bidders and sellers, to be able to rely on the privacy and non-malleability of bids. *Bid privacy* requires that no player may learn any other player's bid (other than perhaps revealing the winning bid upon the completion of the auction). *Bid non-malleability* means that a player's bid may not be manipulated to generate a related bid. For example, if a party generates a bid of $\$n$, then another party should not be able to use this bid to produce a bid of $\$n + 1$. Note that bid privacy does not necessarily imply bid non-malleability — indeed it is possible to design auction protocols that would hide a bid of $\$n$ while still allowing others to generate a related bid $\$n + 1$.

These properties are crucial in many standard bidding processes. For example, a sealed bid auction is an auction where bidders submit private (sealed) bids in attempts to purchase property, selling to the highest bidder. Clearly, the first bidder's bid value must be kept secret from other potential bidders to prevent those bidders from having an unfair advantage. Similarly, bid malleability may allow a dishonest bidder Bob to present a bid just slightly over Alice's bid, again, gaining an unfair advantage. Finally, the auction itself must be conducted correctly, awarding the item to the highest bidder for the amount of their bid.

A Vickrey auction is a type of sealed-bid auction where instead paying the value of their own bid, the highest bidder wins but the price paid is the value of the second-highest bid. This type of auction gives bidders an incentive to bid their true value, but requires privacy and non-malleability of each bid, and correctness in determining the winner and price.

MPC can be used to easily achieve all these features since it is only necessary to embed the desired properties into the function used to jointly execute the auction. All the participants can verify the function and then rely on the MPC protocol to provide high confidence that the auction will be conducted confidentially and fairly.

Voting. Secure electronic voting, in a simple form, is simply computation of the addition function which tallies the vote. Privacy and non-malleability of

the vote (properties discussed above in the context of auctions) are essential for similar technical reasons. Additionally, because voting is a fundamental civil process, these properties are often asserted by legislation.

As a side note, we remark that voting is an example of an application which may require properties *not covered* by the standard MPC security definitions. In particular, the property of *coercion resistance* is not standard in MPC (but can be formally expressed and achieved (Küsters *et al.*, 2012)). The issue here is the ability of voters to *prove* to a third party how they voted. If such a proof is possible (e.g., a proof might exhibit the randomness used in generating the vote, which the adversary may have seen), then voter coercion is also possible. We don't delve into the specific aspects of secure voting beyond listing it here as a natural application of MPC.

Secure machine learning. MPC can be used to enable privacy in both the inference and training phases of machine learning systems.

Oblivious model inference allows a client to submit a request to a server holding a pre-trained model, keeping the request private from the server S and the model private from the client C . In this setting, the inputs to the MPC are the private model from S , and the private test input from C , and the output (decoded only for C) is the model's prediction. An example of recent work in this setting include MiniONN (Liu *et al.*, 2017), which provided a mechanism for allowing any standard neural network to be converted to an oblivious model service using a combination of MPC and homomorphic encryption techniques.

In the training phase, MPC can be used to enable a group of parties to train a model based on their combined data without exposing that data. For the large scale data sets needed for most machine learning applications, it is not feasible to perform training across private data sets as a generic many-party computation. Instead, hybrid approaches have been designed that combine MPC with homomorphic encryption (Nikolaenko *et al.*, 2013b; Gascón *et al.*, 2017) or develop custom protocols to perform secure arithmetic operations efficiently (Mohassel and Zhang, 2017). These approaches can scale to data sets containing many millions of elements.

Other applications. Many other interesting applications have been proposed for using MPC to enable privacy. A few examples include privacy-preserving

network security monitoring (Burkhart *et al.*, 2010), privacy-preserving genomics (Wang *et al.*, 2015a; Jagadeesh *et al.*, 2017), private stable matching (Doerner *et al.*, 2016), contact discovery (Li *et al.*, 2013; De Cristofaro *et al.*, 2013), ad conversion (Kreuter, 2017), and spam filtering on encrypted email (Gupta *et al.*, 2017).

1.3.1 Deployments

Although MPC has seen much success as a research area and in experimental use, we are still in the early stages of deploying MPC solutions to real problems. Successful deployment of an MPC protocol to solve a problem involving independent and mutually distrusting data owners requires addressing a number of challenging problems beyond the MPC execution itself. Examples of these problems include building confidence in the system that will execute the protocol, understanding what sensitive information might be inferred from the revealed output of the MPC, and enabling decision makers charged with protecting sensitive data but without technical cryptography background to understand the security implications of participating in the MPC.

Despite these challenges, there have been several successful deployments of MPC and a number of companies now focus on providing MPC-based solutions. We emphasize that in this early stage of MPC penetration and awareness, MPC is primarily deployed as an *enabler* of data sharing. In other words, organizations are typically not seeking to use MPC to add a layer of privacy in an otherwise viable application (we believe this is yet forthcoming). Rather, MPC is used to enable a feature or an entire application, which otherwise would not be possible (or would require trust in specialized hardware), due to the value of the shared data, protective privacy legislation, or mistrust of the participants.

Danish sugar beets auction. In what is widely considered to be the first commercial application of MPC, Danish researchers collaborated with the Danish government and stakeholders to create an auction and bidding platform for sugar beet production contracts. As reported in Bogetoft *et al.* (2009), bid privacy and auction security were seen as essential for auction participants. The farmers felt that their bids reflected their capabilities and costs, which they did not want to reveal to Danisco, the only company in Denmark that

processed sugar beets. At the same time, Danisco needed to be involved in the auction as the contracts were securities directly affecting the company.

The auction was implemented as a three-party MPC among representatives for Danisco, the farmer's association (DKS) and the researchers (SIMAP project). As explained by Bogetoft *et al.* (2009), a three party solution was selected, partly because it was natural in the given scenario, but also because it allowed using efficient information theoretic tools such as secret sharing. The project led to the formation of a company, Partisia, that uses MPC to support auctions for industries such as spectrum and energy markets, as well as related applications such as data exchange (Gallagher *et al.*, 2017).

Estonian students study. In Estonia, a country with arguably the most advanced e-government and technology awareness, alarms were raised about graduation rates of IT students. Surprisingly, in 2012, nearly 43% of IT students enrolled in the previous five years had failed to graduate. One potential explanation considered was that the IT industry was hiring too aggressively, luring students away from completing their studies. The Estonian Association of Information and Communication Technology wanted to investigate by mining education and tax records to see if there was a correlation. However, privacy legislation prevented data sharing across the Ministry of Education and the Tax Board. In fact, k -anonymity-based sharing was allowed, but it would have resulted in low-quality analysis, since many students would not have had sufficiently large groups of peers with similar qualities.

MPC provided a solution, facilitated by the Estonian company Cybernetica using their Sharemind framework (Bogdanov *et al.*, 2008a). The data analysis was done as a three-party computation, with servers representing the Estonian Information System's Authority, the Ministry of Finance, and Cybernetica. The study, reported in Cybernetica (2015) and Bogdanov (2015), found that there was no correlation between working during studies and failure to graduate on time, but that more education was correlated with higher income.

Boston wage equity study. An initiative of the City of Boston and the Boston Women's Workforce Council (BWFC) aims to identify salary inequities across various employee gender and ethnic demographics at different levels of employment, from executive to entry-level positions. This initiative is widely

supported by the Boston area organizations, but privacy concerns prevented direct sharing of salary data. In response, Boston University researchers designed and implemented a web-based MPC aggregation tool, which allowed employers to submit the salary data privately and with full technical and legal protection, for the purposes of the study.

As reported by Bestavros *et al.* (2017), MPC enabled the BWWC to conduct their analysis and produce a report presenting their findings. The effort included a series of meetings with stakeholders to convey the risks and benefits of participating in the MPC, and considered the importance of addressing usability and trust concerns. One indirect result of this work is inclusion of secure multi-party computation as a requirement in a bill for student data analysis recently introduced in the United States Senate (Wyden, 2017).

Key management. One of the biggest problems faced by organizations today is safeguarding sensitive data as it is being used. This is best illustrated using the example of authentication keys. This use case lies at the core of the product offering of Unbound Tech (Unbound Tech, 2018). Unlike other uses of MPC where the goal is to protect data owned by multiple parties from exposure, here the goal is to protect from compromise the data owned by a single entity.

To enable a secure login facility, an organization must maintain private keys. Let's consider the example of shared-key authentication, where each user has shared a randomly chosen secret key with the organization. Each time the user U authenticates, the organization's server S looks up the database of keys and retrieves U 's public key sk_U , which is then used to authenticate and admit U to the network by running key exchange.

The security community has long accepted that it is nearly impossible to operate a fully secure complex system, and an adversary will be able to penetrate and stealthily take control over some of the network nodes. Such an advanced adversary, sometimes called Advanced Persistent Threat (APT), aims to quietly undermine the organization. Naturally, the most prized target for APT and other types of attackers is the key server.

