Author:
Wood, Tim N

Title:
Reducing Communication Costs in Multi-Party Computation

Reducing Communication Costs in Multi-Party Computation

Timothy Nicholas Wood

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering

Department of Computer Science
October 2019

Word count: 102481

Abstract

Multi-party computation (MPC) is a way of computing on private data without revealing the data itself. To do this, data is "secret-shared" amongst a set of mutually-distrusting parties, which perform local computations and interactive processes to evaluate a function. MPC protocols are diverse and offer a variety of different security guarantees, and employ different methodologies to optimize for different aspects of the evaluation. The goal of this work is to show how to reduce communication costs involved in different areas of MPC.

An *access structure* defines which sets of parties a so-called adversary can corrupt such that the inputs remain private and the outputs are revealed to the right parties; one type of access structure is called $\mathscr{Q}_2$, under which assumption there are many classical results on what sorts of functions can be evaluated. One of the contributions of this work is to show how special "error-detection" properties that apply in the context of $\mathscr{Q}_2$ access structures can be used to define MPC protocols that improve on the efficiency of classical results.

In the so-called *preprocessing model*, parties evaluate a function on random inputs, and later "derandomize" using the real (private) inputs. This model is popular in MPC as function evaluation is expensive, whereas derandomization is cheap. In this thesis, a generic method is given for outsourcing preprocessing to a set of servers, which means that low-powered clients who wish to evaluate functions on their data can do so.

The final contribution is a protocol that allows parties to mix two different MPC techniques called *garbled circuits* and *secret-sharing* in a secure way, even when the adversary deviates arbitrarily from the protocol description. Mixing these methods is useful as they optimize for different aspects of computation.

Dedication and Acknowledgements

This work is dedicated to the following people, to whom I am sincerely thankful and without whom this work could not have been completed:

To my secondary school teacher, Mr Mohan, for inspiring me to pursue academic study in mathematics;

To my friends and colleagues in the Bristol Crypto Group and COSIC, for keeping me grounded;

Particularly to Nigel Smart, my advisor, whose enthusiasm was a constant encouragement. He always took the time to explain to me things I didn't understand and helped me to grow a lot as a researcher;

To my family and to Ron and Vera Smith, for their love and care over many years;

And to Him to whom all gratitude belongs.

I would also like to thank DARPA and the FWO who funded me during my studies.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .....

DATE: .....

Contents

List of Figures

List of Tables

Acronyms

ABY Arithmetic-Boolean-Yao

AES Advanced Encryption Standard

API Application Programming Interface

CNF Conjunctive Normal Form

CRS Common Reference String

DNF Disjunctive Normal Form

GC Garbled Circuit

GDPR General Data Protection Regulation

IOT Internet Of Things

IT Information-theoretic

ITM Interactive Turing Machine

LAN Local-area Network

LSSS Linear Secret-sharing Scheme

MAC Message Authentication Code

MPC Multi-party Computation

MSP Monotone Span Program

OT Oblivious Transfer

PKC Public-key Cryptography

PPT Probabilistic Polynomial-time

PRF Pseudorandom Function

PRSS Pseudorandom Secret-sharing

PRZS Pseudorandom Zero-sharing

ROM Random Oracle Model

SHE Somewhat-homomorphic Encryption

SVM Support Vector Machine

UC Universal Composability

VSS Verifiable Secret-sharing

WAN Wide-area Network

ZKP Zero-knowledge Proof

Chapter 1

Introduction

Cryptography is becoming more and more important in our everyday lives, as the amount of data we produce by living in a digital world perpetually increases. Indeed, many companies go to great lengths to seek out and exploit personal data to gain an advantage in an increasingly competitive global market. Moreover, breaches of personal data are becoming ever more frequent.

In the European Union, the General Data Protection Regulation (GDPR) mandates that companies should disclose data breaches, and provides a legal framework for the secure storage and use of personal data. This means that fortunately companies are now obliged to provide a significant measure of control of personal data to the users of their services.

Even if users are given access to privacy controls, it does not mean they will make good use of them: with smart home devices (the Internet of Things (IOT)) it has got to the point where the general population is not only happy to have companies listen to their every conversation, but are willing to pay them for the privilege.

Cryptography is one tool, and is indeed the main tool used today, that can be used to redress the balance back towards end users: while it seems the wont of the commercial world to strip consumers of as much privacy as possible to gain competitive market advantage, the job of cryptographers, one might say, is to put it back again.

Historically, the study of cryptography has chiefly involved showing how to keep data safe while in transit or at rest. By contrast, this thesis concerns methods of keeping data secure during computation, and finding efficient ways of doing so. This, perhaps counterintuitive, idea has its roots in the late 1980s when the first results were published. Since that time significant work has gone into making the protocols much more efficient.

The specific tool under consideration in this work is known as multi-party computation (MPC), which is a method of computing on private data held by different entities so that the only information learnt by each party at the end is the result of the computation and whatever can be inferred from the output and each party's own private input. By construction, the protocol always computes the correct function when all parties are honest. Additionally, the protocol must provide security guarantees that refer to the level of tolerance of misbehaviour by corrupt parties the protocol can withstand while ensuring correctness of the computation and the secrecy of inputs. Certain theoretical impossibility results preclude the existence of protocols for guarantees that are "too strong", i.e. when the corrupt parties are assumed to have too much power.

MPC has many applications and is particularly useful in situations in which privacy is paramount. A popular example is that of computing on private medical data, in which multiple hospitals or data centres hold sensitive data on patients, and using MPC, researchers can analyse the data and perform aggregate statistical analysis without directly learning the private information of individual data subjects. On the commercial side, it has been shown how to use MPC: for private contact discovery, allowing two users of a messaging service to discover they are both users of the service without revealing this metadata to the service provider [DMP11]; in machine learning, to allow a model trained on a private dataset to be queried on a client's private input [MRSV19]; in online advertising, to correlate purchases with whether or not an advert has been shown, to determine its success rate [PSSZ15]; and in private auctions, of which the quintessential example is the Danish Sugar Beet auction [BCD+09], and without mention of which no thesis on MPC would be complete.

1.1 This work

In this thesis, various aspects of secret-sharing-based MPC are explored and improved on. The focus is on evaluating arithmetic circuits on secret inputs in a finite field, i.e. addition and multiplication, but many of the protocols can be executed over $\mathbb{F}_2$, the Galois field of two elements. Computation over a large prime field is often used to emulate arithmetic over $\mathbb{Z}$; one of the major downsides of protocols designed to allow for computation of such circuits is that non-linear operations – such as comparisons of secret-shared data – can be expensive in terms of communication; one of the goals of this thesis to to reduce this cost. The other main goal is to provide efficient protocols for so-called $\mathscr{Q}_2$ access structures by taking information-theoretic protocols and using

computational assumptions to improve on them.

In Chapter 3, proofs of folklore results that apply to linear secret-sharing schemes realizing $\mathscr{Q}_2$ access structures are given. These theoretical results are used to define a practically-efficient actively-secure opening procedure, which is put to good use in later chapters.

The preprocessing model involves splitting computation into an expensive *preprocessing phase* and a cheap *online phase*. Before discussing specific MPC protocols, Chapter 4 provides an overview of how preprocessing is used in MPC in different ways, and gives general constructions that are used throughout the remainder of the thesis.

Chapter 5 gives a protocol to "transfer" secret-shared preprocessed data from a set of servers to a set of clients very cheaply, with the idea that low-powered clients can use the preprocessed data to evaluate a circuit by executing the online phase, without having to execute the preprocessing phase themselves.

Using the results from Chapter 3, in Chapter 6 a computationally-secure protocol that uses replicated secret-sharing to evaluate a circuit is given. Noting the inefficiency of replicated secret-sharing for a large number of parties, in Chapter 7 the results are generalized to an arbitrary secret-sharing scheme, which scales much better with the number of parties.

Finally, in Chapter 8, a "mixed" protocol is given, that allows switching between *garbled circuits* and *secret-sharing*, with active security, in the general multi-party setting. Other works focus on small numbers of parties and involve asymmetric procedures that do not appear to be amenable to generalization to large numbers of parties in the active security setting. The experimental results show there is a tradeoff between preprocessing computation and online computation.

1.2 Changes to Submissions and New Contributions

The work of this thesis is taken mostly from the following publications:

- [SSW17] When It's All Just Too Much: Outsourcing MPC-Preprocessing, published at **IMACC 2019**, joint work with Peter Scholl and Nigel Smart.
- [KRSW18] Reducing Communication Channels in MPC, published at **SCN 2018**, joint work with Marcel Keller, Dragos Rotaru, and Nigel Smart.

- [SW19] Error Detection in Monotone Span Programs with Application to Communication-Efficient Multi-party Computation, published at **CT-RSA 2019**, joint work with Nigel Smart.
- [RW19] MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security, in submission, joint work with Dragoș Rotaru.

Descriptions in the preliminaries borrow from these works. Many of the definitions of functionalities and protocols (and consequently proofs) are standard and immutable. All of the proofs have been reformulated and tidied.

Chapter 3 All of the results of this chapter are taken from [KRSW18, SW19], and have been revised into a general form to unify the works. The proofs in these papers were completed by me, as were the main results on error-detection for $\mathscr{Q}_2$ access structures.

Chapter 4 Section 4.3 is the protocol from [SW19]; however, Section 4.2 that models this protocol independently as a functionality, and the proof, are new. Section 4.5, including $\mathcal{F}_{\mathsf{ABB}}$ and Π_{Online}, are standard definitions from the literature, and the proof is new but is a standard argument. Section 4.6 that defined $\mathcal{F}_{\mathsf{Prep}}$ is based on a standard definition of an MPC functionality, for example as described in [LPSY15]. Section 4.7 is new as it explains how to model SPDZ using the newly-defined functionality $\mathcal{F}_{\mathsf{Open}}$.

Chapter 5 Section 5.3 is new. Section 5.4 is unchanged from the publication, although the proofs (which were completed by me for the publication) have been revised and unified with the exposition for the $\mathscr{Q}_2$ case.

Chapter 6 Most of the results of this chapter are taken from [KRSW18], for which the main MPC protocol formulation was a joint effort but the proofs were completed by me. The main difference in this thesis is that a different formulation of functionalities for generating random secrets in Section 6.3 is presented, in order to make the constructions throughout the thesis more modular. These new formulations, however, are very well-known and standard results.

Chapter 7 This chapter uses the fact that Π_{Online} can be executed for any access structure, which was the idea in [SW19]. Section 7.3 is new. Section 7.5 is a revision of the costs given in [SW19] according to the new results from Section 7.3.

Chapter 8 All of the results of this chapter are taken from [RW19], revised to use the functionality $\mathcal{F}_{\mathsf{Prep}}$, which replaces a functionality $\mathcal{F}_{\mathsf{MPC}}$ from the original work. The idea of using cut-and-choose type protocols was a joint effort with my coauthor; the formulation of the protocols and proofs were my contribution. The implementation, discussed briefly in Section 8.6, was completed by the coauthor of this work.

Chapter 2

Preliminaries

This chapter gives the standard definitions, notation, and cryptographic primitives that are used repeatedly throughout this thesis. A large section is devoted to describing the universal composability (UC) model of Canetti [Can00], since all of the protocols in this thesis are proved secure in this model.

2.1 General

2.1.1 Notation

The set of integers is denoted by $\mathbb{Z}$, the set of natural numbers (excluding 0) by $\mathbb{N}$, and the set of real numbers by $\mathbb{R}$. A set is called *countable* if it is in bijection with a subset of $\mathbb{N}$. The symbol $:=$ is used to denote assignment, so that $x := a$ means that the variable x is assigned the value a. Intervals are defined in the following ways: $[a,b] := \{x \in \mathbb{R} : a \leq x \leq b\}$; $(a,b] := \{x \in \mathbb{R} : a < x \leq b\}$; $[a,b) := \{x \in \mathbb{R} : a \leq x < b\}$; and $(a,b) := \{x \in \mathbb{R} : a < x < b\}$. The set $[1,n] \cap \mathbb{Z} = \{i \in \mathbb{Z} : 1 \leq i \leq n\}$ is written as $[n]$. For a real number $a \in \mathbb{R}$, the value $\lceil a \rceil$ is defined as $\min\{b \in \mathbb{Z} : b \geq a\}$, and $\lfloor a \rfloor$ is the value $\max\{b \in \mathbb{Z} : b \leq a\}$. The function log will always be used to refer to the base-2 logarithm. For a set S its cardinality will be denoted by $|S|$, and its powerset by 2^S. The notation S^* is used to describe the set of all words of finite length comprised of symbols in S. An element str of $\{0,1\}^*$ is called a *binary string*, or simply *string*, and its length is denoted by $|str|$. The notation $a \in A$ is used to denote set membership, and the notation $A \ni a$ is used to indicate that A is the variable under consideration and that only those sets A containing the element a are to be taken: thus $\{A \in \{A_k\}_{k\in[t]} : A \ni a\}$ is the set of all sets in $\{A_k\}_{k\in[t]}$ that contain a.

For any ring R, the value $\mathrm{char}(R)$, called the ring's *characteristic*, is defined as the

smallest positive integer $c \in \mathbb{Z}$ such that $c \cdot x = 0$ for all $x \in R$. The finite field of q elements will be denoted by $\mathbb{F}_q$; it is well known that q must be a prime power. A field is called *Boolean* or *binary* if it is a (possibly trivial) finite field extension of $\mathbb{F}_2$. When the specific field is unimportant it will be written as $\mathbb{F}$.

Matrices will be written as uppercase letters and vectors in bold[1]. The space of all matrices of dimension $m \times d$ with entries in the field $\mathbb{F}$ is denoted by $\mathbb{F}^{m \times d}$. Vectors are assumed column vectors (i.e. elements of $\mathbb{F}_q^{m \times 1}$) unless otherwise stated. The notation $\mathbf{0}$ and $\mathbf{1}$ is used to describe the "all zeroes" and "all ones" vector, respectively; the dimensions are omitted where it is clear from context. Vectors in $\mathbb{F}_q^m$ are identified with elements of $\mathbb{F}_q^{m \times 1}$ – that is, column vectors are considered to be matrices with a single column. If $M \in \mathbb{F}_q^{m \times d}$ then the element in the j^{th} column of the i^{th} row is written as $M_{i,j}$. If $\mathbf{x} \in \mathbb{F}_q^m$ then its i^{th} component is written as $\mathbf{x}_i$.

The function $\text{supp} : \mathbb{F}^m \to [m]$ is defined to be $\mathbf{s} \mapsto \{i \in [m] : \mathbf{s}_i \neq 0\}$, called the *support* of $\mathbf{s}$. The *Hamming weight* of a vector is defined as $\mathsf{HW}(\mathbf{s}) \coloneqq |\text{supp}(\mathbf{s})|$. For a matrix $M \in \mathbb{F}^{m \times d}$, its transpose will be denoted by $M^\top$. Given a set $S \subseteq [m]$, the submatrix denoted by M_S is the matrix formed by concatenating all rows of M indexed by S. If S is a singleton set $\{i\}$ then M_i is used in place of $M_{\{i\}}$.

The definition of the orthogonal complement $V^\perp$ of a vector space $V \subseteq \mathbb{F}^d$ is as follows:

$$V^\perp = \left\{ \mathbf{w} \in \mathbb{F}^d : \langle \mathbf{v}, \mathbf{w} \rangle = 0 \right\}$$

where $\langle \mathbf{v}, \mathbf{w} \rangle = \mathbf{v}^\top \cdot \mathbf{w}$ is the standard inner product. A linear map $L : V \to W$ between vector spaces $V \subseteq \mathbb{F}^d$ and $W \subseteq \mathbb{F}^m$ can be represented by a matrix $M \in \mathbb{F}^{m \times d}$, with respect to some choice of bases for V and W. By the Fundamental Theorem of Linear Algebra, $\text{im}(M^\top) = \ker(M)^\perp$. The space $\text{im}(M^\top)$ is normally described as the coimage of M and is denoted by $\text{coim}(M)$, which is the space spanned by the rows of M. Similarly, the space $\ker(M^\top)$ is called the cokernel of M and is denoted by $\text{coker}(M)$.

The level of security a protocol offers is parameterized by the *computational security parameter*, κ, and the *statistical security parameter*, σ, which are used to quantify the ability of an adversary to break the scheme. Roughly speaking, the former is used to quantify the computational power required to break the scheme and the latter to bound the ability of the adversary to learn secret information or the chance of cheating without detection. Later, there will be detailed discussion of how these parameters are used to define the security of a scheme.

[1] A special notation will be used for the share vector of a secret in a linear secret-sharing scheme but they will *not* be written in bold. This is discussed in Section 2.4.

2.1.2 Complexity

The notation ascribed to Bachmann and Landau for describing the asymptotic behaviour of functions will be used throughout. A function $f : \mathbb{Z} \to \mathbb{R}$ is said to be $O(g)$ if $|g|$ eventually bounds $|f|$ from above; that is, if $\exists\, K \in \mathbb{R}$ and $\exists\, N \in \mathbb{N}$ such that $|f(n)| \leq K \cdot |g(n)|$ for all $n > N$. The function is said to be $\Omega(h)$ if $|h|$ eventually bounds $|f|$ from below; that is, $\exists\, K \in \mathbb{R}$ and $\exists\, N \in \mathbb{N}$ such that $|f(n)| \geq K \cdot |g(n)|$ for all $n > N$. The function f is said to be $\Theta(f')$ if it is both $O(f')$ and $\Omega(f')$.

The two primary models of computation considered in this thesis are *circuits* and *Turing machines*. The precise formulations will not be given as they can be found in any undergraduate textbook on complexity theory, and instead the focus here is on agreeing the naming conventions used in later chapters.

A circuit is a representation of a function $f : R^{k_1} \to R^{k_2}$, where R is a ring and k_1 and k_2 are the numbers of inputs and outputs, respectively, as a directed acyclic graph for which each vertex has indegree 2 and outdegree 1. Vertices are called *gates* and edges are called *wires*, and a certain set of wires are designated as *input wires* and another set as *output wires*. Arithmetic circuits evaluate functions for any ring R, where all gates represent either addition, $+$, or multiplication, $\times$, in R. Boolean circuits are a special case in which the ring is $\mathbb{F}_2$, where addition and multiplication correspond to the logical operations XOR and AND, respectively. The circuit is evaluated on an input by setting the appropriate input wires and proceeding gate-by-gate through the graph, executing the operation defined by the vertex at each step. Circuits are called a *uniform* model of computation since all inputs have a fixed length – specifically, the number of input wires specified in the circuit description.

A Turing machine is an abstract machine that processes inputs on a read-only *input tape* according to some program description called a *protocol*, possibly making use of a *random tape* which is another read-only tape that contains an infinitely long string of uniformly random inputs, and that writes to an *output tape*. Processing involves reading from and writing to different tapes, and changing between finitely-many states. Turing machines are *non-uniform*, meaning they can take arbitrary length (but polynomially-bounded) inputs. An interactive Turing machine (ITM) is a Turing machine that has an input tape that can be written to by other ITMs. Because it depends on inputs provided by other ITMs, it is said to be *reactive*.

A Turing machine $\mathcal{A}$ is said to have *oracle* or *black-box* access to an oracle $\mathcal{O}$ if it can make queries and receive responses but does not observe the internal behaviour – the

oracle is therefore a "black box" to the querying entity. This access will be denoted by $\mathcal{A}^{\mathcal{O}}$.

A function will be called *efficiently computable* (or just *efficient*) if it can be computed by a Turing machine in polynomial time, which means that the number of reads from tapes, writes to tapes, and state transitions can be expressed as a polynomial function of the input length. An algorithm is called probabilistic polynomial-time (PPT) if it is probabilistic – i.e. part of the input is a polynomially-bounded random tape – and runs in polynomial-time. Throughout this thesis, a computation is said to run in polynomial time, or a probability is said to be negligible, if it is polynomial or negligible, respectively, as a function of the relevant security parameter.

More generally, an event will be said to occur polynomially-many times if it occurs a number of times that grows as a polynomial function in the input length. If an event occurs polynomially-many times but the precise polynomial is unimportant, the function is denoted by $\text{poly} : \mathbb{R} \to \mathbb{R}$. A function is called negligible, and where the specific function is unimportant is denoted by $\text{negl} : \mathbb{N} \to \mathbb{R}$, if for every polynomial $p \in \mathbb{Z}[X]$, there exists an $N \in \mathbb{N}$ such that $\text{negl}(n) \leq 1/p(n)$ for all $n > N$.

Cryptographic assumptions are based on the idea that superpolynomial-time algorithms become intractible for a large enough input length (i.e. computational security parameter). The choice to parameterize security in terms of exponential (and negligible) functions is essentially arbitrary, although a useful feature of such functions is that a negligible function multiplied by a polynomial function is still negligible. The upshot of this is that a polynomially-bounded adversary still has negligible advantage in winning a security game even if it is permitted to observe an event that occurs with negligible probability a polynomial number of times. Consequently, one can choose the security parameters in such a way that a computationally-bounded adversary *cannot* win a security game except with negligible probability.

The class of non-deterministic polynomial-time algorithms NP can be defined as the set of problems solvable in polynomial time by a non-deterministic Turing machine (i.e. those that allow branching in the protocol description), or as the set of problems whose solutions are verifiable in polynomial time.

2.1.3 Statistics and Probability

An event will be said to occur with overwhelming probability in the security parameter λ if it occurs with probability at least $1 - \text{negl}(\lambda)$, and with high probability if it

occurs with probability at least $1 - 1/\text{poly}(\lambda)$.

An *ensemble* or *family* of distributions is a sequence of distributions over the same sample space Ω, parameterized by a subset of $\mathbb{N}$. For a set S, the notation $x \leftarrow \mathcal{U}(S)$ is used to say that x is sampled uniformly at random from S. For a distribution $\mathcal{D}$ over a sample space Ω, the notation $x \leftarrow \mathcal{D}(\Omega)$ is used to say that x is sampled from Ω according to distribution $\mathcal{D}$. More generally, one defines a σ-algebra over Ω and a distribution $\mathcal{D}$ over this algebra which enables sampling a set $S \leftarrow \mathcal{D}(\Omega)$; in this thesis, this is taken to mean that $|S|$ elements are successively, independently sampled from Ω according to distribution $\mathcal{D}$. A set S is said to be sampled *subject to* some constraint if it is sampled from the space of all possible sets where the constraint holds; for example, another way of writing that a set $\{x_i\}_{i=1}^n \leftarrow \mathcal{U}(\mathbb{F})$ is sampled subject to $\sum_{i=1}^n x_i = 0$ is to sample $\{x_i\}_{i=1}^n \leftarrow \mathcal{U}\left(\left\{\{x_i\}_{i\in[n]} : \sum_{i\in[n]} x_i = 0\right\}\right)$. In this particular case (that appears frequently in this thesis), it is equivalent to sampling $i^* \leftarrow \mathcal{U}([n])$ arbitrarily, sampling $\{x_i\}_{i\in[n]\setminus\{i^*\}} \leftarrow \mathcal{U}(\mathbb{F})$, and setting $x_{i^*} := -\sum_{i\in[n]\setminus\{i^*\}} x_i$.

Computational Indistinguishability

Definition 2.1 (Computational indistinguishability)**.** Two ensembles of distributions $\mathcal{D} = \{\mathcal{D}_k\}_{k\in\mathbb{N}}$ and $\mathcal{E} = \{\mathcal{E}_k\}_{k\in\mathbb{N}}$, where each distribution is over the same sample space Ω, are said to be *computationally indistinguishable* or *computationally close* if for every PPT Turing machine $\mathcal{A}$, there exists a negligible function $\text{negl} : \mathbb{N} \to \mathbb{R}$ and some $K \in \mathbb{N}$ such that

$$\left| \Pr_{x \leftarrow \mathcal{D}_k(\Omega)}[\mathcal{A}(x) = 1] - \Pr_{x \leftarrow \mathcal{E}_k(\Omega)}[\mathcal{A}(x) = 1] \right| < \text{negl}(k)$$

for all $k > K$. This indistinguishability is denoted by $\mathcal{D} \sim_c \mathcal{E}$.