MPC can play a significant role in *hardening* the key server by splitting its functionality into two (or more) hosts, say, S_1 and S_2 , and secret-sharing key material among the two servers. Now, an attacker must compromise *both* S_1 and S_2 to gain access to the keys. We can run S_1 and S_2 on two different software stacks to minimize the chance that they will both be vulnerable to

the exploit available to the malware, and operate them using two different sub-organizations to minimize insider threats. Of course, routine execution does need access to the keys to provide authentication service; at the same time, key should never be reconstructed as the reconstructing party will be the target of the APT attack. Instead, the three players, S_1, S_2 , and the authenticating user U , will run the authentication inside MPC, *without ever reconstructing any secrets*, thus removing the singular vulnerability and hardening the defense.

1.4 Overview

Because MPC is a vibrant and active research area, it is possible to cover only a small fraction of the most important work in this book. We mainly discuss generic MPC techniques, focusing mostly on the two-party scenario, and emphasizing a setting where all but one of the parties may be corrupted. In the next chapter, we provide a formal definition of secure multi-party computation and introduce security models that are widely-used in MPC. Although we do not include formal security proofs in this book, it is essential to have clear definitions to understand the specific guarantees that MPC provides. Chapter 3 describes several fundamental MPC protocols, focusing on the most widely-used protocols that resist any number of corruptions. Chapter 4 surveys techniques that have been developed to enable efficient implementations of MPC protocols, and Chapter 5 describes methods that have been used to provide sub-linear memory abstractions for MPC.

Chapters 3–5 target the weak semi-honest adversary model for MPC (defined in Chapter 2), in which it is assumed that all parties follow the protocol as specified. In Chapter 6, we consider how MPC protocols can be hardened to provide security against active adversaries, and Chapter 7 explores some alternative threat models that enable trade-offs between security and efficiency. We conclude in Chapter 8, outlining the trajectory of MPC research and practice, and suggesting possible directions for the future.

2

Defining Multi-Party Computation

In this chapter, we introduce notations and conventions we will use throughout, define some basic cryptographic primitives, and provide a security definition for multi-party computation. Although we will not focus on formal security proofs or complete formal definitions, it is important to have clear security definitions to understand exactly what properties protocols are designed to provide. The protocols we discuss in later chapters have been proven secure based on these definitions.

2.1 Notations and Conventions

We will abbreviate *Secure Multi-Party Computation* as *MPC*, and will use it to denote secure computation among two or more participants. The term *secure function evaluation* (SFE) is often used to mean the same thing, although it can also apply to contexts where only one party provides inputs to a function that is evaluated by an outsourced server. Because two-party MPC is an important special case, which received a lot of targeted attention, and because two-party protocols are often significantly different from the general n -party case, we will use 2PC to emphasize this setting when needed.

We assume existence of direct secure channels between each pairs of participating players. Such channels could be achieved inexpensively through a variety of means, and are out of scope in this book.

We denote encryption and decryption of a message m under key k as $\text{Enc}_k(m)$ and $\text{Dec}_k(m)$. We will refer to protocol participants interchangeably also as parties or players, and will usually denote them as P_1, P_2 , etc. We will denote the adversary by $\mathcal{A}$.

A negligible function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is any function that approaches zero asymptotically faster than any inverse polynomial. In other words, for any polynomial p , $\nu(n) < 1/p(n)$ for all but finitely many n .

We will denote computational and statistical security parameters by κ and σ respectively. The computational security parameter κ governs the hardness of problems that can be broken by an adversary's offline computation — e.g., breaking an encryption scheme. In practice κ is typically set to a value like 128 or 256. Even when we consider security against computationally bounded adversaries, there may be some attacks against an interactive protocol that are not made easier by offline computation. For example, the interactive nature of a protocol may give the adversary only a single opportunity to violate security (e.g., by sending a message that has a special property, like predicting the random value that an honest party will chose in the next round). The statistical security parameter σ governs the hardness of these attacks. In practice, σ is typically set to a smaller value like 40 or 80. The correct way to interpret the presence of two security parameters is that security is violated only with probability $2^{-\sigma} + \nu(\kappa)$, where ν is a negligible function *that depends on the resources of the adversary*. When we consider computationally unbounded adversaries, we omit κ and require $\nu = 0$.

We will use symbol $\in_R$ to denote uniformly random sampling from a distribution. For example we write “choose $k \in_R \{0, 1\}^\kappa$ ” to mean that k is a uniformly chosen κ -bit long string. More generally, we write “ $v \in_R D$ ” to denote sampling according to a probability distribution D . Often the distribution in question is the output of a randomized algorithm. We write “ $v \in_R A(x)$ ” to denote that v is the result of running randomized algorithm A on input x .

Let D_1 and D_2 be two probability distributions indexed by a security parameter, or equivalently two algorithms that each take a security parameter

as input.¹ We say that D_1 and D_2 are *indistinguishable* if, for all algorithms A there exists a negligible function ν such that:

$$\Pr[A(D_1(n)) = 1] - \Pr[A(D_2(n)) = 1] \leq \nu(n)$$

In other words, no algorithm behaves more than negligibly differently when given inputs sampled according to D_1 vs D_2 . When we consider only *non-uniform, polynomial-time* algorithms A , the definition results in *computational indistinguishability*. When we consider *all* algorithms without regard to their computational complexity, we get a definition of *statistical indistinguishability*. In that case, the probability above is bounded by the *statistical distance* (also known as total variation distance) of the two distributions, which is defined as:

$$\Delta(D_1(n), D_2(n)) = \frac{1}{2} \sum_x \left| \Pr[x = D_1(n)] - \Pr[x = D_2(n)] \right|$$

Throughout this work, we use *computational security* to refer to security against adversaries implemented by non-uniform, polynomial-time algorithms. We use *information-theoretic security* (also known as *unconditional* or *statistical security*) to mean security against arbitrary adversaries (even those with unbounded computational resources).

2.2 Basic Primitives

Here, we provide definitions of a few basic primitives we use in our presentation. Several other useful primitives are actually special cases of MPC (i.e., they are defined as MPC of specific functions). These are defined in Section 2.4.

Secret sharing. Secret sharing is another essential primitive, which is at the core of many MPC approaches. Informally, a (t, n) -secret sharing scheme splits the secret s into n shares, such that any $t - 1$ of the shares reveal no information about s , while any t shares allow complete reconstruction of the secret s . There are many variants of possible security properties of secret sharing schemes; we provide one definition, adapted from Beimel and Chor (1993), next.

Definition 2.1. Let D be the domain of secrets and D_1 be the domain of shares. Let $\text{Shr} : D \rightarrow D_1^n$ be a (possibly randomized) sharing algorithm, and

¹In the literature, D_1 and D_2 are often referred to as an *ensemble* of distributions.

$\text{Rec} : D_1^k \rightarrow D$ be a reconstruction algorithm. A (t, n) -secret sharing scheme is a pair of algorithms (Shr, Rec) that satisfies these two properties:

- *Correctness.* Let $(s_1, s_2, \dots, s_n) = \text{Shr}(s)$. Then,

$$\Pr[\forall k \geq t, \text{Rec}(s_{i_1}, \dots, s_{i_k}) = s] = 1.$$

- *Perfect Privacy.* Any set of shares of size less than t does not reveal anything about the secret in the information theoretic sense. More formally, for any two secrets $a, b \in D$ and any possible vector of shares $\mathbf{v} = v_1, v_2, \dots, v_k$, such that $k < t$,

$$\Pr[\mathbf{v} = \text{Shr}(a)|_k] = \Pr[\mathbf{v} = \text{Shr}(b)|_k],$$

where $|_k$ denotes appropriate projection on a subspace of k elements.

In many of our discussions we will use (n, n) -secret sharing schemes, where all n shares are necessary and sufficient to reconstruct the secret.

Random Oracle. Random Oracle (RO) is a heuristic model for the security of hash functions, introduced by Bellare and Rogaway (1993). The idea is to treat the hash function as a public, idealized random function. In the random oracle model, all parties have access to the public function $H : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$, implemented as a stateful oracle. On input string $x \in \{0, 1\}^*$, H looks up its history of calls. If $H(x)$ had never been called, H chooses a random $r_x \in \{0, 1\}^\kappa$, remembers the pair x, r_x and returns r_x . If $H(x)$ had been called before, H returns r_x . In this way, the oracle realizes a randomly-chosen function $\{0, 1\}^* \rightarrow \{0, 1\}^\kappa$.

The random oracle model is a heuristic model, because it captures only those attacks that treat the hash function H as a black-box. It deviates from reality in that it models a public function (e.g., a standardized hash function like SHA-256) as an inherently random object. In fact, it is possible to construct (extremely contrived) schemes that are secure in the random oracle model, but which are insecure whenever H is instantiated by *any* concrete function (Canetti *et al.*, 1998).

Despite these shortcomings, the random oracle model is often considered acceptable for practical applications. Assuming a random oracle often leads to significantly more efficient constructions. In this work we will be careful to state when a technique relies on the random oracle model.