Statistical Indistinguishability

Definition 2.2 (Statistical distance/Total variation distance)**.** Let $\mathcal{D}$ and $\mathcal{E}$ be two distributions over the same sample space Ω. Then the statistical distance is defined as

$$\Delta(\mathcal{D},\mathcal{E}) = \sup_{S\subseteq\Omega} \left\| \Pr_{X \leftarrow \mathcal{D}(\Omega)}[X = S] - \Pr_{X \leftarrow \mathcal{E}(\Omega)}[X = S] \right\|_\infty .$$

In the case where Ω is countable, the definition

$$\Delta(\mathcal{D},\mathcal{E}) := \tfrac{1}{2} \cdot \sum_{\omega\in\Omega} \left| \Pr_{x \leftarrow \mathcal{D}(\Omega)}[x = \omega] - \Pr_{x \leftarrow \mathcal{E}(\Omega)}[x = \omega] \right|$$

can be used, which may be easier to compute.

Definition 2.3 (Statistical indistinguishability (version 1))**.** Two families of distributions $\mathcal{D} = \{\mathcal{D}_k\}_{k\in\mathbb{N}}$ and $\mathcal{E} = \{\mathcal{E}_k\}_{k\in\mathbb{N}}$, where each distribution is over the same sample space Ω, are said to be *statistically indistinguishable* or *statistically close* if there exists a negligible function $\text{negl} : \mathbb{N} \to \mathbb{R}$ for which there exists a $K \in \mathbb{N}$ such that $\Delta(\mathcal{D}_k, \mathcal{E}_k) < \text{negl}(k)$ for all $k > K$. This indistinguishability is denoted by $\mathcal{D} \sim_s \mathcal{E}$.

Statistical indistinguishability may be defined in the following way for comparison with the computational case. Roughly speaking, Definitions 2.3 and 2.4 are equivalent because the distance between the families of distributions is independent of any distinguisher attempting to determine where the distributions differ.

Definition 2.4 (Statistical indistinguishability (version 2))**.** Two ensembles of distributions $\mathcal{D} = \{\mathcal{D}_k\}_{k\in\mathbb{N}}$ and $\mathcal{E} = \{\mathcal{E}_k\}_{k\in\mathbb{N}}$, where each distribution is over the same sample space Ω, are said to be *statistically indistinguishable* or *statistically close* if for every (not necessarily polynomial-time) Turing machine $\mathcal{A}$, there exists a negligible function $\text{negl} : \mathbb{N} \to \mathbb{R}$ and some $K \in \mathbb{N}$ such that

$$\left| \Pr_{x \leftarrow \mathcal{D}_k(\Omega)}[\mathcal{A}(x) = 1] - \Pr_{x \leftarrow \mathcal{E}_k(\Omega)}[\mathcal{A}(x) = 1] \right| < \text{negl}(k)$$

for all $k > K$. This indistinguishability is denoted by $\mathcal{D} \sim_s \mathcal{E}$.

2.1.4 Combinatorics

Knuth Shuffle

The version by Durstenfield [Dur64] of the algorithm due independently to Knuth [Knu97, §3.4.1] and Fisher and Yates [FY48] for randomly permuting a tuple $(1, \ldots, n)$ is given in Figure 2.1.

Knuth Shuffle

Input Positive integer $n \in \mathbb{N}$ and a seed $seed$. (The seed is implicitly used to perform all sampling deterministically.)

Output Random permutation, $\pi \in S_n$.

Method

1. Set $\mathbf{v} = (1, \ldots, n)$.
2. For $i := 1$ to $n-1$,
   a) Sample $j \leftarrow \mathcal{U}(\{i, \ldots, n\})$.
   b) Switch $\mathbf{v}_i$ and $\mathbf{v}_j$.
3. Set π to be the bijection $\pi : [n] \to [n]$ defined by $i \mapsto \mathbf{v}_i$.

Figure 2.1: Knuth Shuffle.

2.2 Universal Composability

To prove a protocol is secure, one sets up a "security game" that models possible behaviour of an adversary and shows that if there exists an adversary that beats the game then some well-established computational hardness assumption is false. In the security game, a challenger poses an arbitrary instance of the computationally-hard problem to the adversary, which must attempt to solve the problem efficiently with non-negligible advantage over guessing the answer to the problem, given oracle access to an adversary against the protocol. If such an adversary can be constructed, then there can be no adversary against the protocol since its existence implies an efficient solution to the hard problem, which does not exist by assumption. This is known as a reduction, since the security of the protocol is reduced to the security of the computational assumption. The protocol is said to be secure under the given assumption.

Security games typically consist of asking the adversary to distinguish between distributions, in which case the *advantage* of the adversary in winning the security game is defined in the following way.

Definition 2.5 (Distinguishing advantage)**.** Let $\mathcal{D}$ and $\mathcal{E}$ be two distributions. The distinguishing *advantage* of a distinguisher $\mathcal{A}$ between these distributions is the value

$$\mathrm{Adv}(\mathcal{A}) := \left| \Pr[\mathcal{A}(\mathcal{D}) = 1] - \Pr[\mathcal{A}(\mathcal{E}) = 1] \right|.$$

Finding the "right" assumption is important from both feasibility and security perspectives: if it is not clear how to prove the protocol is secure based on a given hard

problem, then it may be necessary to prove it secure under a stronger assumption. As an example, consider the hardness assumption called the computational Diffie-Hellman problem (CDH):

Computational Diffie-Hellman Fix a group G generated by some element $g \in G$, so $G = \{g^k : k \in \mathbb{Z}\}$. Given g^a and g^b, compute $g^{a \cdot b}$.

One example of a stronger hardness assumption is the *decisional* Diffie-Hellman (DDH) problem:

Decisional Diffie-Hellman Fix a group G generated by some element $g \in G$, so $G = \{g^k : k \in \mathbb{Z}\}$. Given (g^a, g^b, g^c), determine if $c = a \cdot b$.

This is a stronger hardness assumption since any adversary with access to a CDH adversary solving the computational problem can be used to determine $g^{a \cdot b}$, and compared with the value g^c to give a solution to the DDH problem.

Towards a discussion of the language of "oracle access", consider the *gap* Diffie-Hellman problem:

Gap Diffie-Hellman Fix a group G generated by some element $g \in G$, so $G = \{g^k : k \in \mathbb{Z}\}$. Given g^a and g^b, find $g^{a \cdot b}$ given oracle access to a decisional Diffie-Hellman oracle.

In this case, the protocol would be considered secure under the so-called "gap Diffie-Hellman" assumption. Another way to say the same thing is to say that the protocol is secure under the computational Diffie-Hellman assumption given access to a decisional Diffie-Hellman oracle. This is a more generic way to describe the security of protocols under standard security assumptions without the need to invent names for intermediate assumptions. Indeed, this language is by far the most common in the security framework that will be considered in this thesis.

Motivating Composable Security

Historically, most cryptographic protocols have been proved secure in the standalone model, in which one is only concerned about the security of a single execution of the protocol, isolated from any other information potentially in the system in which the protocol is run. In particular, this means the adversary may only query oracles whose scope is limited to the security game. Consequently, proofs in this model offer no claims

of security when executing multiple protocols (or the same protocol multiple times simultaneously) using the same oracles (understood as PPT ITMs) for every game. For example, if a game forbids the adversary from making specific oracle queries based on the messages sent to and from the challenger of one game, this does not prevent the adversary from making the forbidden oracle queries in a second game. This is not always directly problematic, but it highlights that designing cryptographic protocols that are secure even when executed as part of a larger system while maintaining security is non-trivial.

This problem motivated the development of a composable security framework in which protocols could build on the security of others without the need to prove the security from the ground up each time, mirroring the standard mathematical approach of building theorems by first proving lemmata and propositions. The UC framework introduced by Canetti [Can00] is a security definition with a strong composability guarantee: any protocol proved secure retains its security even when executed alongside, sequentially or simultaneously, arbitrarily many other protocols, or even the same protocol. At its core is the composition theorem:

Theorem 2.1 (Informal, [Can00])**.** *Suppose the protocol* Π *UC-securely realizes* $\mathcal{F}$*, and that* Π' *UC-securely realizes* $\mathcal{F}'$ *and uses* $\mathcal{F}$ *as a subroutine. Then* Π' *UC-securely realizes* $\mathcal{F}'$ *when replacing* $\mathcal{F}$ *with* Π*.*

The upshot of this theorem is that if a protocol makes use of a functionality as an oracle, then this functionality may be replaced by any subprotocol that securely realizes it and the main protocol retains the same security. Multi-party computation (MPC) is a perfect use case for composable protocols since the goal of computation on private data is a complex task requiring multiple cryptographic primitives to achieve full security against an active adversary. In this thesis the variant of UC taken from [Can00, v.2018/12/31, §4.4.2] is used, in which the simulator interacts with the adversary in a black-box way; Canetti showed that this is equivalent to the general definition of UC security that he gives.

Defining Composable Security

To understand composable security, it is helpful to recall the more general notion of security for cryptographic protocols. To prove the security of a protocol Π in practice, one constructs a trusted third party called a functionality $\mathcal{F}$ that performs an "ideal" version of Π, that leaks only an "ideal" amount of information to the adversary – that is, it leaks

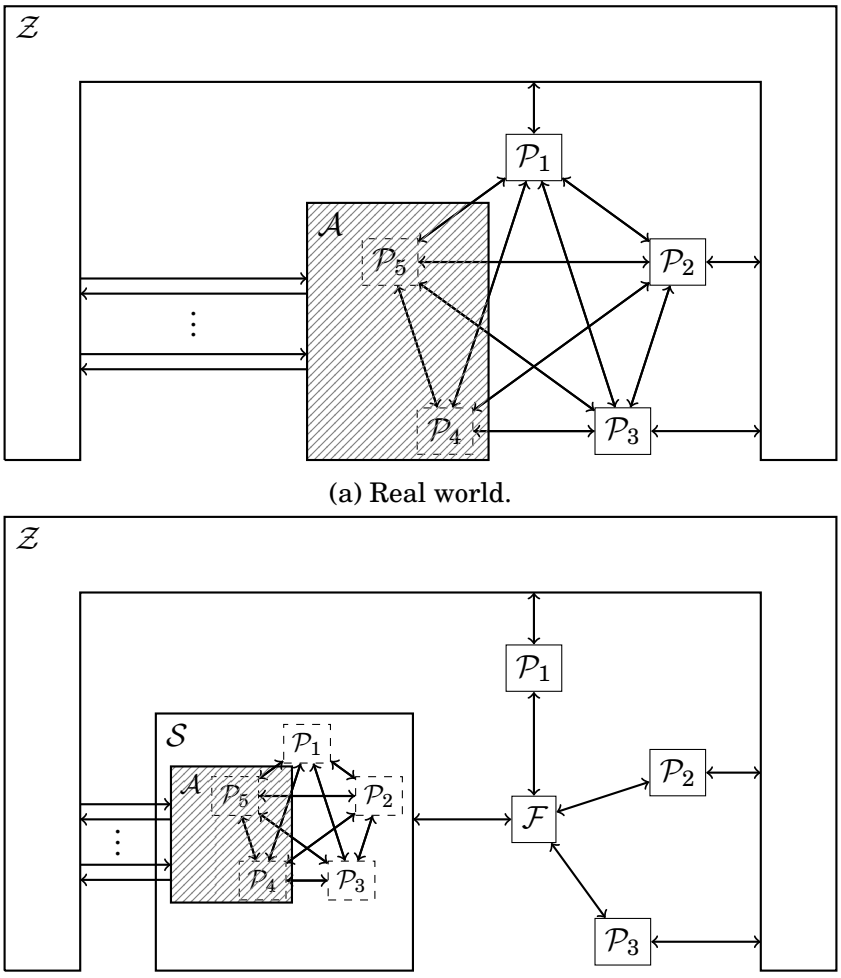
exactly as much as the protocol architect decides is acceptable, and no more. One then argues that Π leaks *at most as much* as $\mathcal{F}$ by showing that the leakage revealed in the execution of Π can be simulated using only the ideal leakage from $\mathcal{F}$. Since $\mathcal{F}$ is secure by construction and nothing more can be learnt from Π than can be learnt from $\mathcal{F}$, the protocol is secure. More concretely, one shows that for any "real-world" adversary $\mathcal{A}$ against Π, there exists an "ideal-world" adversary $\mathcal{S}$, called the *simulator*, against $\mathcal{F}$ with black-box access to $\mathcal{A}$, that simulates the protocol towards $\mathcal{A}$, such that $\mathcal{A}$ cannot tell whether it interacted with $\mathcal{S}$ or with real honest parties.

Towards the goal of protocol composition, a stronger guarantee is required: the executions should appear the same not only to the adversary, but to any so-called *environment* in which the protocol is executed. The execution environment, more formally, is a non-uniform PPT ITM distinguisher $\mathcal{Z}$ that must determine which world of the two worlds – ideal or real – is being executed, with non-negligible advantage over guessing. The environment is more than a mere observer of the execution: $\mathcal{Z}$ is allowed to choose honest parties' inputs and observe their final outputs, and may interact arbitrarily with $\mathcal{A}$ (which controls the corrupt parties) throughout the protocol, including specifying the code $\mathcal{A}$ runs. The only information hidden from $\mathcal{Z}$ is the *intermediate* communication and computation between the first inputs and final outputs of any honest party (either executing as in the description of the protocol or with the ideal functionality), and their random tapes. The idea behind this definition of $\mathcal{Z}$, which has a considerable amount of information to help it distinguish, is that the protocol may be run in some setting in which inputs are received by or sent to some other – potentially corrupted – process in a larger system, and that the corrupt parties in the current protocol execution may be under control of the environment. In other words, the environment captures exactly everything that is external to the protocol execution.

With these "composable" modifications to the security modelling, a protocol is said to realize a functionality UC-securely if for every real-world adversary interacting with real honest parties in the protocol there exists an ideal world simulator interacting with the functionality such that the executions of the two worlds are indistinguishable from one another to the environment.

The notation $\mathcal{F}$ will be used to denote the ITM for the ideal functionality itself, as well as the algorithm it runs. Similarly, $\mathcal{P}_i$ denotes the i^{th} party out of a set $\mathcal{P}$ of n parties, indexed by $[n]$ and executes the corresponding instructions of the protocol written as Π. The environment is a PPT distinguisher $\mathcal{Z}$ that, given any adversary $\mathcal{A}$, a random tape, a security parameter, and an execution of Π or $\mathcal{F}$, outputs a guess as

to which world was executed. The two different possible worlds in which the execution occurs are shown in Figure 2.2. Notice that in both the real and ideal executions, the environment is being made to "think" it is interacting with real honest parties as in the protocol description.



(a) Real world.

(b) Ideal world.

Figure 2.2: Real World Versus Ideal World.

To formalize the definition, for a fixed PPT $\mathcal{A}$ acting on behalf of corrupt parties, and either the protocol Π or the algorithm executed by $\mathcal{F}$, for every $\mathcal{Z}$ the following random variables are defined:

$$\text{EXEC}(\mathcal{Z},\mathcal{A},\Pi) := \left\{\text{EXEC}(\mathcal{Z},\mathcal{A},\Pi)(\lambda,\mathbf{z})\right\}_{\lambda\in\mathbb{N},\mathbf{z}\in\{0,1\}^*}$$

and

$$\text{EXEC}(\mathcal{Z},\mathcal{S}^{\mathcal{A}},\mathcal{F}) := \left\{\text{EXEC}(\mathcal{Z},\mathcal{S}^{\mathcal{A}},\mathcal{F})(\lambda,\mathbf{z})\right\}_{\lambda\in\mathbb{N},\mathbf{z}\in\{0,1\}^*}$$

over the set $\{0,1\}$, where $\mathbf{z}$ is an input the environment handed to it at the beginning (roughly speaking the input from another part of the system, and can be thought of as an "auxiliary information" tape) and λ is the security parameter, and the distribution is taken over the random tapes of all ITMs. The variable $\text{EXEC}()$ outputs 0 if $\mathcal{Z}$ guesses the execution is ideal and 1 if $\mathcal{Z}$ guesses the execution is real. The security guarantee is that for every $\mathcal{A}$ there exists an $\mathcal{S} = \mathcal{S}^{\mathcal{A}}$ such that for every $\mathcal{Z}$ it holds that

$$\text{EXEC}(\mathcal{Z}, \mathcal{A}, \Pi) \sim_s \text{EXEC}(\mathcal{Z}, \mathcal{S}^{\mathcal{A}}, \mathcal{F}).$$

In more detail, if the indexing set of corrupt parties is denoted by A and the number of communication rounds (defined in detail later) by r, then it must hold that

$$\left\{\begin{array}{l} \{\text{INPUTS}_{\mathcal{P}_i}\}_{i\in[n]\setminus A}, \\ (\{\text{MESSAGES}_{\mathcal{A}\leftrightarrow\mathcal{P}_i}\}_{i\in[n]\setminus A})_{k=1}^{r}, \\ (\text{MESSAGES}_{\mathcal{A}\leftrightarrow\mathcal{Z}})_{k=1}^{r}, \\ \{\text{OUTPUTS}_{\mathcal{P}_i}\}_{i\in[n]\setminus A} \end{array}\right\}_{\lambda\in\mathbb{N},\mathbf{z}\in\{0,1\}^*} \sim_s \left\{\begin{array}{l} \{\text{INPUTS}_{\mathcal{P}_i}\}_{i\in[n]\setminus A}, \\ (\{\text{MESSAGES}_{\mathcal{A}\leftrightarrow\mathcal{S}_{\mathcal{P}_i}}\}_{i\in[n]\setminus A})_{k=1}^{r}, \\ (\text{MESSAGES}_{\mathcal{A}\leftrightarrow\mathcal{Z}})_{k=1}^{r}, \\ \{\text{OUTPUTS}_{\mathcal{P}_i}\}_{i\in[n]\setminus A} \end{array}\right\}_{\lambda\in\mathbb{N},\mathbf{z}\in\{0,1\}^*}$$

where the distributions are taken over the random tapes of honest parties, and the random tape of $\mathcal{A}$ which is determined by the environment's auxiliary information tape $\mathbf{z}$, and $\mathcal{S}_{\mathcal{P}_i}$ denotes the part of the simulator $\mathcal{S}$ that is responsible for emulating $\mathcal{P}_i$. Note that only the *honest* parties are assumed to have inputs, reflecting the fact that $\mathcal{A}$ merely sends messages and $\mathcal{S}$ deduces the implicit inputs to pass on to $\mathcal{F}$. The "inputs" of corrupt parties are therefore included in the distribution of the execution implicitly as part of the variable $\text{MESSAGES}_{\mathcal{A}\leftrightarrow\mathcal{S}}$, not in the INPUTS variable. Similarly, corrupt parties need not produce "final outputs" since $\mathcal{Z}$ can compute anything the corrupt parties can after the final messages in the protocol have been sent.

Notice that the messages amongst all parties, except corrupt to corrupt, should appear in the distribution. Technically speaking, the messages amongst honest parties should be included in the distributions, but usually secure channels are assumed, or broadcasts over authenticated channels, so this communication does not reveal any information to the environment.

Constructing a Simulator

Throughout the simulation, $\mathcal{S}$ runs $\mathcal{A}$ as a black box, and any messages output by $\mathcal{A}$ intended for $\mathcal{Z}$ are handed to $\mathcal{Z}$, and the response is handed back to $\mathcal{A}$. The full transcript as viewed by $\mathcal{A}$ is passed on from $\mathcal{S}$ to $\mathcal{Z}$. As for real honest parties, $\mathcal{Z}$ provides initial

inputs and receives final outputs in exactly the same way in both worlds. Thus the only potential differences between executions are the *values* of the inputs and outputs, and the transcript produced.

Throughout the execution of the protocol between $\mathcal{A}$ and $\mathcal{S}$, the simulator must act on behalf of "emulated" honest parties. The requirements on the distributions derived from the full execution transcript, i.e. all inputs and outputs, and the messages the simulator generates on behalf of emulated honest parties, can be expressed as the following three (informal) criteria:

Correctness Final outputs of all parties in a simulation of the protocol where the simulator is handed the real inputs of honest parties should result in the same output as a real execution of the protocol.

Extractability The simulator must be able to deduce the set of inputs of corrupt parties from the transcript.

Equivocation The simulator must be able to generate a transcript that convinces the environment that the set of all messages sent and received in the execution is self-consistent.

These properties are not entirely independent, but they summarize the key points necessary for simulation. (For example, if Correctness and Equivocation hold, then the simulator must have been able to extract inputs of corrupt parties, or guess them with overwhelming probability.) A couple of important observations are highlighted here.

Extractability of the inputs of $\mathcal{A}$ is necessary in order for $\mathcal{F}$ to compute and return the same output as would have been computed in the real world – i.e. for Correctness to hold. Without this, even if the corrupt parties executed the protocol honestly, no party would output the correct result (with high probability). Note that the ability of the simulator to extract the inputs implies the set of honest parties can also do the same, which is potentially problematic; in practice, however, the simulator is assumed to have access to some trapdoor information in the form of a *setup assumption* to allow it to extract, as discussed in detail in Section 2.2. It is important to keep in mind that corrupt parties are not assumed to have explicit inputs, but implicit inputs are deduced from the communication between $\mathcal{A}$ and $\mathcal{S}$.

For Equivocation to hold, self-consistency of messages is required, which is (informally) defined as follows: the messages for final outputs must be simultaneously consistent with all previous messages *and* must result in the adversary computing the

same output that it would have computed in a real execution, despite the fact that emulated communication throughout the protocol generated by $\mathcal{S}$ does not – indeed, cannot – depend on honest parties' inputs since they are unknown to the simulator in the real-world experiment. Thus Correctness and Extractability on their own are not enough to ensure indistinguishability of transcripts.

Proof of indistinguishability

Once a simulator has been constructed, it is necessary to prove that it indeed provides a view indistinguishable from a view in the real world. To do this, a sequence of (polynomially-many) "hybrid" worlds is defined in which $\mathcal{S}$ is initially given the inputs of all honest parties, and then in each subsequent world $\mathcal{S}$ has one input fewer; in the last hybrid world $\mathcal{S}$ knows the inputs of no honest parties. Thus the first and last worlds are perfectly indistinguishable from the real and ideal worlds respectively, by definition. Then it only remains to prove that consecutive hybrid worlds in the sequence are indistinguishable.

In the most formal treatments, a proof should specify a simulator for each hybrid world and show that there is no distinguisher between any consecutive pair. However, sometimes the ability of the simulator to extract inputs and equivocate outputs is independent of the simulator's knowledge of the honest parties' inputs, and in this case the argument for security can be summarized by showing that the protocol transcript reveals no information to the environment, which trivially means the environment cannot distinguish.

Proving the indistinguishability of worlds will often involve giving a reduction to a primitive being used in the protocol. For example, recall that the simulator must "fake" the inputs of honest parties for which it does not know the input, and suppose that the protocol involves each party broadcasting an encryption of their input. Instead of broadcasting an encryption of an honest party's input (which it does not know), the simulator will send an encryption of 0. Then any environment that can distinguish between worlds can distinguish between an encryption of 0 and an encryption of this input. Thus the security reduces to the security of the encryption scheme.

The ability of the environment to distinguish is parameterized by the security parameters, κ and σ. The following definitions are used throughout this thesis, primarily in theorem statements claiming that protocols securely realize functionalities in the UC framework.

Definition 2.6. A protocol is said to realize a functionality UC-securely with statistical security parameter σ if any environment can distinguish with probability at most $O(2^{-\sigma})$.

Definition 2.7. A protocol is said to realize a functionality UC-securely with computational security parameter κ if there exists an environment for which there is a polynomial-time reduction to a computationally hard problem which has computational security κ.

Setup Assumption

It is reasonable to wonder why the simulator should have the ability to extract (implicit) inputs of the adversary based on the messages it sent: indeed, this is undesirable as it implies the adversary should be able to do similarly from honest parties' messages. The classic example of this problem was given by Fischlin and Canetti [CF01]: the existence of a simulator for a commitment scheme in the UC model implies the commitment cannot be hiding, since the simulator must be able to extract the message from the commitment, and hence any "real" honest party would also be able to extract the message. (See Section 2.3.5 for the properties of commitment schemes.)

This demonstrates that there are some functionalities that can never be realized in the UC framework as described so far. However, giving the simulator some trapdoor information via a trusted setup (i.e. a *setup assumption*) can give secure protocols. In such a situation, instead of giving a protocol in the *plain* (or *standard*) model, a protocol is said to be realized in a *hybrid* model, in which the existence of one or more ideal functionalities (oracles) is assumed.