2.3 Security of Multi-Party Computation

Informally, the goal of MPC is for a group of participants to learn the correct output of some agreed-upon function applied to their private inputs without revealing anything else. We now provide a more formal definition to clarify the security properties MPC aims to provide. First, we present the *real-ideal paradigm* which forms the conceptual core of defining security. Then we discuss two different adversary models commonly used for MPC. Finally, we discuss issues of composition—namely, whether security preserved in the natural way when a secure protocol invokes another subprotocol.

2.3.1 Real-Ideal Paradigm

A natural way to define security is to come up with a kind of a “laundry list” of things that constitute a violation of security. For example, the adversary should not be able to learn a certain predicate of another party’s input, the adversary should not be able to induce impossible outputs for the honest parties, and the adversary should not be able to make its inputs depend on honest parties’ inputs. Not only is this a tedious approach, but it is cumbersome and error-prone. It is not obvious when the laundry list could be considered complete.

The real-ideal paradigm avoids this pitfall completely by introducing an incredibly clear “ideal world” that implicitly captures all security guarantees, and defining security in relation to this ideal world. The definition of probabilistic encryption by Goldwasser and Micali (1984) is widely considered to be the first instance of using this approach to define and prove security, although they used different terminology.

Ideal world. In the ideal world, the parties securely compute the function $\mathcal{F}$ by privately sending their inputs to a completely trusted party $\mathcal{T}$, referred to as the *functionality*. Each party P_i has an associated input x_i , which is sent to $\mathcal{T}$, which simply computes $\mathcal{F}(x_1, \dots, x_n)$ and returns the result to all parties. Often we will make a distinction between $\mathcal{F}$ as a trusted party (functionality) and the circuit C that such a party computes on the private inputs.

We can imagine an adversary attempting to attack the ideal-world interaction. An adversary can take control over any of the parties P_i , but not $\mathcal{T}$ (that is the sense in which $\mathcal{T}$ is described as a *trusted* party). The simplicity

of the ideal world makes it easy to understand the effect of such an attack. Considering our previous laundry list: the adversary clearly learns no more than $\mathcal{F}(x_1, \dots, x_n)$ since that is the only message it receives; the outputs given to the honest parties are all consistent and legal; the adversary's choice of inputs is independent of the honest parties'.

Although the ideal world is easy to understand, the presence of a fully-trusted third party makes it imaginary. We use the ideal world as a benchmark against which to judge the security of an actual protocol.

Real world. In the real world, there is no trusted party. Instead, all parties communicate with each other using a protocol. The protocol π specifies for each party P_i a “next-message” function π_i . This function takes as input a security parameter, the party's private input x_i , a random tape, and the list of messages P_i has received so far. Then, π_i outputs either a next message to send along with its destination, or else instructs the party to terminate with some specific output.

In the real world, an adversary can corrupt parties—corruption at the beginning of the protocol is equivalent to the original party being an adversary. Depending on the threat model (discussed next), corrupt parties may either follow the protocol as specified, or deviate arbitrarily in their behavior.

Intuitively speaking, the real world protocol π is considered secure if any effect that an adversary can achieve in the real world can also be achieved by a corresponding adversary in the ideal world. Put differently, the goal of a protocol is to provide security in the real world (given a set of assumptions) that is equivalent to that in the ideal world.

2.3.2 Semi-Honest Security

A *semi-honest* adversary is one who corrupts parties but follows the protocol as specified. In other words, the corrupt parties run the protocol honestly but they may try to learn as much as possible from the messages they receive from other parties. Note that this may involve several colluding corrupt parties pooling their views together in order to learn information. Semi-honest adversaries are also considered *passive* in that they cannot take any actions other than attempting to learn private information by observing a view of a protocol execution. Semi-honest adversaries are also commonly called *honest-but-curious*.

The *view* of a party consists of its private input, its random tape, and the list of all messages received during the protocol. The view of an adversary consists of the combined views of all corrupt parties. Anything an adversary learns from running the protocol must be an efficiently computable function of its view. That is, without loss of generality we need only consider an “attack” in which the adversary simply outputs its entire view.

Following the real-ideal paradigm, security means that such an “attack” can also be carried out in the ideal world. That is, for a protocol to be secure, it must be possible in the ideal world to generate something indistinguishable from the real world adversary’s view. Note that the adversary’s view in the ideal world consists of nothing but inputs sent to $\mathcal{T}$ and outputs received from $\mathcal{T}$. So, an ideal-world adversary must be able to use this information to generate what looks like a real-world view. We refer to such an ideal-world adversary as a *simulator*, since it generates a “simulated” real-world view while in the ideal-world itself. Showing that such a simulator exists proves that there is nothing an adversary can accomplish in the real world that could not also be done in the ideal world.

More formally, let π be a protocol and $\mathcal{F}$ be a functionality. Let C be the set of parties that are corrupted, and let Sim denote a simulator algorithm. We define the following distributions of random variables:

- $\text{Real}_\pi(\kappa, C; x_1, \dots, x_n)$: run the protocol with security parameter κ , where each party P_i runs the protocol honestly using private input x_i . Let V_i denote the final view of party P_i , and let y_i denote the final output of party P_i .
Output $\{V_i \mid i \in C\}, (y_1, \dots, y_n)$.
- $\text{Ideal}_{\mathcal{F}, \text{Sim}}(\kappa, C; x_1, \dots, x_n)$: Compute $(y_1, \dots, y_n) \leftarrow \mathcal{F}(x_1, \dots, x_n)$.
Output $\text{Sim}(C, \{(x_i, y_i) \mid i \in C\}), (y_1, \dots, y_n)$.

A protocol is secure against semi-honest adversaries if the corrupted parties in the real world have views that are indistinguishable from their views in the ideal world:

Definition 2.2. A protocol π *securely realizes* $\mathcal{F}$ *in the presence of semi-honest adversaries* if there exists a simulator Sim such that, for every subset of corrupt parties C and all inputs $x_1, \dots, x_n$, the distributions

$$\text{Real}_\pi(\kappa, C; x_1, \dots, x_n)$$

and

$$\text{Ideal}_{\mathcal{F}, \text{Sim}}(\kappa, C; x_1, \dots, x_n)$$

are indistinguishable (in κ).

In defining Real and Ideal we have included the outputs of all parties, even the honest ones. This is a way of incorporating a correctness condition into the definition. In the case that no parties are corrupt ($C = \emptyset$), the output of Real and Ideal simply consist of all parties' outputs in the two interactions. Hence, the security definition implies that protocol gives outputs which are distributed just as their outputs from the ideal functionality (and this is true even when $\mathcal{F}$ is randomized). Because the distribution of $y_1, \dots, y_n$ in Real does not depend on the set C of corrupted parties (no matter who is corrupted, the parties all run honestly), it is not strictly necessary to include these values in the case of $C \neq \emptyset$, but we choose to include it to have a unified definition.

The semi-honest adversary model may at first glance seem exceedingly weak—simply reading and analyzing received messages barely even seems like an attack at all! It is reasonable to ask why such a restrictive adversary model is worth considering at all. In fact, achieving semi-honest security is far from trivial and, importantly, semi-honest protocols often serve as a basis for protocols in more robust settings with powerful attackers. Additionally, many realistic scenarios do correspond to semi-honest attack behavior. One such example is computing with players who are trusted to act honestly, but cannot fully guarantee that their storage might not be compromised in the future.

2.3.3 Malicious Security

A *malicious* (also known as *active*) may instead cause corrupted parties to deviate arbitrarily from the prescribed protocol in an attempt to violate security. A malicious adversary has all the powers of a semi-honest one in analyzing the protocol execution, but may also take any actions it wants during protocol execution. Note that this subsumes an adversary that can control, manipulate, and arbitrarily inject messages on the network (even through throughout this book we assume direct secure channels between each pair of parties). As before, security in this setting is defined in comparison to the ideal world, but there are two important additions to consider:

Effect on honest outputs. When the corrupt parties deviate from the protocol, there is now the possibility that honest parties' outputs will be affected. For example, imagine an adversary that causes two honest parties to output different things while in the ideal world all parties get identical outputs. This condition is somewhat trivialized in the previous definition — while the definition does compare real-world outputs to ideal-world outputs, these outputs have no dependence on the adversary (set of corrupted parties). Furthermore, we can/should make no guarantees on the final outputs of corrupt parties, only of the honest parties, since a malicious party can output whatever it likes.

Extraction. Honest parties follow the protocol according to a well-defined input, which can be given to $\mathcal{T}$ in the ideal world as well. In contrast, the input of a malicious party is not well-defined in the real world, which leads to the question of what input should be given to $\mathcal{T}$ in the ideal world. Intuitively, in a secure protocol, whatever an adversary can do in the real world should also be achievable in the ideal world by *some suitable choice of inputs* for the corrupt parties. Hence, we leave it to the simulator to choose inputs for the corrupt parties. This aspect of simulation is called *extraction*, since the simulator extracts an effective ideal-world input from the real-world adversary that “explains” the input’s real-world effect. In most constructions, it is sufficient to consider *black-box* simulation, where the simulator is given access only to the oracle implementing the real-world adversary, and not its code.

When $\mathcal{A}$ denotes the adversary program, we write $\text{corrupt}(\mathcal{A})$ to denote the set of parties that are corrupted, and use $\text{corrupt}(\text{Sim})$ for the set of parties that are corrupted by the ideal adversary, Sim . As we did for the semi-honest security definition, we define distributions for the real world and ideal world, and define a secure protocol as one that makes those distributions indistinguishable:

- $\text{Real}_{\pi, \mathcal{A}}(\kappa; \{x_i \mid i \notin \text{corrupt}(\mathcal{A})\})$: run the protocol on security parameter κ , where each honest party P_i (for $i \notin \text{corrupt}(\mathcal{A})$) runs the protocol honestly using given private input x_i , and the messages of corrupt parties are chosen according to $\mathcal{A}$ (thinking of $\mathcal{A}$ as a protocol next-message function for a collection of parties). Let y_i denote the output of each

honest party P_i and let V_i denote the final view of party P_i .

Output $(\{V_i \mid i \in \text{corrupt}(\mathcal{A})\}, \{y_i \mid i \notin \text{corrupt}(\mathcal{A})\})$.

- $\text{Ideal}_{\mathcal{F}, \text{Sim}}(\kappa; \{x_i \mid i \notin \text{corrupt}(\mathcal{A})\})$: Run Sim until it outputs a set of inputs $\{x_i \mid i \in \text{corrupt}(\mathcal{A})\}$. Compute $(y_1, \dots, y_n) \leftarrow \mathcal{F}(x_1, \dots, x_n)$. Then, give $\{y_i \mid i \in \text{corrupt}(\mathcal{A})\}$ to Sim.² Let V^* denote the final output of Sim (a set of simulated views).
Output $(V^*, \{y_i \mid i \notin \text{corrupt}(\text{Sim})\})$.

Definition 2.3. A protocol π *securely realizes* $\mathcal{F}$ in the presence of malicious adversaries if for every real-world adversary $\mathcal{A}$ there exists a simulator Sim with $\text{corrupt}(\mathcal{A}) = \text{corrupt}(\text{Sim})$ such that, for all inputs for honest parties $\{x_i \mid i \notin \text{corrupt}(\mathcal{A})\}$, the distributions

$$\text{Real}_{\pi, \mathcal{A}}(\kappa; \{x_i \mid i \notin \text{corrupt}(\mathcal{A})\})$$

and

$$\text{Ideal}_{\mathcal{F}, \text{Sim}}(\kappa; \{x_i \mid i \notin \text{corrupt}(\text{Sim})\})$$

are indistinguishable (in κ).

Note that the definition quantifies only over the inputs of honest parties $\{x_i \mid i \notin \text{corrupt}(\mathcal{A})\}$. The interaction Real does not consider the corrupt parties to have any inputs, and the inputs of the corrupt parties in Sim is only determined indirectly (by the simulator's choice of what to send to $\mathcal{F}$ on the corrupt parties' behalf). While it would be possible to also define inputs for corrupt parties in the real world, such inputs would merely be “suggestions” since corrupt parties could choose to run the protocol on any other input (or behave in a way that is inconsistent with all inputs).

Reactive functionalities. In the ideal world, the interaction with the functionality consists of just a single round: inputs followed by outputs. It is possible to generalize the behavior of $\mathcal{F}$ so that it interacts with the parties over many rounds of interaction, keeping its own private internal state between rounds. Such functionalities are called *reactive*.

²To be more formal, we can write the simulator Sim as a pair of algorithms $\text{Sim} = (\text{Sim}_1, \text{Sim}_2)$ which capture this two-phase process. Sim_1 (on input κ) outputs $\{x_i \mid i \in \text{corrupt}(\mathcal{A})\}$ and arbitrary internal state Σ . Then Sim_2 takes input Σ and $\{y_i \mid i \in \text{corrupt}(\mathcal{A})\}$, and gives output V^* .

One example of a reactive functionality is as the dealer in a poker game. The functionality must keep track of the state of all cards, taking input commands and giving outputs to all parties in many rounds.

Another example is an extremely common functionality called *commitment*. This functionality accepts a bit b (or more generally, a string) from P_1 and gives output “committed” to P_2 , while internally remembering b . At some later time, if P_1 sends the command “reveal” (or “open”) to the functionality, it gives b to P_2 .

Security with abort. In any message-based two-party protocol, one party will learn the final output before the other. If that party is corrupt and malicious, they may simply refuse to send the last message to the honest party and thereby prevent the honest party from learning the output. However, this behavior is incompatible with our previous description of the ideal world. In the ideal world, if corrupt parties receive output from the functionality then all parties do. This property is called *output fairness* and not all functions can be computed with this property (Cleve, 1986; Gordon *et al.*, 2008; Asharov *et al.*, 2015a).

Typical results in the malicious setting provide a weaker property known as *security with abort*, which requires slightly modifying the ideal functionality as follows. First, the functionality is allowed to know the identities of the corrupt parties. The functionality’s behavior is modified to be slightly reactive: after all parties have provided input, the functionality computes outputs and delivers the outputs to the *corrupt parties only*. Then the functionality awaits either a “deliver” or “abort” command from the corrupted parties. Upon receiving “deliver”, the functionality delivers the outputs to all the honest parties. Upon receiving “abort”, the functionality delivers an abort output ($\perp$) to all the honest parties.

In this modified ideal world, an adversary is allowed to learn the output before the honest parties and to prevent the honest parties from receiving any output. It is important to note, however, that whether an honest party aborts can depend only on the corrupt party’s outputs. In particular, it would violate security if the honest party’s abort probability depended on its own input.

Usually the possibility of blocking outputs to honest parties is not written explicitly in the description of the functionality. Instead, it is generally understood that when discussing security against malicious adversaries, the

adversary has control over output delivery to honest parties and output fairness is not expected.

Adaptive corruption. We have defined both the real and ideal worlds so that the identities of the corrupted parties are fixed throughout the entire interaction. This provides what is known as security against *static corruption*. It is also possible to consider scenarios where an adversary may choose which parties to corrupt during the protocol execution, possibly based on what it learns during the interaction. This behavior is known as *adaptive corruption*.

Security against adaptive corruption can be modeled in the real-ideal paradigm, by allowing the adversary to issue a command of the form “corrupt P_i ”. In the real world, this results in the adversary learning the current view (including private randomness) of P_i and subsequently taking over control of its protocol messages. In the ideal world, the simulator learns only the input and outputs of the party upon corruption, and must use this information to generate simulated views. Of course, the views of parties are correlated (if P_i sends a message to P_j , then that message is included in both parties’ views). The challenge of adaptive security is that the simulator must produce views piece-by-piece. For example, the simulator may be asked to produce a view of P_i when that party is corrupted. Any messages sent by P_j to P_i must be simulated without knowledge of P_j ’s private input. Later, the simulator might be asked to provide a view of P_j (including its private randomness) that “explains” its protocol messages as somehow consistent with whatever private input it had.

In this work we consider only static corruption, following the vast majority of work in this field.

2.3.4 Hybrid Worlds and Composition

In the interest of modularity, it is often helpful to design protocols that make use of other ideal functionalities. For example, we may design a protocol π that securely realizes some functionality $\mathcal{F}$, where the parties of π also interact with another functionality $\mathcal{G}$ in addition to sending messages to each other. Hence, the real world for this protocol includes $\mathcal{G}$, while the ideal world (as usual) includes only $\mathcal{F}$. We call this modified real world the *$\mathcal{G}$ -hybrid world*.

A natural requirement for a security model is *composition*: if π is a $\mathcal{G}$ -hybrid protocol that securely realizes $\mathcal{F}$ (i.e., parties in π send messages and also interact with an ideal $\mathcal{G}$), and ρ is a protocol that securely realizes $\mathcal{G}$, then composing π and ρ in the natural way (replacing every invocation of $\mathcal{G}$ with a suitable invocation of ρ) also results in a secure protocol for $\mathcal{F}$. While we have not defined all of the low-level details of a security model for MPC, it may be surprising that some very natural ways of specifying the details *do not guarantee composability* of secure protocols!

The standard way of achieving guaranteed composition is to use the *universal composability* (UC) framework from Canetti (2001). The UC framework augments the security model that we have sketched here with an additional entity called the *environment*, which is included in both the ideal and real worlds. The purpose of the environment is to capture the “context” in which the protocol executes (e.g., the protocol under consideration is invoked as a small step in some larger calling protocol). The environment chooses inputs for the honest party and receives their outputs. It also may interact arbitrarily with the adversary.

The same environment is included in the real and ideal worlds, and its “goal” is to determine whether it is instantiated in the real or ideal world. Previously we defined security by requiring certain real and ideal views to be indistinguishable. In this setting, we can also absorb any distinguisher of these views into the environment itself. Hence, without loss of generality, the environment’s final output can be just a single bit which can be interpreted as the environment’s “guess” of whether it is instantiated in the real or ideal world.