The key difference to a proof in a hybrid model and a proof in the plain model is that the simulator is required to emulate the functionality to $\mathcal{A}$. This gives the simulator a limited ability to program the oracle with information of its choosing, although the simulated oracle should be indistinguishable from an oracle executed honestly, otherwise the environment can use it to distinguish between worlds.

Random Oracle Model In the random oracle model (ROM), all parties in a protocol execution have access to an oracle that, on input some query, returns a uniformly-sampled output, but always the same output for the same query. In the proof of security for a protocol in the ROM, the simulator is allowed to program the random oracle – that is, it emulates the oracle towards the adversary, but it need not execute the exact ideal

behaviour given in the oracle description. As stated above more generally, the simulator is not allowed to deviate "too much" from the oracle's description, for example by sampling according to a different distribution, since this may allow the environment to determine that a simulation is taking place. By considering the functionality $\mathcal{F}_{\mathsf{RO}}$ in Figure 2.3, the notion of the $\mathcal{F}_{\mathsf{RO}}$-hybrid model coincides with the classical notion of the ROM.

Random Oracle $\mathcal{F}_{\mathsf{RO}}$

Initialize On input $(\texttt{Initialize}, X, sid)$ where sid is a new session identifier and X is a set, create a new dictionary DB with identifiers DB.Ids.

Random Element On input (id, sid), if $id \in \texttt{DB.Ids}$ then return $\texttt{DB}[id]$, and otherwise sample $\texttt{DB}[id] \leftarrow \mathcal{U}(X)$ and return $\texttt{DB}[id]$.

Figure 2.3: Random Oracle Functionality, $\mathcal{F}_{\mathsf{RO}}$.

In the real world, random oracles taking polynomial-length inputs cannot exist since then a truly random oracle must store exponentially-much data while being efficiently queryable. This necessitates the use of hash functions. Typically, protocols require the ROM if stronger properties of a cryptographic hash function are needed than those usually given (i.e. than those given in Section 2.3.1), but will always be instantiated in a real-world protocol using a hash function.

The ROM has received criticism as there are some artificial protocols proved secure in the random oracle model which cannot be instantiated securely by any hash function [CGH98]. Nevertheless, it is generally believed that protocols secure in the random oracle model are secure against practical attacks [BR93].

Common Reference String Model In the common reference string (CRS) model, all parties are handed a string sampled uniformly from an agreed distribution. The benefit of the CRS model over the ROM is that one need not make the heuristic assumption that a hash function behaves like a random oracle. However, it is often more challenging to prove in this model as the simulator has much more limited power to equivocate the transcript.

This model will not be discussed further except to say that it is, in some sense, one of the alternative "base cases" for UC protocols for those who wish to avoid the ROM; for example, Canetti and Fischlin [CF01] showed how to obtain UC commitments in the CRS model.

Global Setup Suppose a protocol Π securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model, and suppose $\widetilde{\Pi}$ securely realizes $\widetilde{\mathcal{F}}$ in the $\mathcal{F}_{\mathsf{RO}},\mathcal{F}$-hybrid model. The composition theorem only guarantees security when each subprotocol has a session-specific instantiation of the random oracle, which in particular means that the same query made to the two different oracles will (with high probability) lead to different outputs. However, in the real world a random oracle is usually replaced with a single hash function everywhere it appears in the protocol description. A model known as the *global* ROM [CDPW07] allows exactly this global replacement of the oracle with a hash function.

Canetti et al. [CDPW07] showed that there is a separation between the attacks mountable against protocols in the non-global random oracle model and those in the global random oracle model. The salient point for this thesis is that all functionalities presented will keep track of the current session using a session identifier sid. If a party calls the functionality with sid different from what was sent in its initialization procedure, the functionality outputs the message Reject to all parties and awaits another message. For the sake of brevity this is taken to be implicit and will not be stated each time a functionality is defined. The first step of almost all protocols in a given hybrid model is for the parties to agree on a session identifier and initialize the oracle, the exact method of which will not be discussed as its choice does not affect the security of the protocols and so may be derived from any public information.

To save on denoting one session identifier for every distinct *type* of oracle (for example, one for a commitment functionality and one for a random oracle), it will be assumed that these functionalities use the *same* session identifier; the conflict only occurs if two of the same type of oracle (for example, two commitment schemes) are initialized with the same sid. This reflects the idea that the whole protocol is "one session", but note that this is merely a choice of notation for this thesis and is not standard.

Theorem Statements

To enable theorem statements with a concrete "number of bits" of security, the following definition is given.

Definition 2.8 (Security with parameter λ)**.** Let $\mathcal{D}_k$ be a sequence of distributions produced by $\mathcal{S}$ and let $\mathcal{E}_k$ be a sequence of distributions produced by $\mathcal{A}$. A scheme is said to be secure with security λ (bits) if the parameter k is chosen so that $\mathrm{negl}(k) \leq 2^{-\lambda}$, where negl is taken from Definition 2.3 in the statistical case $\lambda = \sigma$, or from Definition 2.1 in the computational case $\lambda = \kappa$.

Theorem statements in this thesis will use the terminology of the following definition.

Definition 2.9. The protocol Π is said to realize $\mathcal{F}$ UC-securely with statistical security σ if for every $\mathcal{A}$ there exists an $\mathcal{S}$ such that the distinguishing advantage of any environment $\mathcal{Z}$ is a negligible function in σ.

Observations And Usage of UC in This Thesis

Functionality Interactions Throughout this thesis the functionalities will be described as interacting with the ideal-world adversary, that is, the simulator $\mathcal{S}$. In the literature, functionalities are variously described as interacting nondescriptly with "the adversary" or directly with the environment. The choice to talk about interaction with $\mathcal{S}$ is, compared with the former, to emphasize that the adversary is "ideal", and compared with the latter to highlight that the environment should not "know" whether or not the functionality is being executed.

Agreement Outside the Protocol To avoid cluttering the functionalities with extraneous information, in most of the algorithms presented there is no discussion of how to agree on public information such as which parties are to execute a given protocol, the access structure they assume or the linear secret-sharing scheme (LSSS) that will be used. The reason for this is that such information is beyond the scope of the protocol: for example, provided the functionality accepts a single set of parties to execute the protocol, its security holds.

Furthermore, all of the protocols in this thesis are secure against static adversaries only, which means that the adversary can corrupt parties once at the start of the execution but not dynamically throughout. For this reason, there will not be discussion of special messages sent from the adversary to the functionality to allow the corruption of parties, as is common in the literature, and instead the functionality will be assumed already to know which parties are corrupt, and therefore how to interact with different parties.

The Rushing Adversary In a synchronous network setting, the adversary is allowed to be "rushing" in the sense that it can receive all messages from all honest parties before deciding what message it will send. This is a weak assumption – i.e. the protocols have a stronger security guarantee under this assumption – but it often makes the

simulation a little more involved. As this thesis deals with rushing adversaries, in simulation proofs the simulator will always send input to the adversary before it receives the messages in the same round of communication.

Rewinding One key way in which the UC framework differs from that of the standalone model in terms of the proof technique is that the adversary cannot be rewound by the simulator, since this would be observed by the environment and provide a way for it to distinguish between worlds. The ability to rewind the adversary is a crucial ability in some proofs, such as in proving soundness of zero-knowledge proof (ZKP) protocols. To prove the indistinguishability of the transcripts, it is of course possible to rewind the *environment*; however, this will never enable extraction of the inputs required for the simulation since the reduction sits outside of the environment.

Such strong security comes at a cost: for even some "natural" cryptographic primitives it has been shown there cannot exist a protocol that UC-securely realizes them without a setup assumption. For example, to motivate their UC-secure commitment protocol secure in the CRS model, Canetti and Fischlin [CF01] explained that any UC-secure commitment protocol in the plain model *cannot* hide the committing party's inputs (which is a required property of such schemes), since the UC-security implies there is a simulator that can extract the message from the commitment without any trapdoor information, which means that any corrupt party can do this in the real world with any honest party's commitment.

Convention for Description of Messages Sent In the proofs in this thesis, it will be stated that messages are sent between the adversary and simulator to mean that, on behalf of each honest party, the simulator communicates one or more messages to each corrupt party (controlled by $\mathcal{A}$) or *vice versa*, over the relevant point-to-point channel. The reason for this abstraction is that such explanation is cumbersome and distracting from the important points of the simulation. It *is* important, however, to note that messages sent from the adversary to the simulator need to be parameterized by an index for honest parties, or otherwise express that this happens, because corrupt parties might send different data to different honest parties.

Combined Security In the theorem statements, for protocols realized in hybrid models the level of computational or statistical security is described assuming a *perfect* realization of the hybrid functionalities. This is to avoid making any assumptions on the

realization of these sub-functionalities. In order to determine the overall security of a given protocol, one must look at the security of the realization of *every* functionality. The computational or statistical security of a protocol is the "worst" computational or statistical security out of the protocols chosen to realize the hybrid-world functionalities.

Relaying Messages To the fullest possible extent, the job of the simulator is to act as a relay between $\mathcal{A}$ and $\mathcal{F}$, "translating" messages into the appropriate format back and forth. Thus the simulator is sometimes said to "relay" messages between the two. This is particularly useful language in the following note on extending functionalities since messages need no translation there.

Extending a Functionality One common way to create a functionality is to construct a functionality that performs a subset of the desired commands and then to "extend" it with new commands; in this case, the latter functionality is said to *extend* the former. This approach can be seen, for instance, in [KOS16]. This language will be used in functionalities in this thesis.

Subprotocols Sometimes protocols are said to "make use of a subprotocol". The idea behind this is simply to make the design more modular and the presentation cleaner. Subprotocols are not intended to realize any ideal functionality, since usually this is not possible on their own: it is a presentational choice. Given a protocol Π_A that uses a subprotocol Π_B, the unified protocol is written as $\Pi_A \| \Pi_B$ (or $\Pi_B \| \Pi_A$).

2.3 Cryptographic Primitives and Basic Tools

This section contains definitions of fundamental tools used in cryptography. The proofs are standard.

2.3.1 Hash functions

Hash functions are used to map arbitrary-length inputs to bit-strings of a fixed length. Cryptographic hash functions are required to satisfy one or more guarantees on the level of "unpredictability" of outputs, and consequently are often used to instantiate random oracles efficiently. More formally, a cryptographic hash function H satisfies one or more of the following properties:

First preimage resistance Given a value h in the image, it should be hard to find m such that $h = H(m)$.

Second preimage resistance Given m, it should be hard to find m' such that $H(m') = H(m)$.

Collision resistance It should be hard to find two messages m and m' for which $H(m) = H(m')$.

Some protocols require the use of a random function, which is instantiated using a hash function, and the security reduces to the heuristic assumption that a hash function is "random enough". However, some protocols do not need the full force of a random oracle, in which case the security may depend on, for instance, the assumption that the hash function is collision-resistant.

2.3.2 Pseudorandom Functions

A pseudorandom function (PRF) is a family of efficiently samplable functions whose members are efficiently computable functions that are computationally-indistinguishable from random functions (i.e. an instance of a random oracle). More formally, a PRF

$$\{F_k(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa\}_{k \in \{0,1\}^\kappa}$$

satisfies the following properties:

Samplability Sampling $F_k(\cdot) \leftarrow \mathcal{U}\left(\{F_k(\cdot)\}_{k \in \{0,1\}^\kappa}\right)$ is efficient.

Computability For any $k \in \{0,1\}^\kappa$, for any $x \in \{0,1\}^*$, $F_k(x)$ can be computed in time $\text{poly}(|x|)$.

Pseudorandomness For any $k \in \{0,1\}^\kappa$, it holds that $\{F_k(x)\}_{x \in \{0,1\}^*} \sim_c \mathcal{U}(\{0,1\}^\kappa)$.

2.3.3 MACs

A message authentication code (MAC) is a method of ensuring the validity of a message: specifically, that the message was not altered in transit, and that the message indeed comes from the purported source. Their specific use in MPC is described in Section 2.5.3.

2.3.4 Communication Channels

In almost all MPC protocols for circuits, linear operations come "for free" in the sense that they require little computation and no – or little – communication, whereas multiplications, and non-linear operations in general, require interaction.

A round of communication can be defined as a period of time in which parties send all possible necessary information from the protocol that is not dependent on other parties' messages sent in the same round. The key metrics for communication complexity are the number of rounds and the amount of data sent in a given round. These metrics are used because they are independent of the network hardware, which determines the network *latency* and *bandwidth*.

Throughout this thesis, synchronous communication will be assumed. In this communication model, all messages for a given round are assumed to be delivered successfully before any parties send any messages for the following round. Protocols allowing for asynchrony in the network more accurately model communication over wide-area networks (WANs) such as the Internet but are generally more complex.

Authenticated Channels

The recipient of a message sent over an authenticated channel is sure that the purported sender is indeed the sender, and that the message has not been tampered with in transit. Such channels do not guarantee privacy of communication. Given a graph where the vertex set is the indexing set of parties $[n]$ and edges E are connections between parties, the set of authenticated channels is denoted by $\mathsf{AC}(E)$.

Secure Channels

A secure channel is an authenticated channel which additionally guarantees privacy of communication[2]. To realize a secure channel, parties use an authenticated channel to agree on public-key/secret-key pairs, use these to establish a shared symmetric key, and then use the symmetric key to encrypt all communication, which can then be sent over the authenticated channel. Given a graph where the vertex set is the indexing set of parties $[n]$ and edges E are connections between parties, the set of secure channels is denoted by $\mathsf{SC}(E)$.

[2]One can alternatively define secure channels with privacy but without authentication.

Broadcast Channels

A message sent via a broadcast channel is one in which every honest party (eventually) agrees it received the same message. The concern in this thesis is only on protocols secure *with abort* (explained in Section 2.5), which simplifies the broadcasting procedure.

The functionality $\mathcal{F}_{\mathsf{Broadcast}}$ is given in Figure 2.4 and the protocol $\Pi_{\mathsf{Broadcast}}$ in Figure 2.5. Notice that the protocol need not make use of the random oracle since in the simulation the simulator just relays the messages directly to the functionality.

Functionality $\mathcal{F}_{\mathsf{Broadcast}}$

Initialize On input $(\texttt{Initialize}, sid)$ from all parties, where sid is a new session identifier, set `Abort` to false.

Broadcast On input $(\texttt{Broadcast}, x, sid)$ from $\mathcal{P}_i$, or from $\mathcal{S}$ if $i \in A$,
If $i \in [n] \setminus A$, for all $j \in [n] \setminus \{i\}$ send x to $\mathcal{P}_j$ or to $\mathcal{S}$ if $j \in A$.
If $i \in A$, await a set of values $\{x^j\}_{j \in [n] \setminus A}$ and for each $j \in [n] \setminus A$, send x^j to $\mathcal{P}_j$. If $x^i \neq x^j$ for any $i, j \in [n] \setminus A$, then set `Abort` to true.

Verify On input $(\texttt{Verify}, sid)$ from all parties and $\mathcal{S}$, await a message Abort or OK from $\mathcal{S}$. If the message is OK and `Abort` is false, send the message OK to all honest parties and continue; otherwise, send the message Abort to all honest parties, (locally) output $\perp$, and then halt.

Figure 2.4: Broadcasting Functionality, $\mathcal{F}_{\mathsf{Broadcast}}$.

Protocol $\Pi_{\mathsf{Broadcast}}$

Initialize Each party initializes an empty string str and they agree on a collision-resistant hash function, H.

Broadcast For party $\mathcal{P}_i$ to broadcast a value x to all parties:

1. Party $\mathcal{P}_i$ sends x to all $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ over an authenticated channel.
2. Each party $\mathcal{P}_j$ updates $str^j := str^j \| x$ where $\|$ denotes string concatenation.

Verify To verify all broadcasts so far, each party $\mathcal{P}_i$ does the following:

1. Compute $h^i := H(str^i)$.
2. For all $j \in [n] \setminus \{i\}$, send h^i to $\mathcal{P}_j$ over an authenticated channel.
3. Await h^j from $\mathcal{P}_j$ for all $j \in [n] \setminus \{i\}$.
4. If $h^j \neq h^i$ for any j, send the message Abort to all other parties, (locally) output $\perp$, and halt.

Figure 2.5: Broadcasting Protocol, $\Pi_{\mathsf{Broadcast}}$.

Note that when an honest party aborts, another round of communication is needed to

ensure all other honest parties abort, but this is not the same as the classical Byzantine agreement [LSP82] problem since honest parties always abort if any other party tells them to abort – they do not have to decide whether or not to abort. Note that secure channels are also authenticated so it is assumed each party knows the identity of the sender of each message.

Theorem 2.2. *The protocol* $\Pi_{\mathsf{Broadcast}}$ *UC-securely realizes the functionality* $\mathcal{F}_{\mathsf{Broadcast}}$ *against a static, active, computationally-bounded adversary in the plain model, assuming the parties are connected by a complete network of authenticated channels.*

Proof. Simulation is straightforward: the simulator emulates the honest parties honestly, and simply acts as a relay between $\mathcal{A}$ and $\mathcal{F}_{\mathsf{Broadcast}}$; then $\mathcal{S}$ sends the message Abort to $\mathcal{F}_{\mathsf{Broadcast}}$ if the hash comparison fails, and otherwise sends OK.

A distinguisher $\mathcal{Z}$ between the worlds can be used to break the collision-resistance of the hash function in the following way. Because $\mathcal{S}$ simply relays messages, the only difference between the simulation and the real-world execution is that in the real world the parties only abort if the hashes agree, not the messages themselves. Thus $\mathcal{Z}$ can distinguish only if in the real world the adversary sends different messages to different honest parties so that they do *not* abort in the protocol execution (whereas in the ideal execution they would always abort in this case). The only way of doing this is by finding two sequences of messages, i.e. two strings str^i and str^j, such that $str^i \neq str^j$ but $H(str^i) = H(str^j)$, violating the assumption of collision-resistance of the hash function. Note that the environment can choose all messages in both worlds, since these are the parties' inputs, so second-preimage resistance is not sufficient. □

2.3.5 Commitments

A commitment scheme is a way of ensuring a party is not able to change its inputs after observing the inputs of other parties. It has two intuitive properties:

Hiding The commitment should not reveal anything about the corresponding message.

Binding The party that generated the commitment should not be able to claim convincingly that the message is anything but what was originally used to generate the commitment.

The ideal commitment functionality $\mathcal{F}_{\mathsf{Commit}}$ is given in Figure 2.6 and a standard protocol securely realizing it is given in Figure 2.7.

Functionality $\mathcal{F}_{\mathsf{Commit}}$

Initialize On input $(\texttt{Initialize}, X, sid)$ from all honest parties and $\mathcal{S}$, where X is a set, initialize a new dictionary DB with indexing set DB.Ids.

Commit On input $(\texttt{Commit}, i, x, sid)$ from honest party $\mathcal{P}_i$, or from $\mathcal{S}$ if $\mathcal{P}_i \in \mathcal{A}$, and $(\texttt{Commit}, i, \perp, sid)$ from all other parties, if $x \in X$, compute a new identifier id_x, store $\texttt{DB}[id_x] := (x, i)$, send id_x to all parties and $\mathcal{S}$ and continue.

Open On input $(\texttt{Open}, i, id_x, sid)$ from all honest parties and $\mathcal{S}$, if $(id_x, i) \in \texttt{DB.Ids}$, then send x to all honest parties and $\mathcal{S}$ and continue. Otherwise, await a message Abort or OK from $\mathcal{S}$. If the message is OK, then send x to all honest parties and continue; otherwise, send the message Abort to all honest parties, and then halt.

Figure 2.6: Commitment Functionality, $\mathcal{F}_{\mathsf{Commit}}$.

Protocol Π_{Commit}

This protocol is realized in the $\mathcal{F}_{\mathsf{Broadcast}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.

Initialize The parties do the following:

1. Agree on a set X (elements of which are to be committed), computational security parameter $\kappa \in \mathbb{N}$, and a session identifier sid.
2. Call an instance of $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, \{0,1\}^{2\cdot\kappa}, sid)$.
3. Call an instance of $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Initialize}, sid)$.

Commit For $\mathcal{P}_i$ to commit to an input x, parties do the following:

1. Party $\mathcal{P}_i$ samples $r \leftarrow \mathcal{U}(\{0,1\}^{\kappa})$.
2. Party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{RO}}$ with input $(x \| r \| i, sid)$, where $\|$ denotes concatenation of strings, and receives an output $\tau_x \in \{0,1\}^{2\cdot\kappa}$.
3. Party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Broadcast}, \tau_x, sid)$.

Open To open the value with identifier id_x, parties do the following:

1. Party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Broadcast}, x \| r \| i, sid)$. Let m^j be the message received by $\mathcal{P}_j$.
2. Each party $\mathcal{P}_j$ calls $\mathcal{F}_{\mathsf{RO}}$ with input (m^j, sid) and receives τ_x^j in response.
3. The parties call $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Verify}, sid)$; if $\mathcal{F}_{\mathsf{Broadcast}}$ returns the message OK then they continue; otherwise they (locally) output $\perp$, and then halt.
4. Each party checks that $\tau_x^j = \tau_x$ and if so then it extracts x from $m^j = x \| r$, outputs x, and continues; otherwise, it calls $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid)$, (locally) outputs $\perp$, and then halts.

Figure 2.7: Commitment Protocol, Π_{Commit}.

Theorem 2.3. *The protocol Π_{Commit} UC-securely realizes the functionality $\mathcal{F}_{\mathsf{Commit}}$ against a static, active, computationally-bounded adversary in the $\mathcal{F}_{\mathsf{Broadcast}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.*

Proof. If the adversary behaves honestly, then since the protocol is realized in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model, the simulator can extract any secrets to which corrupt parties commit and forward these to $\mathcal{F}_{\mathsf{Commit}}$.

When an honest party commits to a secret x in the ideal world, the simulator receives some id_x. The simulator then samples some $\tau_x \leftarrow \mathcal{U}\left(\{0,1\}^{2\cdot\kappa}\right)$ instead of calling the local instance of the random oracle and sends this to $\mathcal{A}$ emulating the call to $\mathcal{F}_{\mathsf{Broadcast}}$. Now when this commitment is opened, first $\mathcal{S}$ awaits the revealed secret x from $\mathcal{F}_{\mathsf{Commit}}$, and then programs the random oracle by sampling some $r \leftarrow \mathcal{U}(\{0,1\}^{\kappa})$ and fixing $\mathtt{DB}[x\|r\|i] := \tau_x$.[3] Then to open this secret to $\mathcal{A}$, the simulator sends $x\|r\|i$ to $\mathcal{A}$, emulating the call to $\mathcal{F}_{\mathsf{Broadcast}}$. Thus when the adversary queries the random oracle on $x\|r\|i$, it will receive τ_x.

Now the only problem occurs if any r that was sampled by the simulator when emulating honest parties' commitments was part of a message already queried by $\mathcal{A}$. However, the adversary is computationally-bounded so the number of queries is bounded by some polynomial function in κ, and additionally the number of commitments emulated on behalf of honest parties is bounded by a polynomial function in κ. Let $\mathrm{poly}(\kappa)$ be the total number of queries to the random oracle in the execution. Then the probability that this set contains collisions can be approximated as $\mathrm{poly}(\kappa)^2 \cdot (2 \cdot 2^{2\cdot\kappa})^{-1}$ by the Birthday Bound. (The Birthday Bound [Sch96] says that if q is the total number of queries and S is the codomain of $\mathcal{F}_{\mathsf{RO}}$, then the probability of a collision is approximately $q^2 \cdot (2 \cdot |S|)^{-1}$.) By definition, a protocol secure against a computationally-bounded adversary with parameter κ means that $\mathcal{A}$ cannot perform 2^{κ} operations. Thus $\frac{1}{2} \cdot \mathrm{poly}(\kappa)^2 < 2^{\kappa}$ and hence $\mathrm{poly}(\kappa)^2 \cdot (2 \cdot 2^{2\cdot\kappa})^{-1} < 2^{-\kappa}$.

Note that if the adversary does not call $\mathcal{F}_{\mathsf{RO}}$ before calling $\mathcal{F}_{\mathsf{Broadcast}}$ on some input $x\|r\|i$, then the message it chooses to broadcast, τ_x, is rejected by honest parties (and they abort) unless the value that $\mathcal{F}_{\mathsf{RO}}$ samples when handed the input $x\|r\|i$ later (when the adversary reveals the secret) happens to be τ_x. Since τ_x is sampled from a set of size $2^{2\cdot\kappa}$, the chance that this happens is $2^{-2\cdot\kappa}$.