Next, we define the real and ideal executions, where Z is an environment:

- $\text{Real}_{\pi, \mathcal{A}, Z}(\kappa)$: run an interaction involving adversary $\mathcal{A}$ and environment Z . When Z generates an input for an honest party, the honest party runs protocol π , and gives its output to Z . Finally, Z outputs a single bit, which is taken as the output of $\text{Real}_{\pi, \mathcal{A}, Z}(\kappa)$.
- $\text{Ideal}_{\mathcal{F}, \text{Sim}, Z}(\kappa)$: run an interaction involving adversary (simulator) Sim and environment Z . When Z generates an input for an honest party, the input is passed directly to functionality $\mathcal{F}$ and the corresponding output is given to Z (on behalf of that honest party). The output bit of Z is taken as the output of $\text{Ideal}_{\mathcal{F}, \text{Sim}, Z}(\kappa)$.

Definition 2.4. A protocol π *UC-securely realizes* $\mathcal{F}$ if for all real-world adversaries $\mathcal{A}$ there exists a simulator Sim with $\text{corrupt}(\mathcal{A}) = \text{corrupt}(\text{Sim})$ such that, for all environments Z :

$$\left| \Pr[\text{Real}_{\pi, \mathcal{A}, Z}(\kappa) = 1] - \Pr[\text{Ideal}_{\mathcal{F}, \text{Sim}, Z}(\kappa) = 1] \right| \text{ is negligible (in } \kappa).$$

Since the definition quantifies over all environments, we can always consider absorbing the adversary $\mathcal{A}$ into the environment Z , so that what is left over is the so-called “dummy adversary” (which simply forwards protocol messages as instructed by Z).

In other (non UC-composable) security models, the ideal-world adversary (simulator) can depend arbitrarily on the real-world adversary. In particular, the simulator can do things like internally run the adversary and repeatedly rewind that adversary to a previous internal state. Many protocols are proven in these weaker model where the composability may be restricted. Sequential composition security (i.e., security for protocols which call functionalities in a sequential manner) holds for all protocols discussed in this book.

In the UC model such rewinding is not possible since the adversary can be assumed to be absorbed into the environment, and the simulator is not allowed to depend on the environment. Rather, the simulator must be a *straight-line simulator*: whenever the environment wishes to send a protocol message, the simulator must reply immediately with a simulated response. A straight-line simulator must generate the simulated transcript in one pass, whereas the previous definitions allowed for the simulated transcript or view to be generated without any restrictions. Assuming the other primitives (e.g., OT and commitments) used in these protocols provide UC-security, the malicious secure protocols described in this book are all UC-secure.

2.4 Specific Functionalities of Interest

Here, we define several functionalities that have been identified as particularly useful building blocks for building MPC protocols.

Oblivious Transfer. Oblivious Transfer (OT) is an essential building block for secure computation protocols. It is theoretically equivalent to MPC as shown by Kilian (1988): given OT, one can build MPC without any additional assumptions, and, similarly, one can directly obtain OT from MPC.

PARAMETERS:

1. Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$. $\mathcal{S}$ has input secrets $x_0, x_1 \in \{0, 1\}^n$, and $\mathcal{R}$ has a selection bit $b \in \{0, 1\}$.

FUNCTIONALITY:

- $\mathcal{R}$ receives x_b , $\mathcal{S}$ receives $\perp$.

Figure 2.1: 1-out-of-2 OT functionality $\mathcal{F}^{\text{OT}}$.

PARAMETERS:

1. Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$. Length of committed string n .

FUNCTIONALITY:

- $\mathcal{S}$ sends a string $s \in \{0, 1\}^n$ to $\mathcal{F}^{\text{Comm}}$. $\mathcal{F}^{\text{Comm}}$ sends **committed** to $\mathcal{R}$.
- At some later time, $\mathcal{S}$ sends **open** to $\mathcal{F}^{\text{Comm}}$. $\mathcal{F}^{\text{Comm}}$ sends s to $\mathcal{R}$.

Figure 2.2: Commitment functionality $\mathcal{F}^{\text{Comm}}$.

The standard definition of 1-out-of-2 OT involves two parties, a Sender $\mathcal{S}$ holding two secrets x_0, x_1 and a receiver $\mathcal{R}$ holding a choice bit $b \in \{0, 1\}$. OT is a protocol allowing $\mathcal{R}$ to obtain x_b while learning nothing about the “other” secret x_{1-b} . At the same time, $\mathcal{S}$ does not learn anything at all. More formally:

Definition 2.5. A 1-out-of-2 OT is a cryptographic protocol securely implementing the functionality $\mathcal{F}^{\text{OT}}$ of Figure 2.1.

Later in this chapter we define more formally what it means for a cryptographic protocol to be *secure* in this setting.

Many variants of OT may be considered. A natural variant is 1-out-of- k OT, in which $\mathcal{S}$ holds k secrets, and $\mathcal{R}$ has a choice selector from $[0, \dots, k-1]$. We discuss protocols for implementing OT efficiently in Section 3.7.

PARAMETERS:

1. Two parties: Prover $\mathcal{P}$ and Verifier $\mathcal{V}$.

FUNCTIONALITY:

- $\mathcal{P}$ sends (C, x) to $\mathcal{F}^{\text{zk}}$, where $C : \{0, 1\}^n \rightarrow \{0, 1\}$ is a Boolean circuit with 1 output bit, and $x \in \{0, 1\}^n$. If $C(x) = 1$ then $\mathcal{F}^{\text{zk}}$ sends (**proven**, C) to $\mathcal{V}$. Otherwise, it sends $\perp$ to $\mathcal{V}$.

Figure 2.3: Zero-knowledge proof functionality $\mathcal{F}^{\text{zk}}$.

Commitment. Commitment is a fundamental primitive in many cryptographic protocols. A commitment scheme allows a sender to commit to a secret value, and reveal it at some later time to a receiver. The receiver should learn nothing about the committed value before it is revealed by the sender (a property referred to as *hiding*), while the sender should not be able to change its choice of value after committing (the *binding* property).

Commitment is rather simple and inexpensive in the random oracle model. To commit to x , simply choose a random value $r \in_R \{0, 1\}^k$ and publish the value $y = H(x||r)$. To later reveal, simply announce x and r .

Definition 2.6. *Commitment* is a cryptographic protocol implementing the functionality $\mathcal{F}^{\text{Comm}}$ of Figure 2.2.

Zero-knowledge proof. A zero-knowledge (ZK) proof allows a prover to convince a verifier that it knows x such that $C(x) = 1$, without revealing any further information about x . Here C is a public predicate.

As a simple example, suppose G is a graph and that Alice knows a 3-coloring χ for G . Then Alice can use a ZK proof to convince Bob that G is 3-colorable. She constructs a circuit C_G that interprets its input as an encoding of a 3-coloring and checks whether it is a legal 3-coloring of G . She uses (C_G, χ) as input to the ZK proof. From Bob's point of view, he receives output (**proven**, C_G) if and only if Alice was able to provide a valid 3-coloring of G . At the same time, Alice knows that Bob learned nothing about her 3-coloring χ other than the fact that *some* legal χ exists.

Definition 2.7. A *zero-knowledge proof* is a cryptographic protocol implementing the functionality $\mathcal{F}^{\text{zk}}$ of Figure 2.3.

There are several variants of ZK proofs identified in the literature. Our specific variant is more precisely a zero-knowledge *argument of knowledge*. The distinctions between these variants are not crucial for the level of detail we explore in this book.

2.5 Further Reading

The real-ideal paradigm was first applied in the setting of MPC by Goldwasser *et al.* (1985), for the special case of zero-knowledge. Shortly thereafter the definition was generalized to arbitrary MPC by Goldreich *et al.* (1987). These definitions contain the important features of the real-ideal paradigm, but resulted in a notion of security (against malicious adversaries) that was not preserved under composition. In other words, a protocol could be secure according to these models *when executed in isolation*, but may be totally insecure when two protocol instances are run concurrently.

The definition of security that we have sketched in this book is the Universal Composition (UC) framework of Canetti (2001). Protocols proven secure in the UC framework have the important composition property described in Section 2.3.4, which in particular guarantees security of a protocol instance no matter what other protocols are executing concurrently. While the UC framework is the most popular model with this property, there are other models with similar guarantees (Pfitzmann and Waidner, 2000; Hofheinz and Shoup, 2011). The details of all such security models are extensive and subtle. However, a significantly simpler model is presented by Canetti *et al.* (2015), which is equivalent to the full UC model for the vast majority of cases. Some of the protocols we describe are secure in the random oracle model. Canetti *et al.* (2014) describe how to incorporate random oracles into the UC framework.

Our focus in this book is on the most popular security notions — namely, semi-honest security and malicious security. The literature contains many variations on these security models, and some are a natural fit for real-world applications. We discuss some alternative security models in Chapter 7.

3

Fundamental MPC Protocols

In this chapter we survey several important MPC approaches, covering the main protocols and presenting the intuition behind each approach.