Thus there is no environment that can distinguish between the real and ideal executions except with negligible advantage over guessing. □

[3] The simulator could sample until some r that has not been used before is obtained; however, the analysis of the chance that collisions occur is easier if the sampling is always uniform here and makes no difference to the final result.

2.3.6 Coin-Flipping

The functionality $\mathcal{F}_{\mathsf{CoinFlip}}$ is given in Figure 8.5. The notion of a "secure coin-flipping" functionality is an idea that originates from the '80s at the latest, e.g. [Blu81]. The idea is for a set of parties to obtain random strings of length at least one such that the sampled strings follow a uniform distribution. A protocol Π_{CoinFlip} realizing $\mathcal{F}_{\mathsf{CoinFlip}}$ in the $\mathcal{F}_{\mathsf{Commit}}$-hybrid model is given in Figure 2.9.

Functionality $\mathcal{F}_{\mathsf{CoinFlip}}$

Initialize On input $(\texttt{Initialize}, X, sid)$, await further messages.

Random Element On input $(\texttt{RElt}, sid)$ from all honest parties and $\mathcal{S}$, sample $x \leftarrow \mathcal{U}(X)$, send x to $\mathcal{S}$, and await a message OK or Abort from $\mathcal{S}$. If the message is OK then send x to all parties and continue; otherwise, send the message Abort to all honest parties and halt.

Figure 2.8: Coin-Flipping Functionality, $\mathcal{F}_{\mathsf{CoinFlip}}$.

Protocol Π_{CoinFlip}

This protocol is realized in the $\mathcal{F}_{\mathsf{Commit}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.

Initialize The parties do the following:

1. Agree on a set X from which to sample, a session identifier sid and a computational security parameter κ.
2. Call an instance of $\mathcal{F}_{\mathsf{Commit}}$ with input $(\texttt{Initialize}, \{0,1\}^{2\cdot\kappa}, sid)$.
3. Call an instance of $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, X, sid)$.

Random Element To obtain a random element sampled uniformly from some set X, party $\mathcal{P}_i$ does the following:

1. Sample $str^i \leftarrow \mathcal{U}\left(\{0,1\}^{2\cdot\kappa}\right)$.
2. Call $\mathcal{F}_{\mathsf{Commit}}$ with input $(\texttt{Commit}, i, str^i, sid)$ and $(\texttt{Commit}, j, \perp, sid)$ for all $j \in [n] \setminus \{i\}$.
3. Await the identifiers id_{str^j} for all $j \in [n]$ from $\mathcal{F}_{\mathsf{Commit}}$.
4. Call $\mathcal{F}_{\mathsf{Commit}}$ with input $(\texttt{Open}, j, id_{str^j}, sid)$ for all $j \in [n]$ and await the set $\{str^j\}_{j\in[n]}$ from $\mathcal{F}_{\mathsf{Commit}}$. If $\mathcal{F}_{\mathsf{Commit}}$ sends the message Abort, then (locally) output $\perp$ and then halt; otherwise, continue.
5. Call $\mathcal{F}_{\mathsf{RO}}$ with input $\left(\left(\bigoplus_{j=1}^{n} str^j\right), sid\right)$ and (locally) output the returned value x.

Figure 2.9: Coin-Flipping Protocol, Π_{CoinFlip}.

Theorem 2.4. *The protocol Π_{CoinFlip} UC-securely realizes $\mathcal{F}_{\mathsf{CoinFlip}}$ in the presence of a static, active, computationally-bounded adversary in the $\mathcal{F}_{\mathsf{Commit}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.*

Proof. The simulator runs local instances of $\mathcal{F}_{\mathsf{Commit}}$ and $\mathcal{F}_{\mathsf{RO}}$. When $\mathcal{F}_{\mathsf{CoinFlip}}$ outputs a value x to honest parties and $\mathcal{S}$, the simulator programs the random oracle $\mathcal{F}_{\mathsf{RO}}$ to store $\mathtt{DB}[\bigoplus_{i=1}^{n} str^i] := x$ so that when $\mathcal{A}$ calls $\mathcal{F}_{\mathsf{RO}}$ with input $\bigoplus_{i=1}^{n} str^i$ the simulator will return x.

Note that as there is always at least one honest party, the string in the real world is uniformly-distributed, and as $\mathcal{S}$ honestly emulates the behaviour of honest parties, it is also uniform in the simulation.

Now, as in the proof of Theorem 2.3, a problem with the simulation only occurs if $\mathcal{A}$ queries $\mathcal{F}_{\mathsf{RO}}$ before $\mathcal{S}$ can program it with an output from $\mathcal{F}_{\mathsf{CoinFlip}}$. Since $\mathcal{A}$ is computationally-bounded, it is able to query $\mathcal{F}_{\mathsf{RO}}$ at most a number of times that is polynomial in κ, poly(κ); since the seed string is $2 \cdot \kappa$ bits long and is always uniformly random, the chance of this type of collision is at most $\text{poly}(\kappa)^2 \cdot (2 \cdot 2^{2 \cdot \kappa})^{-1}$ by the Birthday Bound. As for the proof of Π_{Commit}, $\frac{1}{2} \cdot \text{poly}(\kappa)^2 < 2^{\kappa}$, so $\text{poly}(\kappa)^2 \cdot (2 \cdot 2^{2 \cdot \kappa})^{-1} < 2^{-\kappa}$. Thus no environment can distinguish except with negligible advantage over guessing. □

2.4 Secret Sharing

2.4.1 Access Structures

Two main aspects of corruption in MPC are the *number* of corrupted parties and the *type* of corruption. Protocols usually tolerate some fixed corruption threshold, or more generally define which sets of parties can be corrupted in a so-called *access structure*. Different types of corruption will be discussed in Section 2.5.1: for now the focus is on which sets of parties can be corrupted.

Access structures determine which parties or sets of parties are allowed to learn specific information in a given protocol. Every secret in every protocol has an access structure associated with it, explicitly or implicitly. Formally, given a set of parties $\mathcal{P}$ indexed by a set $[n]$, an access structure on those parties is pair of subsets of $2^{\mathcal{P}}$, denoted by Γ, called the qualified sets, and Δ, called the unqualified sets. The access structure is called *monotone* if the superset of any set in Γ is also qualified, and the subset of any set in Δ is unqualified. When the specific set does not matter, a qualified set will be denoted by $\mathcal{Q}$, and an unqualified set by $\mathcal{U}$. An access structure is called *complete* if $\Gamma \cup \Delta = 2^{\mathcal{P}}$. With the monotonicity property, this means that they form a partition of $2^{\mathcal{P}}$. Consequently, complete monotone access structures can be completely specified by the maximal sets Δ^+ of Δ (where maximal here means maximal with respect to the

subset relation). These sets are called the maximally-unqualified sets. Similarly there is a subset Γ^- of Γ of minimally-qualified sets.

In general it is possible to consider "partial" access structures in which some sets of unqualified parties may learn statistically-negligible but non-zero information about the secret from the shares they hold. However, in this thesis attention is restricted to complete monotone access structures and *perfect* secret-sharing schemes realizing them, in which unqualified sets of parties have no information about the secret, and qualified sets always learn the secret.

The term *access structure* refers to the set Γ since this set describes which sets of parties should have access to secrets; in the literature, sometimes the term *adversary structure* is used for the same, although this, strictly speaking, refers to Δ, but usually complete access structures are considered so there is no ambiguity.

To be concise, the access structure will be written in terms of party indices $[n]$ rather than the parties themselves, $\mathcal{P}$. For example, $\Gamma^- = \{\{1,2\},\{1,3\},\{2,3\}\}$ describes an access structure on a set of parties $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$. However, a set of parties $\mathcal{Q}$ may also be said to lie in Γ, i.e. $\mathcal{Q} \in \Gamma$ (rather than the indices) where it is clear from context.

Definition 2.10 ((n,t)-Threshold Access Structure)**.** An (n,t)-threshold access structure is defined on n parties as an access structure in which any set of t parties or fewer is unqualified and any set of $t+1$ or more is qualified.

The example given above is the $(3,1)$-threshold access structure. For an (n,t)-threshold access structure, $|\Gamma^-| = \binom{n}{t+1}$ and $|\Delta^+| = \binom{n}{t}$. Note that in the literature t is sometimes defined to be the least threshold for reconstruction (so the above definition would be considered an $(n,t+1)$-threshold access structure). An $(n,n-1)$-threshold access structure is sometimes described as *full-threshold* since only the collaboration of the full set of parties enables determining the secret. An (n,t)-threshold access structure in which $t \leq \lfloor \frac{n-1}{2} \rfloor$ is called an *honest majority* access structure, and one in which $t \leq \lfloor \frac{n-1}{3} \rfloor$ an *honest supermajority*.

A protocol is said to *realize* or *respect* an access structure if it satisfies the required security definitions even when the adversary corrupts any unqualified set of parties.

Types of Access Structure

The predicate $\mathcal{Q}_\ell$ is defined in the following way.

Definition 2.11 ($\mathcal{Q}_\ell$)**.** An access structure (Γ, Δ) satisfies the predicate $\mathcal{Q}_\ell$ if for every set $\{U_i\}_{i\in I} \in 2^\Delta$ where $|I| \le \ell$ it holds that $\bigcup_{i\in I} U_i \subsetneq [n]$.

An access structure satisfying $\mathcal{Q}_\ell$ is said to "be $\mathcal{Q}_\ell$". Note that for any $\ell \in \mathbb{N}$ an $(n, \lfloor \frac{n-1}{\ell} \rfloor)$-threshold access structure is $\mathcal{Q}_\ell$. Thus this notion is a generalization of threshold access structures, and in particular, one can think of $\mathcal{Q}_2$ and $\mathcal{Q}_3$ as generalizations of *honest majority* and *honest supermajority*, respectively.

Example 2.1. Let (Γ, Δ) be an access structure on 4 parties, where

$$\text{Minimally-qualified sets: } \Gamma^- := \{\{1,2\},\{1,3\},\{1,4\},\{2,3,4\}\}$$
$$\text{Maximally-unqualified sets: } \Delta^+ := \{\{1\},\{2,3\},\{2,4\},\{3,4\}\}.$$

One can check that it is complete and monotone. Moreover, (Γ, Δ) is $\mathcal{Q}_2$.

Definition 2.12 (Dual Access Structure)**.** Given an access structure Γ, the *dual* access structure is defined as $\Gamma^* := \{\mathcal{Q} \subseteq \mathcal{P} : \mathcal{P} \setminus \mathcal{Q} \notin \Gamma\}$.

For example, the dual of an (n,t)-threshold access structure is an $(n, n-t-1)$-threshold access structure. For any $\mathcal{Q}_2$ access structure it holds that $\Gamma^* \subseteq \Gamma$, since for every $\mathcal{Q} \in \Gamma^*$, by definition $(\mathcal{P} \setminus \mathcal{Q}) \notin \Gamma$, which means $\mathcal{Q} \in \Gamma$ (otherwise $\mathcal{Q}$ and $(\mathcal{P} \setminus \mathcal{Q})$ are both in Δ and $\mathcal{Q} \cup (\mathcal{P} \setminus \mathcal{Q}) = \mathcal{P}$, contradicting the fact that Γ is $\mathcal{Q}_2$).

2.4.2 Access Structures to Secret-Sharing

The goal of this section is to show how to share a secret amongst a set of parties under any access structure. The first step is to interpret the access structure as a monotone Boolean function. Such a function can be computed using a monotone span program, and it was shown by Karchmer and Wigdersen [KW93] that MSPs are in bijection with linear secret-sharing schemes, which are more well known.

Access Structures to Boolean Functions

Access structures are in one-to-one correspondence with monotone Boolean functions, whose definition is given here.

Definition 2.13 (Monotone Boolean Function)**.** Define the relation $\prec$ on the set $\{0,1\}^n$ by the following: for any $\mathbf{s}, \mathbf{s}' \in \{0,1\}^n$, $\mathbf{s} \prec \mathbf{s}'$ if and only if $\mathbf{s}_i \le \mathbf{s}'_i$ for every $i \in [n]$. Then

the Boolean function $f : \{0,1\}^n \to \{0,1\}$ with arity n is called *monotone* if for every $\mathbf{s}, \mathbf{s}' \in \{0,1\}^n$, it holds that

$$\mathbf{s} \prec \mathbf{s}' \implies f(\mathbf{s}) \leq f(\mathbf{s}').$$

The relation $\prec$ only induces a partial ordering on $\{0,1\}^n$ and there is no assumption regarding the images of incomparable elements of the domain.

The bijection between monotone Boolean functions and complete monotone access structures is immediate by associating every coordinate of the domain with a party index: consider the standard bijective map encoding subsets as Boolean strings, $g : 2^{[n]} \to \{0,1\}^n$ defined as $g : S \mapsto \mathbf{s}$ where $\mathbf{s}_i = 1 \iff i \in S$; then let

$$\Gamma := \{S \in 2^{[n]} : f(g(S)) = 1\}$$
$$\Delta := \{S \in 2^{[n]} : f(g(S)) = 0\}.$$

Then (Γ, Δ) is a complete monotone access structure on parties indexed by $[n]$.

Boolean Functions to MSPs

Monotone span programs (MSPs) were introduced by Karchmer and Wigderson [KW93] as a model of computation for computing monotone Boolean functions.

Definition 2.14 (Monotone Span Program)**.** An MSP is a quadruple $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$ where $\mathbb{F}$ is a field, $M \in \mathbb{F}^{m \times d}$ is an $m \times d$ matrix over $\mathbb{F}$ with rank d, $\mathbf{t} \in \mathbb{F}^d$ is a non-zero vector, and $\rho : [m] \twoheadrightarrow [n]$ is a surjective map.

For $\mathbf{s} \in \{0,1\}^n$, let $M_{\mathbf{s}}$ denote the submatrix of M obtained by taking the rows indexed by the set $\{j \in [m] : \mathbf{s}_{\rho(j)} = 1\}$. An MSP can be used to define a Boolean function $f : \{0,1\}^n \to \{0,1\}$ in the following way: let $\mathbf{s} \in \{0,1\}^n$; then define

$$f(\mathbf{s}) := \begin{cases} 1 \text{ if } \mathbf{t} \in \operatorname{coim}(M_{\mathbf{s}}) \\ 0 \text{ otherwise} \end{cases}.$$

(Recall that for a matrix A, $\operatorname{coim}(A)$ is the row space of the matrix A.) The function f is monotonic: given $\mathbf{s}$ and $\mathbf{s}'$ such that $\mathbf{s} \prec \mathbf{s}'$, $\operatorname{coim}(M_{\mathbf{s}}) \subseteq \operatorname{coim}(M_{\mathbf{s}'})$, so $\mathbf{t} \in \operatorname{coim}(M_{\mathbf{s}}) \implies \mathbf{t} \in \operatorname{coim}(M_{\mathbf{s}'})$. A vector $\mathbf{s}$ for which $f(\mathbf{s}) = 1$ is said to be *accepted* by the MSP, and otherwise is said to be *rejected*.

The map ρ is called the *row map* and $\mathbf{t}$ the *target vector*. In the literature, it is not always specified that the matrix should have full column rank; however, Beimel et al. [BGP95] showed that performing column operations does not change the Boolean

function that the MSP computes, so this may be assumed without loss of generality. Any vector witnessing that $\mathbf{t} \in \mathrm{coim}(M_{\mathbf{s}})$, i.e. a vector $\boldsymbol{\lambda} \in \mathbb{F}^m$ satisfying ${M_{\mathbf{s}}}^\top \cdot \boldsymbol{\lambda} = \mathbf{t}$, is called a *recombination vector*. If $\mathrm{coker}(M_{\mathbf{s}})$ is non-zero then there are multiple recombination vectors for $\mathbf{s}$.

MSPs to Secret-Sharing

Given a Boolean function that corresponds to some access structure, one can find an MSP that computes it; then, a secret-sharing scheme can be defined in the following way. To share a secret x, the dealer samples $\mathbf{x} \leftarrow \mathcal{U}\left(\mathbb{F}^d\right)$ subject to the constraint that $\langle \mathbf{t}, \mathbf{x} \rangle = x$, constructs a *share vector* $[\![x]\!] := M \cdot \mathbf{x}$, and for each $i \in [n]$ sends $[\![x]\!]_{\mathcal{P}_i} := M_{\mathcal{P}_i} \cdot \mathbf{x}$ to $\mathcal{P}_i$, where $M_{\mathcal{P}_i}$ is the submatrix of rows of M indexed by the set $\{j \in [m] : \rho(j) = i\}$. Here, a distinction is made between subscripts which are sets of *indices* and those which are sets of *parties*. For example, $M_{\{1,2\}}$ denotes the submatrix formed by taking the first two rows of M; however, $M_{\{\mathcal{P}_1, \mathcal{P}_2\}}$ denotes the submatrix indexed by all rows $j \in [m]$ for which $\rho(j) \in \{1,2\}$. Similarly, $[\![x]\!]_1$ denotes the first component of the vector $M \cdot \mathbf{x}$, whereas $[\![x]\!]_{\mathcal{P}_1}$ denotes the vector of shares comprised of the components for which $\rho(j) = 1$. A vector $[\![x]\!]$ is called *qualified* if $\rho(\mathrm{supp}([\![x]\!])) \in \Gamma$ and unqualified otherwise. The notation $[\![\cdot]\!]$ is used to encompass all of the information about the MSP – that is, the field, the matrix, the row map and the target vector.

If a set $\mathcal{Q}$ is qualified then since $\mathbf{t}$ lies in the span of the rows of $M_{\mathcal{Q}}$, there is a linear combination of rows expressed as the recombination vector $\boldsymbol{\lambda} \in \mathbb{F}^m$, such that ${M_{\mathcal{Q}}}^\top \cdot \boldsymbol{\lambda}_{\mathcal{Q}} = \mathbf{t}$; thus the parties in $\mathcal{Q}$ can compute $\langle \boldsymbol{\lambda}_{\mathcal{Q}}, [\![x]\!]_{\mathcal{Q}} \rangle = {\boldsymbol{\lambda}_{\mathcal{Q}}}^\top \cdot M_{\mathcal{Q}} \cdot \mathbf{x} = \mathbf{t}^\top \cdot \mathbf{x} = \langle \mathbf{t}, \mathbf{x} \rangle = x$.

Conversely, if a set $\mathcal{U}$ is unqualified then by definition the vector $\mathbf{t}$ does not lie in the linear span of the rows of $M_{\mathcal{U}}$, i.e. $\mathrm{im}({M_{\mathcal{U}}}^\top)$, so by the Fundamental Theorem of Linear Algebra, it lies in $\ker(M_{\mathcal{U}})$. Thus there is a vector $\mathbf{k} \in \mathbb{F}^d$ satisfying $\langle \mathbf{t}, \mathbf{k} \rangle \neq 0$ and $M_{\mathcal{U}} \cdot \mathbf{k} = \mathbf{0}$; without loss of generality, choose $\mathbf{k}$ such that $\langle \mathbf{t}, \mathbf{k} \rangle = 1$. Now given any share vector $\mathbf{x}$ used to share a secret x, for any $x' \in \mathbb{F}$ define $\mathbf{x}' := \mathbf{x} + (x' - x) \cdot \mathbf{k}$. Then by linearity, $\langle \mathbf{t}, \mathbf{x}' \rangle = x + (x' - x) = x'$, and $[\![x']\!]_{\mathcal{U}} := M_{\mathcal{U}} \cdot \mathbf{x}' = M_{\mathcal{U}} \cdot (\mathbf{x} + (x' - x) \cdot \mathbf{k}) = M_{\mathcal{U}} \cdot \mathbf{x} + (x' - x) \cdot M_{\mathcal{U}} \cdot \mathbf{k} = M_{\mathcal{U}} \cdot \mathbf{x} + (x' - x) \cdot \mathbf{0} = M_{\mathcal{U}} \cdot \mathbf{x} = [\![x]\!]_{\mathcal{U}}$. In words, this says that whatever share vector the unqualified parties hold, every secret in $\mathbb{F}$ is equally as likely, and hence the set of shares held by an unqualified set of parties reveals no information on the secret.

A summary of how an MSP realizes an LSSS is given in Figure 2.10.

Realizing an LSSS from an MSP

Initialize

1. Agree on the access structure Γ and an MSP that realizes it, $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$.
2. For each minimally-qualified set $\mathcal{Q} \in \Gamma^-$, fix $\boldsymbol{\lambda}^{\mathcal{Q}} \in \mathbb{F}^m$ to be any vector such that $\left\{\mathcal{P}_i \in \mathcal{P} : i \in \rho(\mathrm{supp}(\boldsymbol{\lambda}^{\mathcal{Q}}))\right\} \subseteq \mathcal{Q}$ satisfying ${M_{\mathcal{Q}}}^{\top} \boldsymbol{\lambda}^{\mathcal{Q}} = \mathbf{t}$.

Share To share a secret x,

1. Sample $\mathbf{x} \leftarrow \mathcal{U}\left(\{\mathbf{x} \in \mathbb{F}^d : \langle \mathbf{x}, \mathbf{t} \rangle = x\}\right)$.
2. Compute $[\![x]\!] \coloneqq M \cdot \mathbf{x}$.
3. For each $j \in [m]$, send $[\![x]\!]_{\rho(j)}$ to $\mathcal{P}_{\rho(j)}$ over a secure channel.

Reconstruct For a set of parties $\mathcal{Q} \in \Gamma$ to reconstruct, they do the following:

1. Retrieve from memory $\boldsymbol{\lambda}^{\mathcal{Q}'}$ where $\mathcal{Q}' \in \Gamma^-$ is the lexicographically first set satisfying $\mathcal{Q} \supseteq \mathcal{Q}'$,
2. Compute the secret as $x = \langle [\![x]\!]_{\mathcal{Q}'}, \boldsymbol{\lambda}^{\mathcal{Q}'}_{\mathcal{Q}'} \rangle$.

Figure 2.10: Realizing an LSSS from an MSP.

Linearity

Parties can compute any linear function on secrets shared using an LSSS as above by computing the same linear function on their *shares*. To say that the parties "add" shared secrets $[\![x]\!]$ and $[\![y]\!]$ means that for all $i \in [n]$, $\mathcal{P}_i$ computes $[\![x]\!]_{\mathcal{P}_i} + [\![y]\!]_{\mathcal{P}_i}$. Additionally, parties can add a public value a to a secret x that is secret-shared as $[\![x]\!]$ by agreeing at the start of the protocol on some sharing of 1, $[\![1]\!]$, and computing $[\![x]\!] + a \cdot [\![1]\!]$. No secrecy of this sharing need be assumed.

2.4.3 Examples

The sharing and reconstruction methods for the LSSSs used in this thesis are given in this section, along with an example MSP that realizes each type. The initialization step requires the parties to agree on the access structure, and they also agree on a sharing of 1.

Additive Secret-Sharing An additive sharing is denoted by $[\![v]\!]^{\mathtt{A}}$. To share a secret v, the dealer, party $\mathcal{P}_i$, samples $\{[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_j}\}_{j \neq i} \leftarrow \mathcal{U}(\mathbb{F})$, sends $[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_j}$ to $\mathcal{P}_j$ over a secure channel, and fixes $[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i} \coloneqq v - \sum_{\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_j\}} [\![v]\!]^{\mathtt{A}}_{\mathcal{P}_j}$.

Example 2.2. Additive secret-sharing can be represented as an MSP with matrix

$$\begin{array}{c} \mathcal{P}_1 \\ \mathcal{P}_2 \\ \vdots \\ \mathcal{P}_{n-1} \\ \mathcal{P}_n \end{array} \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ -1 & -1 & \cdots & -1 & 1 \end{pmatrix}$$

and $\mathbf{t} = (1, 1, \cdots, 1)^\top \in \mathbb{F}^n$, where the labels to the left of the matrix indicate ρ.

Replicated Secret-Sharing

Ito et al. [ISN93] gave an explicit, constructive method for generating an LSSS for any monotone access structure, called replicated secret-sharing. (It is derived from the conjunctive normal form (CNF) of the access structure and is therefore sometimes called CNF sharing.)

In this scheme, given in Figure 2.11, the secret is split into one share for every set in Δ^+ and every party receives some subset of these shares. It is convenient to talk about the set $\nabla := \{\mathcal{G} \in 2^{\mathcal{P}} : \mathcal{P} \setminus \mathcal{G} \in \Delta^+\}$, that is, the set of complements of sets in Δ^+, rather than Δ^+ itself, because then a party receives a share if and only if it is indexed by some $\mathcal{G} \in \nabla$.

Since each party obtains a set of shares and different parties may receive the same share (if the dealer is honest), the scheme is said to contain *replication*. The notation $x^{\mathrm{R}}_{\mathcal{G}}$ will be used to denote the summand corresponding to set $\mathcal{G} \in \nabla$, x^{R}_{∇} to denote the vector $(x^{\mathrm{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla)$, and $[\![x]\!]^{\mathrm{R}}$ to denote the full vector with replication. Thus the vector of shares held by $\mathcal{P}_i$ is $[\![x]\!]^{\mathrm{R}}_{\mathcal{P}_i} = (x^{\mathrm{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \wedge \mathcal{G} \ni \mathcal{P}_i)$.

Replicated Secret-Sharing

Initialize

1. The parties agree on an access structure, Γ, and compute $\nabla := \{\mathcal{G} \in 2^{\mathcal{P}} : \mathcal{P} \setminus \mathcal{G} \in \Delta^+\}$.
2. The parties agree on a sharing of 1 by agreeing on any $\mathcal{G}^* \in \nabla$, setting $1^{\mathrm{R}}_{\mathcal{G}^*} := 1$ and $1^{\mathrm{R}}_{\mathcal{G}} := 0$ for all $\mathcal{G} \in \nabla \setminus \{\mathcal{G}^*\}$. Set this sharing to be $[\![1]\!]^{\mathrm{R}}$.

Share For party $\mathcal{P}_i$ to share a secret x, it does the following:

1. Sample a set $\{x^{\mathrm{R}}_{\mathcal{G}}\}_{\mathcal{G} \in \nabla} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{\mathcal{G} \in \nabla} x^{\mathrm{R}}_{\mathcal{G}} = x$.
2. For every $\mathcal{G} \in \nabla$, for every $\mathcal{P}_j \in \mathcal{G}$, send $x^{\mathrm{R}}_{\mathcal{G}}$ to $\mathcal{P}_j$ over a secure channel.
3. Each party $\mathcal{P}_j$ concatenates received shares $\{x^{\mathrm{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \wedge \mathcal{G} \ni \mathcal{P}_j\}$ into a share vector $[\![x]\!]^{\mathrm{R}}_{\mathcal{P}_j}$.

Replicated Secret-Sharing (continued)

Reconstruct For a qualified set of parties $\mathcal{Q}$ to learn a secret x amongst themselves,

1. For every $\mathcal{G} \in \nabla$, for any $\mathcal{P}_i \in \mathcal{Q} \cap \mathcal{G}$, for every $\mathcal{P}_j \in \mathcal{Q} \setminus \mathcal{G}$, $\mathcal{P}_i$ sends $x^{\mathrm{R}}_{\mathcal{G}}$ to $\mathcal{P}_j$ over a secure channel.
2. Each party in $\mathcal{Q}$ computes $x = \sum_{\mathcal{G} \in \nabla} x^{\mathrm{R}}_{\mathcal{G}}$.

Figure 2.11: Replicated Secret-Sharing.

Replicated secret-sharing is perfect: an unqualified set of parties is contained in at least one maximally-unqualified set, and is therefore missing at least one share. Since the shares are uniform subject to the constraint that they sum to the secret, this set of parties has no information on the secret. Conversely, any qualified set of parties $\mathcal{Q}$ is not contained in any maximally unqualified set, so for each maximally unqualified set $\mathcal{U}$ there is at least one party $\mathcal{P}_i$ that is in $\mathcal{Q}$ and not in $\mathcal{U}$ (otherwise every party in $\mathcal{Q}$ is also in $\mathcal{U}$, so $\mathcal{Q}$ is unqualified). Thus the parties in $\mathcal{Q}$ collectively hold all shares for all secrets.

DNF Secret-Sharing

Disjunctive normal form (DNF)-based secret-sharing, first given by [ISN87], is given in Figure 2.12.

DNF Secret-Sharing

Initialize

1. The parties agree on an access structure, Γ.
2. For each $\mathcal{Q} \in \Gamma^-$, order the parties in $\mathcal{Q}$ by their party index and set $1^{\mathcal{Q}}_{\mathcal{P}_i} := 1$ where $\mathcal{P}_i$ is the first party in set $\mathcal{Q}$ and $1^{\mathcal{Q}}_{\mathcal{P}_j} := 0$ for all j where $\mathcal{P}_j \in \mathcal{Q}$ and $j > i$. Set this sharing to be $[\![1]\!]^{\mathtt{DNF}}$.

Share For party $\mathcal{P}_i$ to share a secret x, it does the following:

1. For each $\mathcal{Q} \in \Gamma^-$, sample $\{x^{\mathcal{Q}}_{\mathcal{P}_j}\}_{\mathcal{P}_j \in \mathcal{Q}} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{\mathcal{P}_j \in \mathcal{Q}} x^{\mathcal{Q}}_{\mathcal{P}_j} = x$.
2. For each $\mathcal{Q} \in \Gamma^-$, for each $\mathcal{P}_j \in \mathcal{Q}$, send $x^{\mathcal{Q}}_{\mathcal{P}_j}$ to $\mathcal{P}_j$ over a secure channel.

Reconstruct For a qualified set of parties $\mathcal{Q}$ to open a secret x,

1. A qualified set $\mathcal{Q}$ of parties has all shares in the set $\{x^{\mathcal{Q}}_{\mathcal{P}_i}\}_{\mathcal{P}_i \in \mathcal{Q}}$. For every $\mathcal{P}_i \in \mathcal{Q}$, for every $\mathcal{P}_j \in \mathcal{Q} \setminus \mathcal{P}_i$, $\mathcal{P}_i$ sends $x^{\mathcal{Q}}_{\mathcal{P}_i}$ to $\mathcal{P}_j$ over a secure channel.
2. Each party in $\mathcal{Q}$ computes $x = \sum_{\mathcal{P}_i \in \mathcal{Q}} x^{\mathcal{Q}}_{\mathcal{P}_i}$.

Figure 2.12: DNF Secret-Sharing.

Shamir's Secret-Sharing

Shamir's secret-sharing scheme, developed independently by Blakley [Bla79] and Shamir [Sha79], is given in Figure 2.13.

Shamir's Secret-Sharing

Initialize

1. The parties agree on an (n,t)-threshold access structure.
2. The parties assign one distinct element of $\mathbb{F}$ to each party and one for the secret. For simplicity, usually $\mathcal{P}_i$ is assigned the value $i \in \mathbb{F}$ and the secret is assigned 0.
3. The parties agree on a sharing of 1 defined using the constant polynomial $u(X) := 1$.

Share For party $\mathcal{P}_i$ to share a secret x, it does the following:

1. Sample a polynomial $f \leftarrow \mathcal{U}(\mathbb{F}[X])$ of degree at most t subject to $f(0) = x$.
2. For each $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$, send $f(i)$ to $\mathcal{P}_j$ over a secure channel.

Reconstruct For a qualified set of parties $\mathcal{Q}$ of size at least $t+1$ to open a secret x shared via polynomial f,

1. Each party $\mathcal{P}_i \in \mathcal{Q}$ sends their share $f(i)$ to all other parties in $\mathcal{Q}$ over authenticated channels.
2. Each party in $\mathcal{Q}$ interpolates a polynomial f of degree at most t through the points $\{(i, f(i)) : \mathcal{P}_i \in \mathcal{Q}\}$ and computes $x := f(0)$.

Figure 2.13: Shamir's Secret-Sharing.

Example 2.3. Shamir's secret-sharing for an (n,t)-threshold access structure is given by

$$\begin{matrix}\mathcal{P}_1 \\ \mathcal{P}_2 \\ \vdots \\ \mathcal{P}_n\end{matrix}\begin{pmatrix} 1^0 & 1^1 & \cdots & 1^t \\ 2^0 & 2^1 & \cdots & 2^t \\ \vdots & \vdots & \ddots & \vdots \\ n^0 & n^1 & \cdots & n^t \end{pmatrix}$$

i.e. a Vandermonde matrix, where the parties on the left of the matrix indicate the row map, and $\mathbf{t} = (1,0,\cdots,0)^\top$. In this example, when sharing secret s, the vector $\mathbf{x}$ corresponds exactly to the vector of coefficients of the randomly-sampled polynomial.

2.4.4 Multiplicativity

Some of the protocols in this thesis rely on a property of certain LSSSs called *multiplicativity*. If an access structure is $\mathcal{Q}_2$ then there exists an LSSS for which the parties can compute an additive secret-sharing of the product of two secrets (shared in the LSSS)

by *local* computations. In fact, one can derive a new MSP that realizes a new access structure. This MSP will be denoted by $[\![\cdot]\!]^{[2]}$ and will be called the *product* MSP (or LSSS).

In order to determine how to combine shares to obtain an additive sharing, one can use the following method. Recall the definition of the tensor product, $\otimes : \mathbb{F}^{d_1 \times d_2} \times \mathbb{F}^{e_1 \times e_2} \to \mathbb{F}^{d_1 e_1 \times d_2 e_2}$, where for $A = (a_{i,j})$ and $B = (b_{i,j})$,

$$A \otimes B \mapsto \begin{pmatrix} a_{1,1}B & \cdots & a_{1,d_2}B \\ \vdots & \ddots & \vdots \\ a_{d_1,1}B & \cdots & a_{d_1,d_2}B \end{pmatrix}$$

$$= \begin{pmatrix} a_{1,1}\begin{pmatrix} b_{1,1} & \cdots & b_{1,e_2} \\ \vdots & \ddots & \vdots \\ b_{e_1,1} & \cdots & b_{e_1,e_2} \end{pmatrix} & \cdots & a_{1,d_2}\begin{pmatrix} b_{1,1} & \cdots & b_{1,e_2} \\ \vdots & \ddots & \vdots \\ b_{e_1,1} & \cdots & b_{e_1,e_2} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ a_{d_1,1}\begin{pmatrix} b_{1,1} & \cdots & b_{1,e_2} \\ \vdots & \ddots & \vdots \\ b_{e_1,1} & \cdots & b_{e_1,e_2} \end{pmatrix} & \cdots & a_{d_1,d_2}\begin{pmatrix} b_{1,1} & \cdots & b_{1,e_2} \\ \vdots & \ddots & \vdots \\ b_{e_1,1} & \cdots & b_{e_1,e_2} \end{pmatrix} \end{pmatrix}$$

Now define

$$M^{[2]} := \left(\begin{array}{c} M_{\mathcal{P}_1} \otimes M_{\mathcal{P}_1} \\ \hline \vdots \\ \hline M_{\mathcal{P}_n} \otimes M_{\mathcal{P}_n} \end{array}\right).$$

This matrix has height $m^{[2]} := \sum_{i=1}^{n} |\rho^{-1}(i)|^2$ and width $d^{[2]} := d^2$. The map ρ is assumed monotonic (i.e. the rows of M are grouped according to ownership, and are ordered according to party index). Define an augmented rowmap $\rho^{[2]}$ as:

$$\rho^{[2]}(j) := \rho(i) \text{ for all } j \text{ satisfying } \sum_{k<i} |\rho^{-1}(k)|^2 < j < \sum_{k \le i} |\rho^{-1}(k)|^2.$$

Then the original MSP is said to be *multiplicative* if $\mathbf{t} \otimes \mathbf{t}$ lies in $\mathrm{coim}(M^{[2]})$. In other words, if there exist $(\boldsymbol{\mu}^{(i)})_{i \in [n]}$ such that

$$\langle \boldsymbol{\lambda}, [\![a]\!] \rangle \cdot \langle \boldsymbol{\lambda}, [\![b]\!] \rangle = \sum_{i \in [n]} \langle \boldsymbol{\mu}^{(i)}, [\![a]\!]_{\mathcal{P}_i} \otimes [\![b]\!]_{\mathcal{P}_i} \rangle.$$

The vector $\boldsymbol{\mu} := (\boldsymbol{\mu}^{(i)})_{i \in [n]}$ from above is a recombination vector for the product MSP.

Example 2.4. Replicated secret-sharing is always multiplicative if the access structure is $\mathscr{Q}_2$: given two secrets x and y shared using replicated secret-sharing, for every pair

of shares $(x^{\mathrm{R}}_{\mathcal{G}_1}, y^{\mathrm{R}}_{\mathcal{G}_2})$, there is a party that holds both shares. To see this, recall that a party receives the share indexed by the set $\mathcal{G}$ if and only if it is in $\mathcal{G}$, and observe that if $\mathcal{G}_1 \cap \mathcal{G}_2 = \varnothing$ then since $\mathcal{G}_1 = \mathcal{P} \setminus \mathcal{U}_1$ and $\mathcal{G}_2 = \mathcal{P} \setminus \mathcal{U}_2$ for some $\mathcal{U}_1, \mathcal{U}_2 \in \Delta^+$, it holds that $\varnothing = (\mathcal{P} \setminus \mathcal{U}_1) \cap (\mathcal{P} \setminus \mathcal{U}_2) = \mathcal{P} \setminus (\mathcal{U}_1 \cup \mathcal{U}_2)$; i.e. $\mathcal{U}_1 \cup \mathcal{U}_2 = \mathcal{P}$, violating the $\mathcal{Q}_2$ predicate. Thus every cross and diagonal term can be computed locally, which means parties can compute an additive sharing of the product by (arbitrarily) assigning each mixed term to one party.

Example 2.5. It is easy to see that Shamir's secret-sharing can be used to obtain an additive sharing of the product of ℓ secrets by local computations if it computes a $\mathcal{Q}_\ell$ access structure – that is, an $(n, \lfloor \frac{n-1}{\ell} \rfloor)$-threshold access structure – since each party can locally multiply the ℓ shares they hold to obtain a point on a polynomial of degree at most most $\ell \cdot \lfloor \frac{n-1}{\ell} \rfloor \leq n$. This was noted by Ben-Or et al. [BGW88].

Product Access Structure

One can compute the access structure of the product by inspecting the matrix $M^{[2]}$. Let this access structure be $\Gamma^{[2]}$. It clearly holds that $\Gamma^{[2]} \subseteq \Gamma$ since the product MSP is constructed from the original MSP, so any set of parties learning the product of two secrets also knows the original shares.

2.5 MPC

The goal of MPC is to realize the so-called arithmetic black box introduced by Damgård and Nielsen [DN03], the active variant of which, denoted by $\mathcal{F}_{\mathsf{ABB}}$, is given in Figure 2.14.

The two requirements of MPC protocols are that they must be *correct* and *secure*: that is, the output of the protocol must be correct based on the initial inputs, and the protocol must "be secure", which depends on the definition of the security model. Indeed, security is a nebulous term and is multi-faceted, and often defined on an *ad hoc* basis: for example, protocols may be *robust* – all honest parties obtain the correct output – or just secure "*with abort*" – the honest parties either receive the correct output, or the adversary causes the protocol to abort. The *privacy* (or *secrecy*[4]) of honest parties'

[4]Some consider privacy to be "passive" and secrecy "active" in the sense that the former is general, whereas the latter implies the existence of *particular* information a person wishes to keep hidden.

inputs is almost always required. In this section, an overview of some of the prevailing techniques and the meaning of security and correctness in MPC is given.

Functionality $\mathcal{F}_{\mathsf{ABB}}$

Initialize On input $(\texttt{Initialize}, \mathbb{F}, sid)$ from all parties, initialize a new database DB with indexing set DB.Ids and store the field as $\texttt{DB.Field} := \mathbb{F}$.

Input On input $(\texttt{Input}, i, id, x, sid)$ from party $\mathcal{P}_i$ and $(\texttt{Input}, i, id, \perp, sid)$ from all other parties, where $i \in [n]$, id is a new identifier, and $x \in \texttt{DB.Field}$, set $\texttt{DB}[id] := x$ and insert id into DB.Ids.

Add On input $(\texttt{Add}, id_x, id_y, id_z, sid)$ from all parties, if $id_x, id_y \in \texttt{DB.Ids}$ and id_z is a new identifier, set $\texttt{DB}[id_z] := \texttt{DB}[id_x] + \texttt{DB}[id_y]$ and insert id into DB.Ids

Multiply On command $(\texttt{Multiply}, id_x, id_y, id_z, sid)$ from all parties, if $id_x, id_y \in \texttt{DB.Ids}$ and id_z is a new identifier, store $\texttt{DB}[id_z] := \texttt{DB}[id_x] \cdot \texttt{DB}[id_y]$ and insert id_z into DB.Ids

Output To One On input $(\texttt{Output}, i, id, sid)$ from all parties where $id \in \texttt{DB.Ids}$, if $i \in A$ then send $\texttt{DB}[id]$ to the simulator $\mathcal{S}$. If $\mathcal{S}$ responds with OK then continue and otherwise send the message Abort to all parties, and then halt. Otherwise if $i \in [n] \setminus A$ await a message OK or Abort from $\mathcal{S}$. If the message is OK then send $\texttt{DB}[id]$ to $\mathcal{P}_i$ and continue; otherwise send the message Abort to all parties, and then halt.

Output To All On input $(\texttt{Output}, 0, id, sid)$ from all parties, if $id \in \texttt{DB.Ids}$, send $\texttt{DB}[id]$ to the simulator $\mathcal{S}$. If $\mathcal{S}$ responds with OK then send the value $\texttt{DB}[id]$ to all parties continue, and otherwise send the message Abort to all parties, and then halt.

Figure 2.14: Arithmetic Black Box Functionality, $\mathcal{F}_{\mathsf{ABB}}$.

2.5.1 Correctness

The correctness of a protocol Π is defined in terms of whether or not honest parties receive the same outputs they would in the ideal-world experiment in which they interact with a functionality $\mathcal{F}$ that Π should emulate.

Definition 2.15 (Protocol Correctness)**.** Given a protocol Π realizing an n-party functionality $\mathcal{F}$, for every distinguisher $\mathcal{D}$ that is handed the inputs and outputs of all parties, for every initial input $\mathbf{x} \in \{0,1\}^{n \cdot \mathrm{poly}(\sigma)}$ where $\mathrm{poly} \in \mathbb{Z}[X]$ is a polynomial, it holds that

$$\left| \Pr[\mathcal{D}(\mathbf{x}, \Pi(\mathbf{x})) = 1] - \Pr[\mathcal{D}(\mathbf{x}, \mathcal{F}(\mathbf{x})) = 1] \right| \leq \mathrm{negl}(\sigma)$$

where $\mathrm{negl} : \mathbb{N} \to \mathbb{R}$ is a negligible function and the probability is taken over the random

coins of the honest parties and $\mathcal{A}$. The correctness is said to be *perfect* if negl $\equiv 0$, or statistical otherwise.

In the UC framework, the environment $\mathcal{Z}$ has strictly more information than the distinguisher $\mathcal{D}$ in the above definition, so correctness is subsumed into security.
There are multiple ways of relaxing correctness:

Security with abort The adversary obtains output and can cause honest parties to abort before they receive output.

Security with identifiable abort As above, but the honest parties can identify at least one of the corrupt parties when an abort occurs.

Fairness Either all parties obtain correct output or none of them do.

Guaranteed output delivery/Robustness The adversary cannot prevent honest parties from obtaining output once the inputs have been given.

The focus in this thesis is on providing *security with abort*, primarily because stronger guarantees require heavier machinery (i.e. they typically incur greater computation and communication complexity), leading to less efficient protocols, and because in real-world applications it is considered a "good enough" level of security.

Adversary Types

Monolithic The corrupt parties operate under a global strategy defined by a PPT Turing machine called the adversary.

Rushing In a given round[5], the adversary receives all honest parties' communication before sending its own.

Passive/Active Corruption of parties can be either *passive* or *active*. A passively-corrupt party follows the protocol honestly but may try to glean information from the communication tapes (i.e. the transcript) about other parties' inputs. An actively-corrupt party may deviate arbitrarily from the protocol. Protocols secure in the presence of active adversaries are said to provide the strongest security guarantees since they require the fewest assumptions on the adversary. Security against passive adversaries is essential for any MPC protocol to satisfy the secrecy property. Passive

[5]See Section 2.3.4 for the definition of *round*.

security is also known as Semi-honest, Honest-but-curious, Eavesdropping, Fault-tolerance, and active as Malicious and Byzantine.

Static/Adaptive A static adversary corrupts a set of parties at the beginning of the protocol and cannot later corrupt other parties. An adaptive adversary can corrupt parties arbitrarily throughout the protocol.

Computationally-bounded/unbounded The adversary is either assumed to be computationally-bounded, or unbounded. A computationally-bounded adversary's computing power is parameterized by the computational security parameter κ.

Remark 2.1. It is possible for multiple adversaries to be working independently in any given protocol execution, each corrupting different sets of parties. However, any monolithic adversary learns at least as much as any subset of corruptions since it can correlate the information and so in the literature this assumption is almost always made.

Remark 2.2. A rushing adversary can be dealt with trivially by all parties first committing to their message in one round, and then in the next round opening the commitment. Actively-secure protocols often amortize this cost by opening secrets optimistically and deferring correctness checks to a single check at the end of the protocol.

2.5.2 Privacy

At a high level, an MPC protocol is said to achieve privacy if no party can learn anything more about other parties' inputs than what can be deduced from the final output and its own inputs alone. As with correctness, privacy is implied in the UC framework, since if the environment can determine the inputs of honest parties from the protocol transcript, then it can use this to distinguish between simulator's emulated inputs and the real honest parties' inputs.

It is necessary to think carefully about what functions should be permitted when performing MPC. For example, any protocol securely computing the XOR of two secret field elements, one input from each of two parties, will always reveal the other party's secret input. A less trivial example is computing the intersection of two private sets: a malicious party could simply provide as input set to the protocol the whole universe of inputs, and will then learn exactly the set provided as input by the other party. Under the definition of privacy provided it is possible to create protocols to realize these functionalities securely, but they are always susceptible to these attacks unless one is careful to impose restrictions on the parties' inputs.

2.5.3 Main Techniques and Paradigms

Since MPC involves interaction amongst sets of parties, communication efficiency is crucial. Round complexity divides the literature into two broad categories: constant-round protocols that use so-called *garbled circuits*, and variable-round protocols that use an LSSS. In the former, no communication is required to evaluate the circuit: it is only required for the initial "garbling" and for decoding the result. In the latter, parties continually communicate during circuit evaluation until the result is obtained. One advantage in LSSS-based solutions is that they directly allow reactive computation, meaning that parties can compute a circuit, reveal the output, and then compute further on secret data not yet revealed; garbled circuits tend to be non-reactive since they require (highly) function-dependent preprocessing. Generally speaking, garbled circuits are more efficient in a wide-area network (WAN) setting, where latency is the main bottleneck, and LSSS-based MPC is more efficient in a local-area network (LAN), though this is not a hard-and-fast rule. As this thesis deals with LSSS-based MPC protocols until Chapter 8, discussion of garbled circuits (GCs) is deferred until then.

Preprocessing Model: SPDZ Family of Protocols

Perhaps the most common technique employed in MPC protocols to improve their efficiency is to generate preprocessed data that is essentially the desired circuit or function evaluated on *random* inputs, and then to "derandomize" it later using the real inputs. The advantage of doing this is that since the randomized version reveals nothing about the real circuit inputs (indeed, they may not even be known prior to the derandomization step), the parties can perform checks to ensure that the randomized versions are correct with overwhelming probability in σ. This leaves very little room for the adversary to deviate from the protocol when derandomizing the circuit.

Another advantage to this technique is that while actually computing the circuit or function may be expensive – for full-threshold MPC, public-key cryptography (PKC) is needed, for example somewhat-homomorphic encryption (SHE) [BDOZ11, DPSZ12, KPR18] or oblivious transfer (OT) [NNOB12, FKOS15, KOS16] – the derandomization process is generally very cheap and is information-theoretic (IT)[6].