All of the approaches discussed can be viewed as a form of computing under encryption, or, more specifically, as secret-sharing the input data and computing on the shares. For example, an encryption $\text{Enc}_k(m)$ of a message m with a key k can be seen as secret-sharing m , where one share is k and the other is $\text{Enc}_k(m)$. We present several fundamental protocols illustrating a variety of generic approaches to secure computation, as summarized in Figure 3.1. All of the protocols of this section target the semi-honest adversary model (Section 2.3.2). We discuss malicious-secure variants in Chapter 6. All of these

protocol	# parties	# rounds	circuit
Yao's GC (Section 3.1)	2	constant	Boolean
GMW (Section 3.2)	many	circuit depth	Boolean or arithmetic
BGW (Section 3.3)	many	circuit depth	Boolean or arithmetic
BMR (Section 3.5)	many	constant	Boolean
GESS (Section 3.6)	2	constant	Boolean <i>formula</i>

Figure 3.1: Summary of semi-honest MPC protocols discussed in this chapter.

protocols build on oblivious transfer, which we discuss how to implement efficiently in Section 3.7.

3.1 Yao's Garbled Circuits Protocol

Yao's Garbled Circuits protocol (GC) is the most widely known and celebrated MPC technique. It is usually seen as best-performing, and many of the protocols we cover build on Yao's GC. While not having the best known communication complexity, it runs in constant rounds and avoids the costly latency associated with approaches, such as GMW (described in Section 3.2), where the number of communication rounds scales with the circuit depth.

3.1.1 GC Intuition

The main idea behind Yao's GC approach is quite natural. Recall, we wish to evaluate a given function $\mathcal{F}(x, y)$ where party P_1 holds $x \in X$ and P_2 holds $y \in Y$. Here X and Y are the respective domains for the inputs of P_1 and P_2 .

Function as a look-up table. First, let's consider a function $\mathcal{F}$ for which the input domain is small and we can efficiently enumerate all possible input pairs, (x, y) . The function $\mathcal{F}$ can be represented as a look-up table T , consisting of $|X| \cdot |Y|$ rows, $T_{x,y} = \langle \mathcal{F}(x, y) \rangle$. The output of $\mathcal{F}(x, y)$ is obtained simply by retrieving $T_{x,y}$ from the corresponding row.

This gives us an alternative (and much simplified!) view of the task at hand. Evaluating a look-up table can be done as follows. P_1 will encrypt T by assigning a randomly-chosen strong key to *each* possible input x and y . That is, for each $x \in X$ and each $y \in Y$, P_1 will choose $k_x \in_R \{0, 1\}^\kappa$ and $k_y \in_R \{0, 1\}^\kappa$. It will then encrypt T by encrypting each element $T_{x,y}$ of T with *both* keys k_x and k_y , and send the encrypted (and randomly permuted!) table $\langle \text{Enc}_{k_x, k_y}(T_{x,y}) \rangle$ to P_2 .

Now our task is to enable P_2 to decrypt (only) the entry $T_{x,y}$ corresponding to players' inputs. This is done by having P_1 send to P_2 the keys k_x and k_y . P_1 knows its input x , and hence simply sends key k_x to P_2 . The key k_y is sent to P_2 using a 1-out-of- $|Y|$ Oblivious Transfer (Section 2.4). Once P_2 receives k_x and k_y , it can obtain the output $\mathcal{F}(x, y)$ by decrypting $T_{x,y}$ using those keys. Importantly, no other information is obtained by P_2 . This is because P_2 only

has a single pair of keys, which can only be used to open (decrypt) a single table entry. We stress that, in particular, it is important that neither partial key, k_x or k_y , by itself can be used to obtain partial decryptions or even determine whether the partial key was used in the obtaining a specific encryption.¹

Point-and-Permute. A careful reader may wonder how P_2 knows which row of the table T to decrypt, as this information is dependent on the inputs of both parties, and, as such, is sensitive.

The simplest way to address this is to encode some additional information in the encrypted elements of T . For example, P_1 may append a string of σ zeros to each row of T . Decrypting the wrong row with high probability ($p = \frac{1}{2^\sigma}$) will produce an entry which will not end with σ zeros, and hence will be rejected by P_2 .

While the above approach works, it is inefficient for P_2 , who expects to need to decrypt half of the rows of the table T . A much better approach, often called *point-and-permute*,² was introduced by Beaver *et al.* (1990). The idea is to interpret part of the key (namely, the last $\lceil \log |X| \rceil$ bits of the first key and the last $\lceil \log |Y| \rceil$ bits of the second key) as a pointer to the permuted table T , where the encryption will be placed. To avoid collisions in table row allocation, P_1 must ensure that the pointer bits don't collide within the space of keys k_x or within the space of k_y ; this can be done in a number of ways. Finally, strictly speaking, key size must be maintained to achieve the corresponding level of security. As a consequence, rather than viewing key bits as a pointer, players will append the pointer bits to the key and maintain the desired key length.

In the subsequent discussions, we assume that the evaluator knows which row to decrypt. In protocol presentations, we may or may not explicitly include the point-and-permute component, depending on context.

¹Consider a counter-example. Suppose P_2 was able to determine that a key k_x that it received from P_1 was used in encrypting r_x rows. Because some input combinations may be invalid, the encrypted look-up table T may have a unique number of rows relying on k_x , which will reveal x to P_1 , violating the required security guarantees.

²This technique was not given a name by Beaver *et al.* (1990). Rather, this name came to be widely used by the community around 2010, as GC research progressed and need for a name arose. This technique is different from the *permute and point* technique introduced and coined in the information-theoretic garbled circuit construction of Kolesnikov (2005), which we discuss in Section 3.6.

Managing look-up table size. Clearly, the above solution is inefficient as it scales linearly with the domain size of $\mathcal{F}$. At the same time, for small functions, such as those defined by a single Boolean circuit gate, the domain has size 4, so using a look-up table is practical.

The next idea is to represent $\mathcal{F}$ as a Boolean circuit C and evaluate each gate using look-up tables of size 4. As before, P_1 generates keys and encrypts look-up tables, and P_2 applies decryption keys without knowing what each key corresponds to. However, in this setting, we cannot reveal the plaintext output of intermediate gates. This can be hidden by making the gate output also a key whose corresponding value is unknown to the evaluator, P_2 .

For each wire w_i of C , P_1 assigns two keys k_i^0 and k_i^1 , corresponding to the two possible values on the wire. We will refer to these keys as *wire labels*, and to the plaintext wire values simply as *wire values*. During the execution, depending on the inputs to the computation, each wire will be associated with a specific plaintext value and a corresponding wire label, which we will call *active value* and *active label*. We stress that the evaluator can know only the *active label*, but not its corresponding *value*, and not the *inactive label*.

Then, going through C , for each gate G with input wires w_i and w_j , and output wire w_t , P_1 builds the following encrypted look-up table:

$$T_G = \begin{pmatrix} \text{Enc}_{k_i^0, k_j^0}(k_t^{G(0,0)}) \\ \text{Enc}_{k_i^0, k_j^1}(k_t^{G(0,1)}) \\ \text{Enc}_{k_i^1, k_j^0}(k_t^{G(1,0)}) \\ \text{Enc}_{k_i^1, k_j^1}(k_t^{G(1,1)}) \end{pmatrix}$$

For example, if G is an AND gate, the look-up table will be:

$$T_G = \begin{pmatrix} \text{Enc}_{k_i^0, k_j^0}(k_t^0) \\ \text{Enc}_{k_i^0, k_j^1}(k_t^0) \\ \text{Enc}_{k_i^1, k_j^0}(k_t^0) \\ \text{Enc}_{k_i^1, k_j^1}(k_t^1) \end{pmatrix}$$

Each cell of the look-up table encrypts the *label corresponding to the output computed by the gate*. Crucially, this allows the evaluator P_2 to obtain the intermediate active labels on internal circuit wires and use them in the evaluation of $\mathcal{F}$ under encryption without ever learning their semantic value.

P_1 permutes the entries in each of the look-up tables (usually called *garbled tables* or *garbled gates*), and sends all the tables to P_2 . Additionally, P_1 sends (only) the active labels of all wires corresponding to the input values to P_2 . For input wires belonging to P_1 's inputs to $\mathcal{F}$, this is done simply by sending the wire label keys. For wires belonging to P_2 's inputs, this is done via 1-out-of-2 Oblivious Transfer.

Upon receiving the input keys and garbled tables, P_2 proceeds with the evaluation. As discussed above, P_2 must be able to decrypt the correct row of each garbled gate. This is achieved by the point-and-permute technique described above. In our case of a 4-row garbled table, the point-and-permute technique is particularly simple and efficient — one pointer bit is needed for each input, so there are two total pointer bits added to each entry in the garbled table. Ultimately, P_2 completes evaluation of the garbled circuit and obtains the keys corresponding to the output wires of the circuit. These could be sent to P_1 for decryption, thus completing the private evaluation of $\mathcal{F}$.