Beaver's Circuit Randomization A common technique amongst LSSS-based MPC protocols is that of circuit randomization. This technique due to Beaver [Bea92] allows

[6]While the evaluation is IT, the generation of Beaver triples may require a computational assumption, in which case the complete protocol only has computational security.

a circuit to be evaluated with IT security assuming the existence of so-called Beaver triples. These are triples of secret-shared values

$$(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket a \cdot b \rrbracket)$$

where usually one denotes $c \coloneqq a \cdot b$. Together with any sharing of 1, denoted by $\llbracket 1 \rrbracket$, two secrets $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$ may be multiplied as follows: the parties reveal the values $\llbracket x - a \rrbracket \coloneqq \llbracket x \rrbracket - \llbracket a \rrbracket$ as a public value r and $\llbracket y - b \rrbracket \coloneqq \llbracket y \rrbracket - \llbracket b \rrbracket$ as a public value s, and compute

$$\llbracket x \cdot y \rrbracket \coloneqq r \cdot s \cdot \llbracket 1 \rrbracket + s \cdot \llbracket a \rrbracket + r \cdot \llbracket b \rrbracket + \llbracket a \cdot b \rrbracket.$$

The security of the protocol comes from the fact that the secrets a and b are uniformly random and therefore essentially one-time-pad-encrypt the inputs when they are opened.

Squaring secrets can be computed using triples, but can also be done with a special type of preprocessing that can be cheaper to generate, given by [DKL+13]. Suppose the parties have a pair $(\llbracket a \rrbracket, \llbracket a^2 \rrbracket)$. Then the parties open $r \coloneqq x - a$ and compute

$$\llbracket x^2 \rrbracket \coloneqq r \cdot (\llbracket x \rrbracket + \llbracket a \rrbracket) + \llbracket a^2 \rrbracket.$$

When making use of Beaver's circuit randomization, circuit evaluation is split into a *preprocessing phase* in which one Beaver triple is generated for each multiplication in the arithmetic circuit, and an *online phase* in which the triple is derandomized as described above.

This circuit randomization trivially extends to any multiplicative depth d by computing all monomials of the product

$$\prod_{i=1}^{d} (\llbracket 1 \rrbracket + \llbracket a_i \rrbracket)$$

as preprocessing, and then derandomizing by broadcasting secrets $\llbracket x_i \rrbracket - \llbracket a_i \rrbracket$ for all $i \in [d]$, where $\llbracket x_i \rrbracket$ are the secrets to be multiplied. However, the length of the tuples increases exponentially in d and generating them becomes impractical.

Authentication In order to be secure in the presence of an active adversary, triples are generated using some form of authentication, which ensures that any additive error introduced on the triple *after its generation* can be detected. Indeed, notice that if Beaver's circuit randomization technique is used, then circuit evaluation on actual circuit inputs involves only opening secrets and then computing linear operations on

secret-shared data. Thus, for active security in the online phase (i.e. when the actual circuit is being evaluated), it suffices to ensure that corrupt parties do not introduce additive errors on secrets.

The precise method of authentication used generally depends on the access structure and secret-sharing scheme being used. Some LSSSs contain a sort of internal redundancy that precludes such tampering of secrets essentially "for free"; this is explored in detail in Chapter 3. Another way to guarantee correctness is to use a verifiable secret-sharing (VSS) scheme, in which secrets are shared "robustly": the honest parties can always reconstruct secrets if the adversary chooses to corrupt shares or to abort.

In full-threshold LSSS-based protocols, authentication is achieved using IT MACs. The SPDZ protocol due to Damgård et al. [DPSZ12] uses SHE to generate Beaver triples with IT MACs; an overview of the MACs now follows. In the SPDZ protocol, additive secret-sharing is used, denoted here by $[\![\cdot]\!]^{\mathsf{A}}$. In addition to holding a secret x as a secret-sharing $[\![x]\!]^{\mathsf{A}}$, parties also hold a sharing of the MAC $[\![\gamma(x)]\!]^{\mathsf{A}} := [\![\alpha \cdot x]\!]^{\mathsf{A}}$ where α is a MAC "key" sampled uniformly at random from the field, held by the parties as another sharing $[\![\alpha]\!]^{\mathsf{A}}$. This MAC is linear, so any linear function on the secret – which can be computed without communication by linearity of the LSSS – can also be evaluated on the shares of the MAC to obtain a sharing of a MAC on the output. The authenticated sharing comprising both the secret and its MAC is denoted by $[\![\cdot]\!] = \left([\![\cdot]\!]^{\mathsf{A}}, [\![\gamma(\cdot)]\!]^{\mathsf{A}}\right)$.

Remark 2.3. It is important to note that to say the parties hold the sharing $[\![\gamma(x)]\!]^{\mathsf{A}}$ is not the same as saying $\mathcal{P}_i$ holds $[\![\alpha]\!]^{\mathsf{A}}_{\mathcal{P}_i} \cdot x$ for all $i \in [n]$ (though this is indeed a valid sharing of the MAC on x): they hold uniformly random field elements $\left\{[\![\alpha \cdot x]\!]^{\mathsf{A}}_{\mathcal{P}_i}\right\}_{i \in [n]}$ subject to the constraint that $\sum_{i \in [n]} [\![\alpha \cdot x]\!]^{\mathsf{A}}_{\mathcal{P}_i} = \alpha \cdot x$.

Recall that for Beaver multiplication, a sharing of 1 is required, which means that the parties must have an *authenticated* sharing of 1, and not just some additive sharing $[\![1]\!]^{\mathsf{A}}$. For SPDZ, since the global MAC key α is additively shared as $[\![\alpha]\!]^{\mathsf{A}}$, an authenticated sharing of 1 can be obtained by $\mathcal{P}_1$ setting $[\![1]\!]_{\mathcal{P}_1} := (1, [\![\alpha]\!]^{\mathsf{A}}_{\mathcal{P}_1})$ and all other parties setting $[\![1]\!]_{\mathcal{P}_i} := (0, [\![\alpha]\!]^{\mathsf{A}}_{\mathcal{P}_i})$.

Another issue is that of how parties provide inputs with authentication. This has a simple solution: in order to obtain a MAC on an actual secret, parties can derandomize a random sharing which already has a MAC: suppose the parties hold $[\![r]\!] := \left([\![r]\!]^{\mathsf{A}}, [\![\alpha \cdot r]\!]^{\mathsf{A}}\right)$ for some uniformly random r unknown to any party; then for party $\mathcal{P}_i$ to input a secret x, the parties send the shares of $[\![r]\!]^{\mathsf{A}}$ (but not the shares of the MAC) to $\mathcal{P}_i$, then $\mathcal{P}_i$ reconstructs r, computes $\varepsilon := x - r$, and broadcasts, and then all parties set $[\![x]\!] := [\![r]\!] +$

$\varepsilon \cdot [\![1]\!]$; i.e. they compute $[\![x]\!]^{\mathtt{A}} := [\![r]\!]^{\mathtt{A}} + \varepsilon \cdot [\![1]\!]^{\mathtt{A}}$ and $[\![\gamma(x)]\!]^{\mathtt{A}} := [\![\gamma(r)]\!]^{\mathtt{A}} + \varepsilon \cdot [\![\alpha]\!]^{\mathtt{A}}$.

The process of verifying MACs is not a costly operation and is performed infrequently. Further detail is given in Section 4.7.

Sacrifice The process of generating triples with authentication as used in SPDZ is not the focus of this work. However, the technique known as *sacrificing* used in [DPSZ12] and other works will be used later on, and so the outline is given here.

Authentication alone does not not ensure that that the relation $c = a \cdot b$ holds for each triple. This verification is performed by "sacrificing" one triple to check the correctness of another: parties execute a coin-flipping protocol to agree on some random field element ρ, open $[\![r]\!] := [\![a]\!] - [\![a']\!]$ and $[\![s]\!] := [\![b]\!] - \rho \cdot [\![b']\!]$, and open

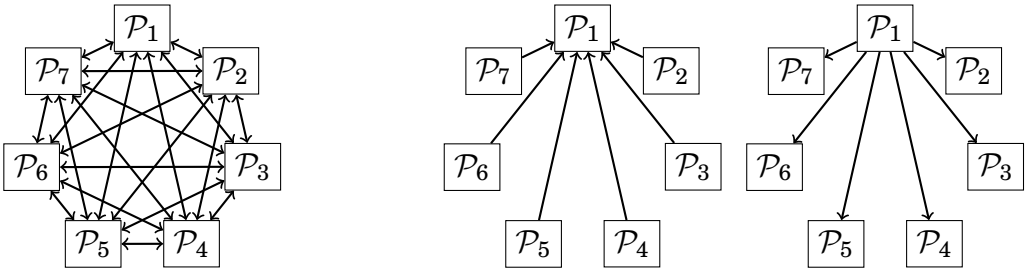
$$[\![t]\!] := r \cdot s \cdot [\![1]\!] + s \cdot [\![a']\!] + r \cdot \rho \cdot [\![b']\!] + \rho \cdot [\![c']\!] - [\![c]\!]$$

and check that $t = 0$. If this check passes then the parties output $([\![a]\!], [\![b]\!], [\![c]\!])$ and discard $([\![a']\!], [\![b']\!], [\![c']\!])$. If the coin-flipping was unbiased then the probability that $c \neq a \cdot b$ but it holds that $t = 0$ is the probability that the adversary sets $c + \varepsilon$ and manages to choose δ to add to c' so that $\rho \cdot \delta - \varepsilon = 0$, which can be done with probability at most $1/|\mathbb{F}|$ by guessing ρ ahead of time and modifying shares accordingly *before* the triples are authenticated.

Techniques have been developed to amortize the sacrifice checks in certain settings: For example, in the honest majority threshold setting, Choudhury and Patra [CP17] give a probabilistic approach to verifying the correctness of Beaver triples shared using Shamir's secret-sharing. However, the cost of generating extra triples and sacrificing them to obtain full active security is generally substantial.

Opening secrets efficiently The derandomization procedure involves "opening" secrets. In many actively-secure MPC protocols, this is *not* the same as all parties broadcasting their share(s) since correctness is not guaranteed until a later point in the protocol execution as an amortized batch check. For example, if MACs are used to authenticate as described above, then multiple secrets can be revealed, and their MACs can be verified at a later point in time. With this perspective, the apparent $O(n^2)$ overhead can be avoided: the opening procedure can be performed in two rounds by opening to a *single* party and having that party broadcast the reconstructed secret, as shown in Figure 2.15.

If such a technique is used, it must be possible for the parties to detect if $\mathcal{P}_1$ decides to change what the broadcast element should be. This opening procedure has been



(a) One round, $O(n^2)$ communication.

(b) Two rounds, $O(n)$ communication.

Figure 2.15: Complexity of Broadcasting.

used in MPC protocols designed to scale well with the number of parties; for example, Damgård and Nielsen [DN07] offered IT protocols for different types of adversaries using this procedure.

Authentication and Sacrifice in smaller fields Recall that the chance that the adversary cheats but the honest parties do not abort depends on the size of the field, as $|\mathbb{F}|^{-1}$. To obtain statistical security $2^{-\sigma}$ for the sacrifice check, $\lceil \sigma / \log |\mathbb{F}| \rceil$ triples must be used to check each triple; for authentication, the parties must hold $\lceil \sigma / \log |\mathbb{F}| \rceil$ MACs on every secret (or, equivalently, one MAC in an extension field of degree $\lceil \sigma / \log |\mathbb{F}| \rceil$).

2.6 Literature Overview

This section contains an overview – which is far from exhaustive – of articles important for understanding the heritage of protocols in this thesis. A timeline is presented in Figure 2.16.

The secret-sharing scheme due to Blakley and Shamir [Sha79, Bla79] based on polynomial interpolation is ubiquitous in the literature of MPC protocols for threshold access structures. Independently and concurrently, they showed how a secret could be split up into n pieces and one piece given to each party so that any coalition of parties of size larger than some fixed threshold t could recover the secret. The linearity of the scheme – that is, that the sum of the shares of secrets is a sharing of the sum of secrets – makes it suitable for use in MPC. One of the benefits of Shamir's scheme over many others is that each party need only hold a single share per secret; the main disadvantage is that it only works for threshold access structures.

Goldreich et al. [GMW87] showed that any function is computable assuming a computationally-bounded adversary corrupting any number of parties. The idea was to take

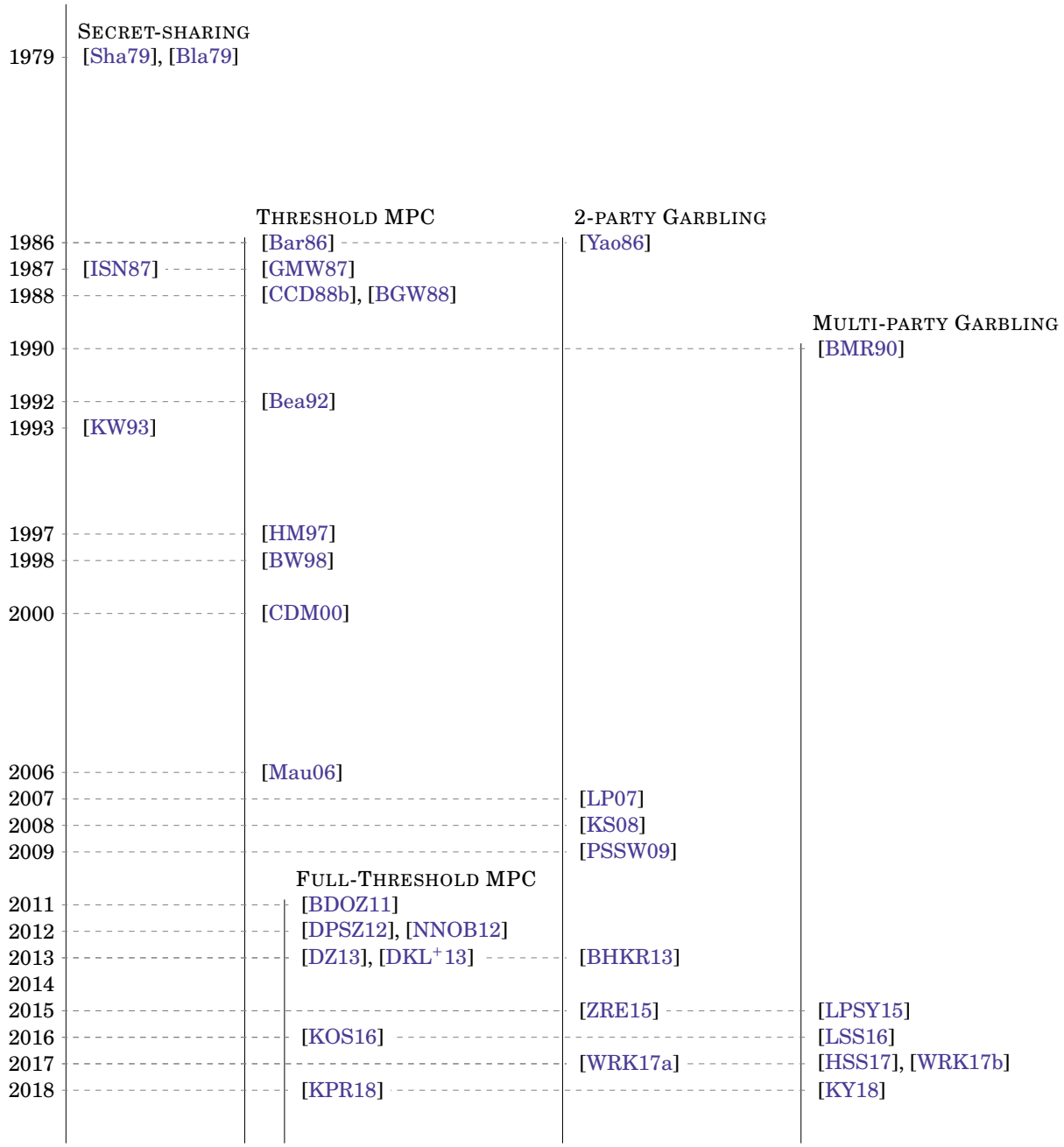


Figure 2.16: Highlights in the Timeline of MPC.

a passive protocol and bootstrap to active security by proving in zero-knowledge[7] that local operations were performed according to the protocol specification. The protocol requires authenticated channels and a broadcast channel.

Shortly after this, Chaum et al. [CCD88b] showed that assuming parties are con-

[7]It suffices to understand this as a type of proof in which a prover proves it knows the witness to an NP statement to a verifier without the verifier learning anything about the witness.

nected by secure channels, then with no cryptographic assumptions parties can compute a function on their combined secret inputs, even if the adversary corrupts up to one third of the total number of parties actively, or up to one half passively. The protocol relies on a procedure known as *cut and choose* to ensure secrets are dealt correctly, which incurs a negligible probability of erroneous output. Concurrently, the protocol of Ben-Or et al. [BGW88], known as the BGW protocol, showed essentially the same result (indeed, exactly the same result in the passive case). However, in the active case, CCD is correct except with negligible probability in σ, whereas BGW uses error-correcting codes to correct any errors. Another distinction is that BGW computes an arithmetic circuit whereas CCD computes a Boolean circuit. An efficient implementation of threshold protocols was given in VIFF [DGKN09].

Following this, Kilian [Kil88] showed that any circuit in NC^1 can be computed by two players using a primitive known as OT. The advantage of this construction over others is that it does not require generic zero-knowledge proofs for NP statements, instead only depending on the existence of OT (which is well-established) and a result due to Barrington [Bar86] that gives a method for constructing a branching program of bounded width and polynomial size for any NC^1 circuit. Two decades later, in the same vein of work founding MPC on OT, Ishai et al. [IPS08] showed how to use OT more efficiently to overlay passively-secure protocols with a "consistency checking" procedure to obtain active security in the full-threshold setting.

Secret-sharing has long been of interest independently of MPC. Ito et al. [ISN87] showed how to construct a LSSS for any access structure. Karchmer and Wigdersen [KW93] gave a mathematical description of secret-sharing schemes in terms of linear spaces and generating matrices, known as MSPs. It was shown by Hirt and Maurer [HM97, HM00] and Beaver and Wool [BW98] that if an access structure is $\mathscr{Q}_2$ then fault-tolerant protocols without cryptographic assumptions exist for these access structures. Maurer [Mau06] showed the same result using replicated secret sharing, where if the access structure is $\mathscr{Q}_2$ then the protocols roughly coincide, and then showed that assuming the access structure is $\mathscr{Q}_3$, by using a VSS scheme a robust protocol can be constructed also using replicated secret-sharing, which is essentially a generalization of the actively-secure $(n, \lfloor \frac{n-1}{3} \rfloor)$-threshold protocols from [CCD88b, BGW88]. Various results in the secret-sharing literature, such as [CDM00] have led to efficiency improvements in LSSS-based MPC.

Beaver [Bea92] showed how to evaluate a circuit on random inputs in such a way that it may be derandomized later inexpensively, which has recently been a boon to

the world of MPC, for example in [BDOZ11, DPSZ12, DKL+13, KOS16, KPR18]. More generally, the idea of using preprocessing has also been used in the context of evaluating Boolean circuits [NNOB12, DZ13, FKOS15].

In a line of work quite independent from LSSS-based MPC, in 1986, Yao [Yao86, Oral presentation] introduced a technique called *circuit garbling* and showed that two parties could use it to evaluate a Boolean circuit securely on their combined input, assuming passive corruption. The protocol requires only a constant number of rounds and uses OT. Later, Beaver et al. [BMR90] showed how to garble circuits in the n-party honest-majority setting assuming secure channels and a broadcast channel. The advantage of the GC approach over the LSSS approach is that a circuit can be garbled in a constant number of rounds (i.e. independently of the multiplicative depth of the circuit, on which LSSS-based MPC protocols typically depend). They also showed how to garble in such a way that the evaluator need only decrypt one ciphertext per gate, instead of all four, in a technique called *point-and-permute*. To select a few highlights in the recent history of circuit garbling: Lindell and Pinkas [LP07] showed how to obtain an efficient 2-party protocol secure against active adversaries; Kolesnikov and Schneider [KS08] showed how to garble XOR gates essentially for free with a stronger assumption on the PRF used for encryption – a technique called FreeXOR; Pinkas et al. [PSSW09] used polynomial interpolation to reduce the number of ciphertexts that need to be sent per gate; Bellare et al. [BHKR13] showed how to use a fixed-key block cipher to generate the ciphertexts; and Zahur et al. [ZRE15] used the asymmetry of the garbler/evaluator dichotomy to halve the communication costs involved in garbling AND gates. More recently, security against active adversaries in the n-party setting [HSS17, WRK17b, KY18] has been a fruitful area of research, where OT has been used extensively.

Chapter 3

Error-detection and Share Reconstruction

This chapter is based on work published at CT-RSA 2019 under the title Error Detection in Monotone Span Programs with Application to Communication-Efficient Multi-party Computation *[SW19] and was joint work with Nigel Smart. The main contribution of that work – error-detection for $\mathscr{Q}_2$ access structures – is discussed here; its application to multi-party computation (MPC) is more the focus of Chapter 7.*

This chapter In this chapter, some important properties of linear secret-sharing schemes (LSSSs) that compute $\mathscr{Q}_2$ access structures are explored – in particular, they offer a form of "free authentication". One can view this as part-way towards a verifiable secret-sharing (VSS) scheme, except that errors can only be detected, not corrected.

The notion of share reconstructability is also defined and explored, which is a property of certain LSSSs and can be viewed as a relaxation of VSS where honest parties can always determine all shares if they collaborate. It differs from VSS as honest parties do not *know* the set of honest parties, so a given honest party does not know which set of shares of other parties to collaborate to determine the remaining shares. LSSSs that are share reconstructable provide optimal communication for the protocols described in the following chapters of this thesis.

3.1 Overview

In the full-threshold SPDZ protocol [DPSZ12] and its successors, e.g. [DKL+13, KOS16], authentication is achieved with additively-homomorphic message authentication codes

(MACs), as was outlined in Section 2.5.3: for each secret that is shared amongst the parties, the parties also share a MAC on that secret. Since the authentication is additively homomorphic and the sharing scheme is linear, the sum (and consequently scalar multiple) of authenticated shares is authenticated "for free" by performing the addition (or scalar multiplication) on the associated MACs. It was shown in Section 2.5.3 that, assuming the parties are provided with authenticated Beaver triples, multiplication is also just a linear operation.

One important branch of the authentication methodology contributing significantly to its practical efficiency is the amortization of verification costs by batch-checking MACs, a technique developed in [BFO12, DPSZ12, DKL+13], amongst other works. The idea is that one can check the correctness of *multiple* linear relations by performing a *single* check on a random linear combination of the data items, which has statistical security if the random coefficients are from a set of size $O(2^\sigma)$.

An orthogonal approach to batch verification for $(3,1)$-threshold access structures was introduced by Furakawa et al. [FLNW17]. There they used redundancy in replicated secret-sharing to reduce verification of multiple secrets to a simple procedure in which parties checked the consistency of (a hash of) their views of the execution.

In the following chapters, full MPC protocols generalizing [FLNW17] are given. These generalizations depend on the work in this chapter, which extends the "view comparison" methodology to any $\mathscr{Q}_2$ access structure on any number of parties. This is achieved by proving a folklore result that roughly says that an LSSS is error-detecting if and only if it is $\mathscr{Q}_2$. While this result was fairly-well understood by the community, its precise formulation and its use in making MPC protocols more efficient was neglected. Indeed, it was noted by Fehr [Feh99] that a share vector is accepted in an monotone span program (MSP) if and only if it is rejected by its dual, but the result in this chapter that shows that the adversary is unable to change share vectors (non-trivially) without detection for $\mathscr{Q}_2$ access structure was not shown, since this fact is of limited interest outside of MPC.

This chapter explores the two instances in which error-detecting can be performed if the access structure is $\mathscr{Q}_2$ (which occur frequently MPC protocols), which occur when secrets are revealed:

- If a single party $\mathcal{P}_i$ is required to learn a secret, a form of error-detection can be done on the shares it receives from other parties.
- If all parties are required to learn a secret, the parties engage in a round of commu-

nication in which not all parties need to communicate with each other. The parties each reconstruct a view of what they think other parties have received, even if they have not communicated with all other parties, and then after opening many secrets, each party hashes the reconstructed view and checks every other party's hash value against their own, and aborts if they differ.

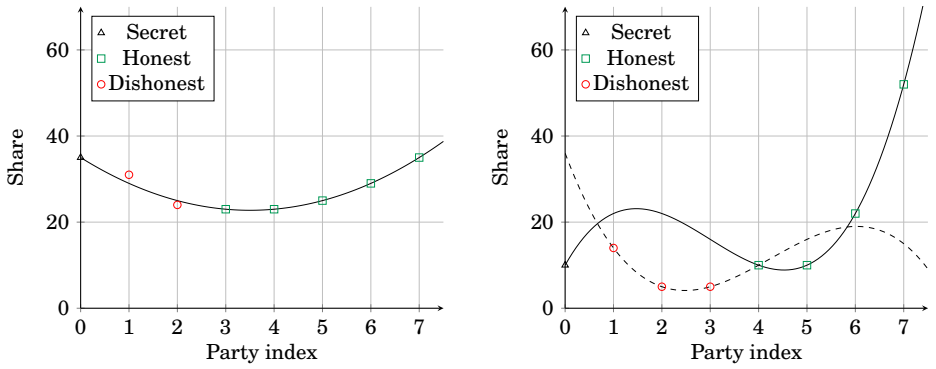
Motivating Example

To motivate a study of error-*detection*, and to aid intuition, consider the following example that uses error-*correction*. Suppose the $(7,2)$-threshold access structure, which is $\mathscr{Q}_3$, is computed using Shamir's secret-sharing scheme. Any 3 parties together hold 3 distinct points on a quadratic, and so they can uniquely identify the polynomial and hence the secret (defined as the polynomial evaluated at 0). If the adversary corrupts any 2 parties (i.e. an unqualified set) and corrupts their shares, then there are 5 correct shares remaining. This means that if all parties broadcast their share, the honest parties can always identify which shares are correct and which are not, since there is only one set of 5 shares all lying on the same quadratic, and hence the erroneous shares may be recomputed and the error thereby corrected. This is shown in Figure 3.1a, where every honest party can determine the correct sharing and hence the correct secret regardless of what corrupt parties broadcast.

This property allows the construction of an MPC protocol with identifiable abort, and hence (via a VSS scheme), robust MPC. In fact, Cramer et al. [CDG+05] showed how the correspondence between LSSSs and linear codes reveals an efficient method by which qualified parties can *correct* any errors in a set of shares for some secret if the access structure is $\mathscr{Q}_3$, since if this holds then a strongly-multiplicative LSSS realizing it allows honest parties to correct any errors introduced by the adversary[1]. This is not a direct connection to error-correction codes since such LSSSs do not necessarily allow unique decoding of the entire share vector: it is only the component of the share vector corresponding to the *secret* that is guaranteed to be correct.

The results presented here are parallel, but for $\mathscr{Q}_2$. If the access structure is $\mathscr{Q}_2$ then any LSSS realizing it allows honest parties to agree on whether or not the secret is correct: thus one obtains a form of error-*detection*. This reveals why the hash-verification procedure enables error-detection and hence security with abort. Again, an example is

[1] In a strongly-multiplicative LSSS, any set of parties whose complement is an unqualified set can compute the product of two secrets. Replicated secret-sharing is strongly multiplicative if the access structure is $\mathscr{Q}_3$.



(a) Error correction.

(b) Error detection.

Figure 3.1: Errors in Secret-Sharing.

instructive: consider the $\mathscr{Q}_2$ access structure $(7,3)$-threshold and Shamir's scheme. Any 4 parties hold 4 distinct points on a cubic, so they can uniquely identify the polynomial and hence the secret. If the adversary corrupts 3 parties and alters their shares, then there are 4 correct shares remaining. Unlike in the above example, *every* 4 shares lie on a cubic (since 4 distinct points determine a unique cubic). If all parties broadcast their shares, the honest parties cannot know which 3 of the other 6 shares it sees are correct. In Figure 3.1b, honest party $\mathcal{P}_4$ cannot decide which polynomial is correct from the broadcasted shares alone. However, $\mathcal{P}_4$ *does* know that if not all 7 shares lie on the same cubic, then some shares must be incorrect: since at least 4 shares are correct (held by honest parties), there is no possible corruption of 3 shares which causes all 7 points to lie on the same cubic but for the cubic to be different from the original equation.

In the example above, the error can be detected since the honest parties' shares determine the cubic, but this is because the only way all 7 points lie on the same cubic is if the shares are correct. In different secret-sharing schemes, it is possible for an unqualified set of shares to be altered so that the resulting share vector is still "valid", but in this section it will be shown that "valid" alterations *cannot* change the encoded secret if the access structure is $\mathscr{Q}_2$.

Dispensing with MACs, the protocol presented here can be instantiated with authentication over small finite fields, or use an LSSS over a ring. In the latter case MSP must be defined over the ring, and it should be noted that while *authentication* is now possible in a ring rather than just a field, the protocols in this thesis are still not (immediately) amenable to computation over arbitrary rings as some procedures (such as the sacrifice step of generating Beaver triples, outlined in Section 2.5.3) require that the ring be an integral domain.

3.2 Opening to One Party

In this section, it will be shown that for an LSSS realizing a $\mathscr{Q}_2$ access structure, if the share vector $[\![x]\!]$ for some secret x is modified with an error vector $\mathbf{e} \in \mathbb{F}^m$ with unqualified support then $[\![x]\!] + \mathbf{e}$ is either no longer a valid share vector (i.e. is not in $\mathrm{im}(M)$), or the error vector encodes 0, and so by linearity $[\![x]\!] + \mathbf{e}$ also encodes x. This immediately gives an efficient method by which a party to whom a secret is opened (by all other parties sending that party all of their shares) can check whether or not the adversary has introduced an error. This process is described at the end of this section. The procedure of opening to a single party is used in MPC protocols for the parties to provide input and obtain output in an actively-secure manner.

In more detail, it will be shown that for any MSP computing any $\mathscr{Q}_2$ access structure, there exists a matrix N such that for any vector $\mathbf{e} \neq \mathbf{0}$ for which $\{\mathcal{P}_i : i \in \rho(\mathrm{supp}(\mathbf{e}))\} \notin \Gamma$, it holds that either $N \cdot \mathbf{e} \neq \mathbf{0}$, or $N \cdot \mathbf{e} = \mathbf{0}$ and $\langle \mathbf{e}, \mathbf{t} \rangle = 0$.

Lemma 3.1. *For any MSP $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$ computing a $\mathscr{Q}_2$ access structure Γ, for any vector $[\![x]\!] \in \mathbb{F}^m$, define $\mathcal{X} := \{\mathcal{P}_i : i \in \rho([\![x]\!])\}$; then*

$$\mathcal{X} \notin \Gamma \implies \begin{cases} [\![x]\!] \notin \mathrm{im}(M), \text{ or} \\ [\![x]\!] \in \mathrm{im}(M) \text{ and } [\![x]\!] = M \cdot \mathbf{x} \text{ for some } \mathbf{x} \in \mathbb{F}^d \text{ where } \langle \mathbf{x}, \mathbf{t} \rangle = 0. \end{cases}$$

Proof. If $\mathcal{X} \notin \Gamma$ then $\mathcal{P} \setminus \mathcal{X} \in \Gamma$ since the access structure is $\mathscr{Q}_2$. Thus there is at least one set $\mathcal{Q} \in \Gamma$ where $\mathcal{Q} \subseteq \mathcal{P} \setminus \mathcal{X}$ for which $[\![x]\!]_{\mathcal{Q}} = \mathbf{0}$, by definition of $\mathcal{X}$. Recall that for a qualified set $\mathcal{Q}$ of parties to reconstruct the secret, they take the appropriate recombination vector $\boldsymbol{\lambda}$ (which has the property that $\{\mathcal{P}_i \in \mathcal{P} : i \in \rho(\mathrm{supp}(\boldsymbol{\lambda}))\} \subseteq \mathcal{Q}$) and compute $s = \langle \boldsymbol{\lambda}, [\![x]\!] \rangle$. For this particular $\mathcal{Q}$ and corresponding recombination vector $\boldsymbol{\lambda}$, it holds that $\langle \boldsymbol{\lambda}, [\![x]\!] \rangle = \langle \boldsymbol{\lambda}_{\mathcal{Q}}, [\![x]\!]_{\mathcal{Q}} \rangle$ and $\langle \boldsymbol{\lambda}_{\mathcal{Q}}, [\![x]\!]_{\mathcal{Q}} \rangle = \langle \boldsymbol{\lambda}_{\mathcal{Q}}, \mathbf{0} \rangle = 0$ by the above, so the secret is 0.

Finally, if $[\![x]\!] \in \mathrm{im}(M)$, then $[\![x]\!] = M \cdot \mathbf{x}$ for some $\mathbf{x} \in \mathbb{F}^d$ where $\langle \mathbf{t}, \mathbf{x} \rangle = 0$, since at least one qualified set, $\mathcal{Q}$, computes the secret as 0. Otherwise, $[\![x]\!] \notin \mathrm{im}(M)$, as claimed. □

The following lemma shows that if the adversary, controlling an unqualified set of parties, adds an error vector $\mathbf{e}$ to a share vector $[\![x]\!]$, the resulting vector $\mathbf{c} := [\![x]\!] + \mathbf{e}$ will either not be a valid share vector, or will encode the same secret as $[\![x]\!]$ (by linearity). Adding in an error $\mathbf{e}$ that does not change the value of the secret can be viewed as the adversary rerandomizing the shares of corrupt parties.

Lemma 3.2. *Let $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$ be an MSP computing $\mathscr{Q}_2$ access structure Γ and $\mathbf{c} := [\![x]\!] + \mathbf{e}$ be the observed set of shares, given as a valid share vector $[\![x]\!]$ encoding secret x,*

with error $\mathbf{e} \in \mathbb{F}^m$. *Then there exists a matrix* N *such that*

$$\{\mathcal{P}_i \in \mathcal{P} : i \in \rho(\mathrm{supp}(\mathbf{e}))\} \notin \Gamma \implies \textit{either } \mathbf{e} \textit{ encodes the error } e = 0, \textit{ or } N \cdot \mathbf{c} \neq \mathbf{0}$$

Proof. Let N be any matrix whose rows form a basis of $\ker(M^\top)$ and let $\mathbf{e} \in \mathbb{F}^m$. By the Fundamental Theorem of Linear Algebra, $\ker(M^\top) = \mathrm{im}(M)^\perp$, so $[\![x]\!] \in \mathrm{im}(M)$ if and only if $N \cdot [\![x]\!] = \mathbf{0}$. Since $\{\mathcal{P}_i \in \mathcal{P} : i \in \rho(\mathrm{supp}(\mathbf{e}))\} \notin \Gamma$, it holds by Lemma 3.1 that either $\mathbf{e} \notin \mathrm{im}(M)$, or $\mathbf{e} \in \mathrm{im}(M)$ and $e = 0$.

If $\mathbf{e} \in \mathrm{im}(M)$ then $e = 0$ as required, while if $\mathbf{e} \notin \mathrm{im}(M)$ then $N \cdot \mathbf{e} \neq \mathbf{0}$. In the latter case, since $[\![x]\!] \in \mathrm{im}(M)$, it holds that $N \cdot [\![x]\!] = \mathbf{0}$ and hence $N \cdot \mathbf{c} = N \cdot ([\![x]\!] + \mathbf{e}) = N \cdot [\![x]\!] + N \cdot \mathbf{e} = \mathbf{0} + N \cdot \mathbf{e} \neq \mathbf{0}$. □

The matrix N is called the *cokernel* of M, and is analogous to the parity-check matrix of the code defined by generator matrix M from Coding Theory. The method to open a secret to a single party $\mathcal{P}_i$ is then immediate: all parties send their shares to $\mathcal{P}_i$, who then concatenates the shares into a share vector $[\![x]\!]$ and computes $N \cdot [\![x]\!]$. Since the adversary controls an unqualified set of parties, if $N \cdot [\![x]\!] = 0$ then by Lemma 3.2 the share vector $[\![x]\!]$ encodes the correct secret. In this case, $\mathcal{P}_i$ recalls any recombination vector $\boldsymbol{\lambda}$ and computes the secret as $s = \langle \boldsymbol{\lambda}, [\![x]\!] \rangle$, and otherwise tells the parties to abort.

It was shown in a report by Fehr [Feh99] that the matrix N in fact realizes the dual access structure by associating to it the same row map ρ as for M. Thus the result given is a special case of this observation. However, Lemma 3.1 only holds for $\mathcal{Q}_2$ access structures and is the crucial property needed for constructing an MPC protocol secure with abort.

3.3 Opening to All Parties

In this section, the method for opening secrets to all parties with authentication is given.

To open a secret in a passively-secure protocol, all parties can broadcast all of their shares so that all parties can reconstruct the secret. This method contains redundancy if the access structure is not full-threshold since proper subsets of parties can reconstruct the secret by definition of the access structure. This implies the existence of "minimal" communication patterns for each access structure and LSSS, in which parties only communicate sufficiently for every party to have all shares corresponding to a qualified set of parties.

In the active setting, the redundancy allows verification of opened secrets: honest parties can check *all* parties' broadcasted shares for correctness – for example, that

all shares lie on a polynomial of the appropriate degree for Shamir's secret-sharing. When reducing communication with the aim of avoiding the redundancy of broadcasting, honest parties must still be able to detect when the adversary sends inconsistent or erroneous shares.

The trick of the protocol presented here (for arbitrary LSSSs) is for parties to receive just enough shares to reconstruct the shares they did not see (which is at least the same as the number of shares they require to reconstruct the secret), and then at the end check that every party reconstructed the same share vector. To understand this more concretely, consider Shamir's secret-sharing scheme once more: a set of $t+1$ distinct points determines a unique polynomial of degree at most t that passes through them. This fact not only enables the secret to be computed using $t+1$ shares, but additionally allows the entire polynomial, and consequently all other shares, to be determined. The opening protocol then simply involves each party sending its share to its t left neighbours; by symmetry, each party will receive t shares, and hence can interpolate a polynomial, and thus determine the secret as well as all shares. After doing this possibly many times, the parties can take a hash of the concatenation of the polynomials (say, by concatenating the coefficient vectors) and at a later point in time comparing these hashes. If the hashes are all the same, then all reconstructed vectors are the same, and hence all parties must have computed the correct secrets. This, in essence, is the idea behind the protocol of Furakawa et al. [FLNW17] tailored to replicated secret-sharing in the $(3,1)$-threshold setting.

For a given LSSS, there is no reason *a priori* why parties should be able to reconstruct *shares* of all parties given just enough shares to reconstruct the secret (though in Shamir's secret-sharing this is indeed the case). To allow parties to perform reconstruction, each party is assigned a set of shares that it will receive, encoded as a map $\mathsf{q} : [n] \to 2^{[m]}$ defined as follows: for each $\mathcal{P}_i \in \mathcal{P}$, define $\mathsf{q}(i)$ to be a set $S_i \subseteq [m]$ such that:

- $\ker(M_{S_i}) = \{\mathbf{0}\}$; that is, the kernel of the submatrix M restricted to the rows indexed by S_i, is trivial; and
- $\rho^{-1}(\{i\}) \subseteq S_i$, where ρ^{-1} denotes the preimage of the row map ρ; that is, each party includes all of their own shares in the set S_i.

Remark 3.1. If $\ker(M) \neq \{\mathbf{0}\}$, such a map does not necessarily exist. In this situation, the MSP admits sharings of 0 with unqualified support. In other words, for some LSSSs it is not the case that *any* qualified set of parties have enough information to reconstruct all shares, though obviously the full set of parties suffices and in this case one can set

$\mathsf{q}(i) := [m]$ for every $i \in [n]$. On the other extreme, a formal description of MSPs in which all qualified sets of parties can reconstruct the entire share vector is given in Section 3.6.

In Section 3.5 a somewhat-optimized algorithm for finding a map q is given. These sets are used as follows. Each $\mathcal{P}_i$ receives a set of shares, denoted by $[\![x^i]\!]_{\mathsf{q}(i)}$, for a given secret x, where one hopes that $x^i = x^j$ for every $i, j \in [n]$. Then in order to reconstruct all shares, $\mathcal{P}_i$ tries to find $\mathbf{x}^i$ such that $[\![x^i]\!]_{\mathsf{q}(i)} = M_{\mathsf{q}(i)} \cdot \mathbf{x}^i$ and then computes $[\![x^i]\!] := M \cdot \mathbf{x}^i$ as the reconstructed share vector, which is then used to update the hash function (locally). Trivially, one can take $\mathsf{q}(i) = [m]$ for every $\mathcal{P}_i \in \mathcal{P}$, which corresponds to broadcasting all shares; however, better choices of q result in better communication efficiency.

If such an $\mathbf{x}^i$ does not exist then it must be because the adversary sent one or more incorrect shares, because $[\![x^i]\!]_{\mathsf{q}(i)}$ should be a subvector of *some* share vector. In this case, the party or parties unable to reconstruct send a message to all parties to abort.

If such an $\mathbf{x}^i$ does exist for each party then the adversary could still cause different parties to reconstruct different share vectors (and thus output different secrets), but then the hashes would differ and the honest parties would abort. The first condition, $\ker(M_{S_i}) = \{\mathbf{0}\}$, ensures that if all parties follow the protocol, they all reconstruct the *same* share vector, since there are multiple possible share vectors for a given secret, otherwise an honest execution may lead to an abort.

Indeed, the only thing the adversary can do without causing abort – either when opening secrets or later on when hashes are compared – is to change its shares so that when they are combined with the honest parties' shares, they form a valid share vector. Intuitively, one can think of this as the adversary rerandomizing the shares owned only by corrupt parties, which is not possible in Shamir or replicated secret-sharing, but in general is possible if and only if the LSSS admits non-trivial share vectors with unqualified support (for example, in disjunctive normal form (DNF)-based secret-sharing that was described in Section 2.4.3). Note that Lemma 3.2 prevents the adversary from changing the secret by changing shares of corrupt parties (without this being detected). If the adversary does choose to force different honest parties to compute different share vectors (albeit sharing the same secret), then the honest parties will only abort after comparing hash outputs, instead of during the opening procedure; note that if this is the only form of cheating then the output will actually be correct, even though the parties abort, which is undesirable as it offers "more ways" the adversary can cause the protocol to abort, but does not make a difference since the adversary can always abort just before output is given.

The intuition above can be summarized in the following lemma, that says that if all

parties are able to reconstruct share vectors and the share vectors are consistent, then the adversary cannot have introduced an error.

Lemma 3.3. *Let* $\mathsf{q} : [n] \to 2^{[m]}$ *be any map satisfying* $\ker(M_{S_i}) = \{\mathbf{0}\}$ *and* $\rho^{-1}(\{i\}) \subseteq S_i$ *for all* $i \in [n]$ *for an LSSS realizing a* $\mathscr{Q}_2$ *access structure, and let* $[\![x^i]\!]_{\mathsf{q}(i)}$ *denote the subvector of shares received by party* $\mathcal{P}_i$ *for a given secret. Suppose it is possible for each party* $\mathcal{P}_i \in \mathcal{P}$ *to find a vector* $\mathbf{x}^i$ *such that* $[\![x^i]\!]_{\mathsf{q}(i)} = M_{\mathsf{q}(i)} \cdot \mathbf{x}^i$ *and let* $[\![x^i]\!] \coloneqq M \cdot \mathbf{x}^i$ *for each* $i \in [n]$*. If* $[\![x^i]\!] = [\![x^j]\!]$ *for every pair of honest parties* $\mathcal{P}_i$ *and* $\mathcal{P}_j$*, then* $x^i = x^j$ *for all* $i, j \in [n]$*.*

Proof. The existence of q follows from the fact that "at worst" one can take $\mathsf{q}(i) = [m]$ for every $\mathcal{P}_i \in \mathcal{P}$. Since each party $\mathcal{P}_i$ can find some $\mathbf{x}^i$ such that $[\![x^i]\!]_{\mathsf{q}(i)} = M_{\mathsf{q}(i)} \cdot \mathbf{x}^i$, and $\ker(M_{\mathsf{q}(i)}) = \{\mathbf{0}\}$ for every $\mathcal{P}_i \in \mathcal{P}$ by the first requirement in the definition of q, in fact $\mathbf{x}^i$ is the unique solution in each case.

Let $\mathcal{A}$ denote the set of corrupt parties. Since $\mathcal{A}$ is unqualified, the honest parties form a qualified set $\mathcal{Q} = \mathcal{P} \setminus \mathcal{A}$ since the access structure is $\mathscr{Q}_2$.

Each honest party uses their own shares in the reconstruction process by the second requirement in the definition of q, so if $[\![x^i]\!] = [\![x^j]\!]$ for every pair of honest parties $\mathcal{P}_i$ and $\mathcal{P}_j$, then in particular they all agree on a qualified subvector defined by honest shares – i.e. $[\![x^i]\!]_{\mathcal{Q}} = [\![x^j]\!]_{\mathcal{Q}}$ for every pair of honest parties $\mathcal{P}_i$ and $\mathcal{P}_j$. Thus some qualified subvector of the share vector is well defined, which uniquely defines the secret by definition of MSP. □

As mentioned in the introduction, the results in the last two sections are somewhat analogous to the result of Cramer et al. [CDG+05, Thm 1] which roughly shows that for a strongly multiplicative LSSS implementing a $\mathscr{Q}_3$ access structure, honest parties can always agree on the correct secret (when all parties broadcast their shares).

Why not hash the secrets? In light of the fact that parties are merely agreeing on the opened value each time a secret is opened, intuitively one might think it suffices to compare the secrets reconstructed each time, and not the share vectors. However, this is not sufficient as one must account for a *rushing* adversary (see Section 2.5.1) as in the following example.

Example 3.1. This example demonstrates that hashing the secrets opened to all parties is insufficient to prevent an adversary from causing honest parties to agree on the

correct output. Consider the MSP defined by the matrix

$$\begin{matrix}\mathcal{P}_1\\ \mathcal{P}_1\\ \mathcal{P}_2\\ \mathcal{P}_3\\ \mathcal{P}_4\end{matrix}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 1 & 1\\ 0 & 0 & 1\end{pmatrix}$$

where the labels of the rows indicate the row map and where $\mathbf{t} = (1,1,1)$. One can check that the associated access structure is defined by

$$\Gamma^- = \{\{1,2\},\{1,3\},\{1,4\},\{2,3,4\}\}.$$

Let the map q be defined as follows:

$$\begin{aligned}\mathsf{q}(1) &= \{1,2\},\\ \mathsf{q}(2) &= \{1,2\},\\ \mathsf{q}(3) &= \{1,3\},\\ \mathsf{q}(4) &= \{1,4\}.\end{aligned}$$

One can check that it satisfies the requisite properties. Suppose the adversary corrupts party $\mathcal{P}_1$. In this case, since q specifies that every other party receives a share only from $\mathcal{P}_1$, $\mathcal{A}$ can easily cause the parties to open the secret $\hat{s}$ of its choosing as follows:

When $\mathcal{A}$ receives $\mathcal{P}_2$'s share $\mathbf{s}_3$, it samples $\mathbf{x}'$ subject to $\langle \mathbf{t}, \mathbf{x}' \rangle = \hat{s}$ and $\langle (1,0,1), \mathbf{x}' \rangle = \mathbf{s}_3$.[2] Then it computes

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 0 & 1\end{pmatrix} \cdot \mathbf{x}' = \begin{pmatrix}\mathbf{s}'_1\\ \mathbf{s}'_2\\ \mathbf{s}_3\end{pmatrix}$$

and sends $\mathbf{s}'_1$ and $\mathbf{s}'_2$ to $\mathcal{P}_2$.

When $\mathcal{A}$ receives $\mathcal{P}_3$'s share $\mathbf{s}_4$, it samples $\mathbf{x}''$ subject to $\langle \mathbf{t}, \mathbf{x}'' \rangle = \hat{s}$ and $\langle (0,1,1), \mathbf{x}'' \rangle = \mathbf{s}_4$. Then it computes

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 1\end{pmatrix} \cdot \mathbf{x}'' = \begin{pmatrix}\mathbf{s}''_1\\ \mathbf{s}''_2\\ \mathbf{s}_4\end{pmatrix}$$

and sends $\mathbf{s}''_1$ and $\mathbf{s}''_2$ to $\mathcal{P}_3$.

[2] Note that system is underconstrained so $\mathbf{x}'$ certainly exists.