We note that a round of communication may be saved and sending the output labels by P_2 for decryption by P_1 can be avoided. This can be done simply by P_1 including the decoding tables for the output wires with the garbled circuit it sends. The decoding table is simply a table mapping each label on each *output* wire to its semantics (i.e. the corresponding plaintext value. Now, P_2 obtaining the output labels will look them up in the decoding table and obtain the output in plaintext.

At an intuitive level, at least, it is easy to see that this circuit-based construction is secure in the semi-honest model. Security against a corrupt P_1 is easy, since (other than the OT, which we assume has been separately shown to satisfy the OT security definition) that party receives no messages in the protocol! For a corrupt P_2 , security boils down to the observation that the evaluator P_2 never sees both labels for the same wire. This is obviously true for the input wires, and it holds inductively for all intermediate wires (knowing only one label on each incoming wire of the gate, the evaluator can only decrypt one ciphertext of the garbled gate). Since P_2 does not know the correspondence between plaintext values and the wire labels, it has no information about the plaintext values on the wires, except for the output wires where the association between labels and values is explicitly provided by P_1 . To simulate P_2 's view, the simulator Sim_{P_2} chooses random active labels for each wire, simulates the three “inactive” ciphertexts of each garbled gate as

dummy ciphertexts, and produces decoding information that decodes the active output wires to the function’s output.

3.1.2 Yao’s GC Protocol

Figure 3.2 formalizes Yao’s gate generation, and Figure 3.3 summarizes Yao’s GC protocol. For simplicity of presentation, we describe the protocol variant based on Random Oracle (defined in Section 2.2), even though a weaker assumption (the existence of pseudo-random functions) is sufficient for Yao’s GC construction. The Random Oracle, denoted by H , is used in implementing garbled row encryption. We discuss different methods of instantiating H in practice in Section 4.1.4). The protocol also uses Oblivious Transfer, which requires public-key cryptography.

For each wire label, a pointer bit, p_i , is added to the wire label key following the point-and-permute technique described in Section 3.1.1. The pointer bits leak no information since they are selected randomly, but they allow the evaluator to determine which row in the garbled table to decrypt, based on the pointer bits for the two active wires it has for the inputs. In Section 4.1 we discuss several ways for making Yao’s GC protocol more efficient, including reducing the size of the garbled table to just two ciphertexts per gate (Section 4.1.3) and enabling XOR gates to be computed without encryption (Section 4.1.2).

3.2 Goldreich-Micali-Wigderson (GMW) Protocol

As noted before, computation under encryption can be naturally viewed as operating on secret-shared data. In Yao’s GC, the secret sharing of the active wire value is done by having one player (generator) hold two possible wire labels w_i^0, w_i^1 , and the other player (evaluator) hold the active label w_i^b . In the GMW protocol (Goldreich *et al.*, 1987; Goldreich, 2004), the secret-sharing of the wire value is more direct: the players hold additive shares of the active wire value.

The GMW protocol (or just “GMW”) naturally generalizes to more than two parties, unlike Yao’s GC, which requires novel techniques to generalize to more than two parties (see Section 3.5).

PARAMETERS:

Boolean circuit C implementing function $\mathcal{F}$, security parameter κ .

GC GENERATION:

1. *Wire Label Generation.* For each wire w_i of C , randomly choose wire labels,

$$w_i^b = (k_i^b \in_R \{0, 1\}^\kappa, p_i^b \in_R \{0, 1\}),$$

such that $p_i^b = 1 - p_i^{1-b}$.

2. *Garbled Circuit Construction.* For each gate G_i of C in topological order:

- (a) Assume G_i is a 2-input Boolean gate implementing function $g: w_c = g(w_a, w_b)$, where input labels are $w_a^0 = (k_a^0, p_a^0)$, $w_a^1 = (k_a^1, p_a^1)$, $w_b^0 = (k_b^0, p_b^0)$, $w_b^1 = (k_b^1, p_b^1)$, and the output labels are $w_c^0 = (k_c^0, p_c^0)$, $w_c^1 = (k_c^1, p_c^1)$.

- (b) Create G_i 's garbled table. For each of 2^2 possible combinations of G_i 's input values $v_a, v_b \in \{0, 1\}$, set

$$e_{v_a, v_b} = H(k_a^{v_a} \parallel k_b^{v_b} \parallel i) \oplus w_c^{g_i(v_a, v_b)}$$

Sort entries e in the table by the input pointers, placing entry e_{v_a, v_b} in position $\langle p_a^{v_a}, p_b^{v_b} \rangle$

3. *Output Decoding Table.* For each circuit-output wire w_i (the output of gate G_j) with labels $w_i^0 = (k_i^0, p_i^0)$, $w_i^1 = (k_i^1, p_i^1)$, create garbled output table for both possible wire values $v \in \{0, 1\}$. Set

$$e_v = H(k_i^v \parallel \text{"out"} \parallel j) \oplus v$$

(Because we are xor-ing with a single bit, we just use the lowest bit of the output of H for generating the above e_v .) Sort entries e in the table by the input pointers, placing entry e_v in position p_i^v . (There is no conflict, since $p_i^1 = p_i^0 \oplus 1$.)

Figure 3.2: Yao's Garbled Circuit protocol: GC generation

PARAMETERS: Parties P_1 and P_2 with inputs $x \in \{0,1\}^n$ and $y \in \{0,1\}^n$ respectively. Boolean circuit C implementing function $\mathcal{F}$.

PROTOCOL:

1. P_1 plays the role of GC generator and runs the algorithm of Figure 3.2. P_1 then sends the obtained GC $\widehat{C}$ (including the output decoding table) to P_2 .
2. P_1 sends to P_2 active wire labels for the wires on which P_1 provides input.
3. For each wire w_i on which P_2 provides input, P_1 and P_2 execute an Oblivious Transfer (OT) where P_1 plays the role of the Sender, and P_2 plays the role of the Receiver:
 - (a) P_1 's two input secrets are the two labels for the wire, and P_2 's choice-bit input is its input on that wire.
 - (b) Upon completion of the OT, P_2 receives active wire label on the wire.
4. P_2 evaluates received $\widehat{C}$ gate-by-gate, starting with the active labels on the input wires.
 - (a) For gate G_i with garbled table $T = (e_{0,0}, \dots, e_{1,1})$ and whose active input labels are $w_a = (k_a, p_a)$, $w_b = (k_b, p_b)$, P_2 computes active output label $w_c = (k_c, p_c)$:

$$w_c = H(k_a \parallel k_b \parallel i) \oplus e_{p_a, p_b}$$
5. Obtaining output using output decoding tables. Once all gates of $\widehat{C}$ are evaluated, using "out" for the second key to decode the final output gates, P_2 obtains the final output labels which are equal to the plaintext output of the computation. P_2 sends the obtained output to P_1 , and they both output it.

Figure 3.3: Yao's Garbled Circuit Protocol

3.2.1 GMW Intuition

The GMW protocol can work both on Boolean and arithmetic circuits. We present the two-party Boolean version first, and then briefly explain how the protocol can be generalized to the arithmetic setting, as well as to more than two parties. As with Yao's protocol, we assume players P_1 with input x and P_2 with input y have agreed on the Boolean circuit C representing the computed function $\mathcal{F}(x, y)$.

The GMW protocol proceeds as follows. For each input bit $x_i \in \{0, 1\}$ of $x \in \{0, 1\}^n$, P_1 generates a random bit $r_i \in_R \{0, 1\}$ and sends all r_i to P_2 . Next, P_1 obtains a secret sharing of each x_i among P_1 and P_2 by setting its share to be $x_i \oplus r_i$. Symmetrically, P_2 generates random bit masks for its inputs y_i and sends the masks to P_1 , secret sharing its input similarly.

P_1 and P_2 proceed in evaluating C gate by gate. Consider gate G with input wires w_i and w_j and output wire w_k . Let P_1 holds shares s_i^1 and s_j^1 on w_i and w_j , and P_2 hold shares s_i^2 and s_j^2 on the two wires. Without loss of generality, assume C consists of NOT, XOR and AND gates.

Both NOT and XOR gates can be evaluated without any interaction. A NOT gate is evaluated by P_1 flipping its share of the wire value, which flips the shared wire value. An XOR gate on wires w_i and w_j is evaluated by players xor-ing the shares they already hold. That is, P_1 computes its output share as $s_k^1 = s_i^1 \oplus s_j^1$, and P_2 correspondingly computes its output share as $s_k^2 = s_i^2 \oplus s_j^2$. The computed shares s_k^1, s_k^2 indeed are shares of the active output value: $s_k^1 \oplus s_k^2 = (s_i^1 \oplus s_j^1) \oplus (s_i^2 \oplus s_j^2) = (s_i^1 \oplus s_i^2) \oplus (s_j^1 \oplus s_j^2) = v_1 \oplus v_2$.

Evaluating an AND gate requires interaction and uses 1-out-of-4 OT a basic primitive. From the point of view of P_1 , its shares s_i^1, s_j^1 are fixed, and P_2 has two Boolean input shares, which means there are four possible options for P_2 . If P_1 knew P_2 's shares, then evaluating the gate under encryption would be trivial: P_2 can just reconstruct the active input values, compute the active output value and secret-share it with P_2 . While P_2 cannot do that, it can do the next best thing: prepare such a secret share for *each* of P_2 's possible inputs, and run 1-out-of-4 OT to transfer the corresponding share. Specifically, let

$$S = S_{s_i^1, s_j^1}(s_i^2, s_j^2) = (s_i^1 \oplus s_i^2) \wedge (s_j^1 \oplus s_j^2)$$

be the function computing the gate output value from the shared secrets on the two input wires. P_1 chooses a random mask bit $r \in_R \{0, 1\}$ and prepares a

table of OT secrets:

$$T_G = \begin{pmatrix} r \oplus S(0,0) \\ r \oplus S(0,1) \\ r \oplus S(1,0) \\ r \oplus S(1,1) \end{pmatrix}$$

Then P_1 and P_2 run an 1-out-of-4 OT protocol, where P_1 plays the role of the sender, and P_2 plays the role of the receiver. P_1 uses table rows as each of the four input secrets, and P_2 uses its two bit shares as the selection to choose the corresponding row. P_1 keeps r as its share of the gate output wire value, and P_2 uses the value it receives from the OT execution.

Because of the way the OT inputs are constructed, the players obtain a secret sharing of the gate output wire. At the same time, it is intuitively clear that the players haven't learned anything about the other player's inputs or the intermediate values of the computation. This is because effectively only P_2 receives messages, and by the OT guarantee, it learns nothing about the three OT secrets it did not select. The only thing it learns is its OT output, which is its share of a random sharing of the output value and therefore leaks no information about the plaintext value on that wire. Likewise, P_1 learns nothing about the selection of P_2 .

After evaluating all gates, players reveal to each other the shares of the output wires to obtain the output of the computation.

Generalization to more than two parties. We now sketch how to generalize this to the setting where n players $P_1, P_2, \dots, P_n$ evaluate a boolean circuit $\mathcal{F}$. As before, player P_j secret-shares its input by choosing $\forall i \neq j, r_i \in_R \{0, 1\}$, and sending r_i to each P_i . The parties $P_1, P_2, \dots, P_n$ proceed by evaluating C gate-by-gate. They evaluate each gate G as follows:

- For an XOR gate, the players locally add their shares. Like the two-party case, no interaction is required and correctness and security are assured.
- For an AND gate $c = a \wedge b$, let $a_1, \dots, a_n, b_1, \dots, b_n$ denote the shares of a, b respectively held by the players. Consider the identity

$$\begin{aligned} c = a \wedge b &= (a_1 \oplus \dots \oplus a_n) \wedge (b_1 \oplus \dots \oplus b_n) \\ &= \left(\bigoplus_{i=1}^n a_i \wedge b_i \right) \oplus \left(\bigoplus_{i \neq j} a_i \wedge b_j \right) \end{aligned}$$

Each player P_j computes $a_j \wedge b_j$ locally to obtain a sharing of $\bigoplus_{i=1}^n a_i \wedge b_i$. Further, each pair of players P_i, P_j jointly computes the shares of $a_i \wedge b_j$ as described above in the two-party GMW. Finally, each player outputs the XOR of all obtained shares as the sharing of the result $a \wedge b$.

3.3 BGW Protocol

One of the first multi-party protocols for secure computation is due to Ben-Or, Goldwasser, and Wigderson (Ben-Or *et al.*, 1988), and is known as the “BGW” protocol. Another somewhat similar protocol of Chaum, Crépeau, and Damgård was published concurrently (Chaum *et al.*, 1988) with BGW, and the two protocols are often considered together. For concreteness, we present here the BGW protocol for n parties, which is somewhat simpler.

The BGW protocol can be used to evaluate an arithmetic circuit over a field $\mathbb{F}$, consisting of addition, multiplication, and multiplication-by-constant gates. The protocol is heavily based on Shamir secret sharing (Shamir, 1979), and it uses the fact that Shamir secret shares are homomorphic in a special way—the underlying shared value can be manipulated obliviously, by suitable manipulations to the individual shares.

For $v \in \mathbb{F}$ we write $[v]$ to denote that the parties hold Shamir secret shares of a value v . More specifically, a dealer chooses a random polynomial p of degree at most t , such that $p(0) = v$. Each party P_i then holds value $p(i)$ as their share. We refer to t as the *threshold* of the sharing, so that any collection of t shares reveals no information about v .

The invariant of the BGW protocol is that for every wire w in the arithmetic circuit, the parties hold a secret-sharing $[v_w]$ of the value v_w on that wire. Next, we sketch the protocol with a focus on maintaining this invariant.

Input wires. For an input wire belonging to party P_i , that party knows the value v on that wire in the clear, and distributes shares of $[v]$ to all the parties.

Addition gate. Consider an addition gate, with input wires α, β and output wire γ . The parties collectively hold sharings of incoming wires $[v_\alpha]$ and $[v_\beta]$, and the goal is to obtain a sharing of $[v_\alpha + v_\beta]$. Suppose the incoming sharings correspond to polynomials p_α and p_β , respectively. If each party P_i locally adds their shares $p_\alpha(i) + p_\beta(i)$, then the result is that each party holds a point

on the polynomial $p_\gamma(x) \stackrel{\text{def}}{=} p_\alpha(x) + p_\beta(x)$. Since p_γ also has degree at most t , these new values comprise a valid sharing $p_\gamma(0) = p_\alpha(0) + p_\beta(0) = v_\alpha + v_\beta$.

Note that addition gates require no communication among the parties. All steps are local computation. The same idea works to multiply a secret-shared value by a public constant — each party simply locally multiplies their share by the constant.

Multiplication gate. Consider a multiplication gate, with input wires α, β and output wire γ . The parties collectively hold sharings of incoming wires $[v_\alpha]$ and $[v_\beta]$, and the goal is to obtain a sharing of the product $[v_\alpha \cdot v_\beta]$. As above, the parties can locally multiply their individual shares, resulting in each party holding a point on the polynomial $q(x) = p_\alpha(x) \cdot p_\beta(x)$. However, in this case the resulting polynomial may have degree as high as $2t$ which is too high.

In order to fix the excessive degree of this secret sharing, the parties engage in a degree-reduction step. Each party P_i holds a value $q(i)$, where q is a polynomial of degree at most $2t$. The goal is to obtain a valid secret-sharing of $q(0)$, but with correct threshold.

The main observation is that $q(0)$ can be written as a linear function of the party's shares. In particular,

$$q(0) = \sum_{i=1}^{2t+1} \lambda_i q(i)$$

where the λ_i terms are the appropriate Lagrange coefficients. Hence the degree-reduction step works as follows:

1. Each party³ P_i generates and distributes a threshold- t sharing of $[q(i)]$. To simplify the notation, we do not give names the polynomials that underly these shares. However, it is important to keep in mind that each party P_i chooses a polynomial of degree at most t whose constant coefficient is $q(i)$.
2. The parties compute $[q(0)] = \sum_{i=1}^{2t+1} \lambda_i [q(i)]$, using local computations. Note that the expression is in terms of addition and multiplication-by-constant applied to secret-shared values.

³Technically, only $2t + 1$ parties need to do this.

Since the values $[q(i)]$ were shared with threshold t , the final sharing of $[q(0)]$ also has threshold t , as desired.

Note that multiplication gates in the BGW protocol require communication/interaction, in the form of parties sending shares of $[q(i)]$. Note also that we require $2t + 1 \leq n$, since otherwise the n parties do not collectively have enough information to determine the value $q(0)$, as q may have degree $2t$. For that reason, the BGW protocol is secure against t corrupt parties, for $2t < n$ (i.e., an honest majority).

Output wires. For an output wire α , the parties will eventually hold shares of the value $[v_\alpha]$ on that wire. Each party can simply broadcast its share of this value, so that all parties can learn v_α .

3.4 MPC From Preprocessed Multiplication Triples

A convenient paradigm for constructing MPC protocols is to split the problem into a *pre-processing* phase (before the parties' inputs are known) and an *online* phase (after the inputs are chosen). The pre-processing phase can produce correlated values for the parties, which they can later “consume” in the online phase. This paradigm is also used in some of the leading malicious-secure MPC protocols discussed in Section 6.

Intuition. To get an idea of how to defer some of the protocol effort to the pre-processing phase, recall the BGW protocol. The only real cost in the protocol is the communication that is required for every multiplication gate. However, it is not obvious how to move any of the related costs to a pre-processing phase, since the costs are due to manipulations of secret values that can only be determined in the online phase (i.e., they are based on the circuit inputs). Nonetheless, Beaver (1992) showed a clever way to move the majority of the communication to the pre-processing phase.

A *Beaver triple* (or *multiplication triple*) refers to a triple of secret-shared values $[a], [b], [c]$ where a and b are randomly chosen from the appropriate field, and $c = ab$. In an offline phase, such Beaver triples can be generated in a variety of ways, such as by simply running the BGW multiplication subprotocol on random inputs. One Beaver triple is then “consumed” for each multiplication gate in the eventual protocol.