Finally, when $\mathcal{A}$ receives $\mathcal{P}_4$'s share $\mathbf{s}_5$, it samples $\mathbf{x}'''$ subject to $\langle \mathbf{t}, \mathbf{x}''' \rangle = \hat{s}$ and $\langle (0,0,1), \mathbf{x}''' \rangle = \mathbf{s}_5$. Then it computes

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{x}''' = \begin{pmatrix} \mathbf{s}_1''' \\ \mathbf{s}_2''' \\ \mathbf{s}_5 \end{pmatrix}$$

and sends $\mathbf{s}_1'''$ and $\mathbf{s}_2'''$ to $\mathcal{P}_4$.

Now $\mathcal{P}_2$, $\mathcal{P}_3$ and $\mathcal{P}_4$ agree on a secret $\hat{s}$. While the *shares* are now inconsistent (i.e. it does not hold that $\mathbf{s}_1' = \mathbf{s}_1'' = \mathbf{s}_1'''$ and $\mathbf{s}_2' = \mathbf{s}_2'' = \mathbf{s}_2'''$), a comparison of the hash of the secret would not reveal this cheating behaviour.

3.4 Error-detection in Standard LSSSs

Perhaps the four most well-known LSSSs are additive, Shamir's, replicated, and DNF-based, as were outlined in Section 2.4.3. In this section, the error-detection property is explained as it applies to the last three LSSSs (since additive secret-sharing cannot be used to realize a $\mathscr{Q}_2$ access structure).

3.4.1 Shamir's Secret-Sharing

In the language of Coding Theory, error-detection for Shamir's secret-sharing is just checking whether or not the syndrome of the codeword (i.e. the share vector) is $\mathbf{0}$. For a $(5,2)$-threshold access structure, Shamir's secret-sharing can be expressed as an MSP $(\mathbb{F}, M, \mathbf{t}, \rho) = \mathcal{M}$ (where $\mathbb{F} := \mathbb{Q}$ for simplicity) where M and ρ are given by

$$\begin{matrix} \mathcal{P}_1 \\ \mathcal{P}_2 \\ \mathcal{P}_3 \\ \mathcal{P}_4 \\ \mathcal{P}_5 \end{matrix} \begin{pmatrix} 1^0 & 1^1 & 1^2 \\ 2^0 & 2^1 & 2^2 \\ 3^0 & 3^1 & 3^2 \\ 4^0 & 4^1 & 4^2 \\ 5^0 & 5^1 & 5^2 \end{pmatrix}$$

and $\mathbf{t} = (1,0,0)^\top$. The cokernel can be written as the image of the matrix

$$N = \begin{pmatrix} 1 & 0 & -6 & 8 & -3 \\ 0 & 1 & -3 & 3 & -1 \end{pmatrix}.$$

Now observe that if it holds for some vector $\mathbf{e}$ that $\{\mathcal{P}_i \in \mathcal{P} : i \in \rho(\mathrm{supp}(\mathbf{e}))\} \notin \Gamma$, then $\mathbf{e}$ encodes the evaluations of a polynomial f of degree at most 2 that has at least three zeroes, so $f \equiv 0$ and hence $\mathbf{e} = \mathbf{0}$, and so trivially $N \cdot \mathbf{e} = \mathbf{0}$.

3.4.2 Replicated Secret-Sharing

Consider the replicated secret-sharing for the $(3,1)$-threshold access structure, described using an MSP as follows:

$$\begin{matrix}\mathcal{P}_1\\ \mathcal{P}_1\\ \mathcal{P}_2\\ \mathcal{P}_2\\ \mathcal{P}_3\\ \mathcal{P}_3\end{matrix}\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\\1&0&0\\0&1&0\\0&0&1\end{pmatrix}$$

where $\mathbf{t} = (1,1,1)^\top \in \mathbb{F}^3$. Let N be the cokernel of M; one possible choice is:

$$N = \begin{pmatrix}1&0&0&-1&0&0\\0&1&0&0&-1&0\\0&0&1&0&0&-1\end{pmatrix}.$$

As with Shamir's secret-sharing, there are no share vectors with unqualified support: if $N \cdot \mathbf{e} = \mathbf{0}$, then $\mathbf{e}$ is of the form $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)^\top$ where $\mathbf{e}_i \in \mathbb{F}$. However, the image under ρ of the support of an error with this form (for which the encoded secret is non-zero) is necessarily qualified: for example, $\rho(\mathrm{supp}((1,0,0,1,0,0)^\top)) = \{1,2\} \in \Gamma$). Thus the adversary cannot change the share vector so that the new shares encode the same secret, let alone change the share so that it shares a different secret, without the resulting vector not being in $\mathrm{im}(M)$.

3.4.3 DNF-based Sharing

Consider the $\mathscr{Q}_2$ access structure given by

$$\Gamma^- = \{\{1,2\},\{1,3\},\{1,4\},\{2,3,4\}\}$$
$$\Delta^+ = \{\{1\},\{2,3\},\{2,4\},\{3,4\}\}.$$

One choice of MSP is:

$$\begin{array}{c} \mathcal{P}_1 \\ \mathcal{P}_1 \\ \mathcal{P}_1 \\ \mathcal{P}_2 \\ \mathcal{P}_2 \\ \mathcal{P}_3 \\ \mathcal{P}_3 \\ \mathcal{P}_4 \\ \mathcal{P}_4 \end{array} \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

where $\mathbf{t} = (1,0,0,0,0,0)^\top \in \mathbb{F}^6$. Let N be the cokernel of M, for some choice of basis; one possible choice is:

$$N = \begin{pmatrix} -1 & 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & -1 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

For example, the vector $(0,0,0,0,0,0,\mathbf{e}_5,0,-\mathbf{e}_5)^\top \in \mathbb{F}^9$ is the image of $(0,0,0,0,\mathbf{e}_5,-\mathbf{e}_5)^\top$ for any $\mathbf{e}_5 \in \mathbb{F}$, and it has unqualified support, namely $\{3,4\}$. However, observe that its image under N is $\mathbf{0}$, as required.

3.5 Finding a Reconstruction Map

Before providing the opening protocol, first an algorithm to find a map q such that $|\mathrm{supp}(\mathsf{q}(i))|$ is "quite small", for all i, is given. See Section 3.3 for the definition of q. In the MPC protocols later, each party computes this algorithm and selects the lexicographically first map, having already agreed on the access structure Γ and ordering its sets lexicographically. There is a choice as to whether to minimize the number of implied uni- or bi-directional channels. In the algorithm, by choosing the qualified sets as described one (crudely) optimizes the number of uni-directional channels.

For the algorithm given in Figure 3.2, the MSP matrix M is assumed to have linearly-independent columns, since if not then a linearly-independent subset can be taken to obtain a new LSSS which realizes the same access structure [BGP95]. It is also assumed that ρ is monotonically increasing so that each party owns a contiguous submatrix of M. If it is not, the rows can be interchanged so that this is true.

Algorithm for Determining Map q

Input MSP $(\mathbb{F}, M, \mathbf{t}, \rho)$ and corresponding access structure Γ. Recall $M \in \mathbb{F}^{m \times d}$.

Output Map q.

Algorithm

1. For each $\mathcal{P}_i \in \mathcal{P}$,
   a) Set $A := M_{\mathcal{P}_i}$, $S := \varnothing$, $k := 1$.
   b) Set $\mathcal{Q} \in \Gamma^-$ to be a smallest minimally qualified set containing $\mathcal{P}_i$.
   c) Set $k := 1$.
   d) While $\text{rank}(A) < d$ and $k \leq m$ Do
      i. While $\mathcal{P}_{\rho(k)} \notin \mathcal{Q}$ and $k < m$ Do
         A. Set $k := k + 1$.
      ii. End While.
      iii. If $k \leq m$ and the row M_k of M is linearly independent of the rows of A then
         A. Append M_k as a new row of A.
         B. $S := S \cup \{k\}$.
      iv. End If.
      v. Set $k := k + 1$.
   e) End While.
   f) $k := 1$.
   g) While $\text{rank}(A) < d$ and $k \leq m$ Do
      i. While $\mathcal{P}_{\rho(k)} \in \mathcal{Q}$ and $k < m$ Do
         A. Set $k := k + 1$.
      ii. End While.
      iii. If $k \leq m$ and the row M_k of M is linearly independent of the rows of A then
         A. Insert M_k into A.
         B. $S := S \cup \{k\}$.
      iv. End If.
      v. $k := k + 1$.
   h) End While.
   i) Set $\text{q}(i) := S$.
2. End For.
3. Output q.

Figure 3.2: Algorithm for Determining Map q.

3.6 Share-Reconstructability

To conclude this chapter, the notion of *share-reconstructability* is defined. An MSP that is share-reconstructable is one in which share vectors can be reconstructed entirely from

any qualified subvector. They are of interest because their use offers particularly good communication efficiency in Π_{Open}, as $\mathsf{q}(i)$ is "as small as possible" for each $\mathcal{P}_i \in \mathcal{P}$, which minimizes communication. If an MSP that is both share-reconstructable *and* ideal (such as Shamir's scheme) is used, then the communication cost attains its minimum for that access structure.

Definition 3.1. Let $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$ be an MSP, where $\ker(M) = \{\mathbf{0}\}$, computing a $\mathscr{Q}_2$ access structure Γ. $\mathcal{M}$ is called *share-reconstructable* if for every $\mathbf{s} \in \mathrm{im}(M)$, for every $\mathcal{Q} \in \Gamma$, $\mathbf{s}_\mathcal{Q}$ uniquely determines the vector $\mathbf{s}$.

The following lemma is a restatement but provides a more concrete check of whether or not a given MSP is share-reconstructable, though it requires the computation of exponentially-many submatrices (naïvely).

Lemma 3.4. *Let $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$ be an MSP, where $\ker(M) = \{\mathbf{0}\}$, computing a $\mathscr{Q}_2$ access structure Γ. Then $\mathcal{M}$ is share-reconstructable if and only if for every $\mathcal{Q} \in \Gamma$ it holds that $\ker(M_\mathcal{Q}) = \{\mathbf{0}\}$.*

Proof. Suppose $M_\mathcal{Q}$ has full column rank for all $\mathcal{Q} \in \Gamma$, and that there exist $\mathbf{s}, \mathbf{s}' \in \mathrm{im}(M)$ such that $\mathbf{s} \neq \mathbf{s}'$ but $\mathbf{s}_\mathcal{Q} = \mathbf{s}'_\mathcal{Q}$ for some $\mathcal{Q} \in \Gamma$. Then let $M \cdot \mathbf{x} = \mathbf{s}$ and $M \cdot \mathbf{x}' = \mathbf{s}'$. Then $M_\mathcal{Q} \cdot (\mathbf{x} - \mathbf{x}') = M_\mathcal{Q} \cdot \mathbf{x} - M_\mathcal{Q} \cdot \mathbf{x}' = \mathbf{s}_\mathcal{Q} - \mathbf{s}'_\mathcal{Q} = \mathbf{0}$, so since $\ker(M_\mathcal{Q}) = \{\mathbf{0}\}$, we have $\mathbf{x} = \mathbf{x}'$. But then $\mathbf{s} = M \cdot \mathbf{x} = M \cdot \mathbf{x}' = \mathbf{s}'$, which is a contradiction. Thus no such pair of share vectors exist, so $\mathcal{M}$ is share-reconstructable.

Suppose there exists some $\mathcal{Q} \in \Gamma$ such that $\ker(M_\mathcal{Q}) \neq \{\mathbf{0}\}$ and suppose we are given $\mathbf{s}_\mathcal{Q} \neq \mathbf{0}$. Fix some $\mathbf{x}$ such that $\mathbf{s}_\mathcal{Q} = M_\mathcal{Q} \cdot \mathbf{x}$ and fix $\mathbf{k} \in \ker(M_\mathcal{Q}) \setminus \{\mathbf{0}\}$. We have $M_\mathcal{Q} \cdot (\mathbf{x} + \mathbf{k}) = M_\mathcal{Q} \cdot \mathbf{x} + M_\mathcal{Q} \cdot \mathbf{k} = \mathbf{s}_\mathcal{Q} + \mathbf{0} = \mathbf{s}_\mathcal{Q}$. Let $\mathbf{s} = M \cdot \mathbf{x}$ and $\mathbf{s}' = M \cdot (\mathbf{x} + \mathbf{k})$. Since $\ker(M) = \{\mathbf{0}\}$ and $\mathbf{k} \neq \mathbf{0}$ it holds that $M \cdot \mathbf{k} \neq \mathbf{0}$, so $\mathbf{s}' = M \cdot (\mathbf{x} + \mathbf{k}) = M \cdot \mathbf{x} + M \cdot \mathbf{k} \neq M \cdot \mathbf{x} = \mathbf{s}$; but $\mathbf{s}_\mathcal{Q} = \mathbf{s}'_\mathcal{Q}$, so $\mathbf{s}_\mathcal{Q}$ does not have a unique reconstruction, and hence $\mathcal{M}$ is not share-reconstructable. □

Example 3.2. Shamir's secret-sharing scheme is an example of a share-reconstructable LSSS since any t rows of the Vandermonde matrix are linearly independent.

Example 3.3. Consider the access structure defined by $\Gamma^- = \{\{1,2\},\{1,3\},\{1,4\},\{2,3,4\}\}$ and $\Delta^+ = \{\{1\},\{2,3\},\{2,4\},\{3,4\}\}$ computed by the MSP given as follows:

$$\begin{matrix} \mathcal{P}_1 \\ \mathcal{P}_1 \\ \mathcal{P}_2 \\ \mathcal{P}_3 \\ \mathcal{P}_4 \end{matrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

where $\mathbf{t} = (1,1,1)^\top$. The reader may check that $\ker(M_\mathcal{Q}) = \{\mathbf{0}\}$ for every $\mathcal{Q} \in \Gamma^-$, so given $\mathbf{s}_\mathcal{Q}$ there is always a unique solution to $M_\mathcal{Q} \cdot \mathbf{x} = \mathbf{s}_\mathcal{Q}$ for $\mathbf{x}$, and hence $\mathbf{s}$ can be determined from $\mathbf{s}_\mathcal{Q}$ by finding $\mathbf{x}$ and computing $\mathbf{s} = M \cdot \mathbf{x}$.

We know by Lemma 3.1 that for any MSP computing a $\mathscr{Q}_2$ access structure, share vectors all have qualified support unless they encode the secret 0; to allow a unique construction of each share vector from qualified subsets, share-reconstructable MSPs are those for which the secret 0 is only encoded via the zero vector or with share vectors with qualified support.

Lemma 3.5. *Let $\mathcal{M} = (\mathbb{F}, M, \mathbf{t}, \rho)$ be an MSP, where $\ker(M) = \{\mathbf{0}\}$, computing a $\mathscr{Q}_2$ access structure Γ. Then $\mathcal{M}$ is share-reconstructable if and only if*

$$\mathbf{s} \in \mathrm{im}(M) \implies \rho(\mathrm{supp}(\mathbf{s})) \in \Gamma \cup \{\varnothing\}$$

for all $\mathbf{s} \in \mathbb{F}^m$.

In other words, $\mathcal{M}$ is share-reconstructable if and only if the only share vector with unqualified support which encodes the secret 0 is the zero vector. For intuition, one can think of Shamir's scheme: the only polynomial of degree at most t which has at least $n - t > t$ zeros is the zero polynomial. The statement is corollary to the previous lemmata by linearity of the MSP, but we give the formal proof below.

Proof. Suppose that $\mathcal{M}$ is not share-reconstructable. Then there exists $\mathcal{Q} \in \Gamma$ for which $\exists\ \mathbf{x} \in \ker(M_\mathcal{Q})$ such that $\mathbf{x} \neq \mathbf{0}$. Since $M_\mathcal{Q} \cdot \mathbf{x} = \mathbf{0}$ it holds that $\rho(\mathrm{supp}(M\mathbf{x})) \subseteq \mathcal{P} \setminus \mathcal{Q}$, which is unqualified, since Γ is $\mathscr{Q}_2$. Since $\ker(M) = \{\mathbf{0}\}$ and $\mathbf{x} \neq \mathbf{0}$, it holds that $M \cdot \mathbf{x} \neq \mathbf{0}$. Now $M \cdot \mathbf{x} \in \mathrm{im}(M)$, is non-zero, and has unqualified support. In other words, we have $M \cdot \mathbf{x} \in \mathrm{im}(M)$ and $\mathrm{supp}(M \cdot \mathbf{x}) \notin \Gamma \cup \{\varnothing\}$.

Conversely, suppose there exists some $\mathbf{s} \in \mathrm{im}(M)$ such that $\mathrm{supp}(\mathbf{s}) \notin \Gamma \cup \{\varnothing\}$, i.e., $\mathbf{s} \neq \mathbf{0}$ and $\mathrm{supp}(\mathbf{s})$ is unqualified. Let $\mathbf{x}$ be such that $\mathbf{s} = M \cdot \mathbf{x}$, which exists since $\mathbf{s} \in \mathrm{im}(M)$. Then $\mathcal{Q} := \mathcal{P} \setminus \rho(\mathrm{supp}(\mathbf{s}))$ is qualified since Γ is $\mathscr{Q}_2$. Since $\mathbf{s}_\mathcal{Q} = \mathbf{0}$, we have $M_\mathcal{Q} \cdot \mathbf{x} = \mathbf{s}_\mathcal{Q} = \mathbf{0}$, so $\mathbf{x} \in \ker(M_\mathcal{Q})$. If $\mathbf{x} = \mathbf{0}$ then $\mathbf{s} = M \cdot \mathbf{x} = \mathbf{0}$; but $\mathrm{supp}(\mathbf{s}) \neq \varnothing$, so this is not the case, and so $\ker(M_\mathcal{Q}) \neq \{\mathbf{0}\}$, and thus $\mathcal{M}$ is not share-reconstructable by Lemma 3.4. □

Share-reconstructable MSPs yield comparatively communication-efficient instantiations of our protocol because to each $\mathcal{P}_i$ the map q can assign a smallest $\mathcal{Q} \in \Gamma^-$ containing $\mathcal{P}_i$.

Theorem 3.1. *For every $\mathscr{Q}_2$ access structure there exists a share-reconstructable MSP computing it.*

Proof. Replicated secret-sharing is always share-reconstructable since a qualified set of parties together hold all shares by definition and so can vacuously compute the shares held by all other parties. □

Chapter 4

Modelling Preprocessing

This chapter is based on results from [SW19], as detailed in Section 1.2, which was joint work with Nigel Smart.

This chapter This chapter describes how to model secret-sharing-based multi-party computation (MPC) (with active security) less abstractly than previous work, by the functionalities explicitly referencing the linear secret-sharing schemes (LSSSs) being used to execute MPC. Modelling in this way provides a framework for the MPC protocols described later which is crucial for the protocol presented in Chapter 5.

4.1 Overview

In this chapter, a preprocessing functionality is built up from an "opening" functionality, which takes a LSSS and adds authentication. This allows one to talk of *shares* held by parties as in [DPSZ12] instead of handles of elements in an "authenticated dictionary" as in MASCOT [KOS16] and Overdrive [KPR18]. One of the benefits of modelling using linear authentication functionalities was that it apparently helped to unify the somewhat-homomorphic encryption (SHE) and oblivious transfer (OT) approaches of preprocessing in the full-threshold context.

Concretely, the reasons for talking about shares rather than identifiers are as follows.

- Abstracting from the view that the secrets are shared using a form of authenticated LSSS makes it difficult to build protocols that use this data *explicitly*. If the protocol is built with a particular functionality in mind (as is the case for many protocols),

it is logical to built up in this way. However, if a modular approach is taken with the view in mind of allowing greater latitude in what protocols can be realized from more basic subprotocols, modelling in this way imposes unnecessary restriction. For example, in the outsourcing protocol in Chapter 5, the natural process of resharing a secret amongst a different set of parties is not possible when treating preprocessed data merely as a dictionary.

- By modelling the "opening" of secrets specifically, an *asynchronous* protocol can be obtained by changing the opening functionality. So far the discussion has been limited to synchronous networks, but in the functionality $\mathcal{F}_{\mathsf{Open}}$ presented in this chapter, communication in online protocols *exclusively* uses commands in $\mathcal{F}_{\mathsf{Prep}}$ (as an extension of $\mathcal{F}_{\mathsf{Open}}$). Consequently, any asynchronous protocol securely realizing $\mathcal{F}_{\mathsf{Open}}$ should give an asynchronous protocol realizing $\mathcal{F}_{\mathsf{ABB}}$.

The compromise taken in this chapter is to treat the authentication as black-box but to retain the shares, which reflects the fact that there exist different ways of authenticating depending on the access structure and LSSS. For example, it was shown in Chapter 3 that a LSSS realizing a $\mathscr{Q}_2$ access structure is in a sense "self-authenticating"; in the full-threshold context, different types of information-theoretic (IT) message authentication code (MAC) can be used – for example, the global MAC used in SPDZ, described in Section 2.5.3. This modelling also dovetails well with non-LSSS-based MPC protocols (i.e. garbled circuits [LPSY15, LSS16, WRK17a, WRK17b, HSS17, HOSS18b, HOSS18a]) which often realize functionalities generating authenticated secret-shared bits.

In more detail, the parties will realize an "opening" functionality $\mathcal{F}_{\mathsf{Open}}$ that checks whether or not shares are correct as the parties open secrets. The main difference between this functionality and the linear authentication functionality is that it does not store secrets itself: instead, an initialization step "commits" the parties to open only "valid" secrets in the scheme. (One can think of this as commitments to shares of the MAC in the full-threshold setting, which is achieved in SPDZ by the parties broadcasting an encryption of their share of the MAC key.) This is closer to what actually happens in a protocol execution since shares can be locally modified by an active adversary: instead of the functionality storing secrets and the simulator introducing an error, the functionality accepts shares and causes an error *specifically* by sending an invalid set of shares.

The benefits of modelling in this way manifest themselves throughout the remainder of this thesis, allowing all of the protocols to be phrased in the same language, realizing

the same functionalities with respect to different access structures. This does not make the modelling here better *per se*, but it enables description of a wider range of protocols in the same language, which is very helpful for the protocols in this thesis. In previous works such as [DPSZ12,DKL+13], shares (not handles to secrets) were modelled but the functionalities were specific to the LSSS and method of authentication.

4.2 Opening Functionality

In this section the functionality $\mathcal{F}_{\mathsf{Open}}$ is presented, which is generic in the sense that it allows any linear secret-sharing scheme combined with a linear authentication scheme to check secrets are opened correctly.

The degree of abstraction of a functionality is a design choice. For example, the classical arithmetic black box from Section 2.5 could more succinctly be described as a box accepting any arithmetic circuit as input and providing the parties with output. Instead, in practice cryptographers choose to realize a black box in which more fundamental operations, specifically, addition and multiplication, can be computed separately.

The functionality $\mathcal{F}_{\mathsf{Open}}$ that will be presented in this section is not a significant abstraction from the protocol that will realize it in Chapter 3: its purpose is to encode all information about secret-sharing, and error-detection therein, with high-level function calls. Abstracting to this degree, and no further, has the advantage of allowing *both* abstract references to opening secrets with authentication, *and* low-level manipulation of the secret-shared data, without the need to fiddle around with the details of how secrets are authenticated.

The functionality is not a full verifiable secret-sharing (VSS) scheme as the secret may not necessarily be learnt by honest parties if the adversary causes an abort to occur. This relaxation matches the technique common to MPC literature of conceding full robustness to "security with abort". The functionality also involves a "checking" procedure – that is, secrets are opened optimistically and then checked later – which is analogous to the verification step of a VSS scheme. The key difference between this "authenticated opening" functionality and a full VSS scheme is its failure to achieve robustness.

Authentication During initialization of the functionality, the parties specify the LSSS they wish to use. In the context of an LSSS realizing a $\mathscr{Q}_2$ access structure, the results of Chapter 3 show that no further information beyond the description of the LSSS is

required to allow secrets to be authenticated. However, it may be that parties wish to authenticate in a different way – for example, using an IT MAC, such as in the full-threshold setting. For clarity of notation, this detail is omitted from the functionality description, for which the focus is the $\mathscr{Q}_2$ setting, but one can think of the LSSS provided in the command (as $[\![\cdot]\!]$) as "containing" information regarding the authentication. A high-level view of how this works is given in Section 4.7.

From this point onwards, $[\![\cdot]\!]$ is taken to mean a secret is secret-shared *and* authenticated – be that "for free" in the $\mathscr{Q}_2$ setting, or with MACs in the full-threshold setting.

The functionality $\mathcal{F}_{\mathsf{Open}}$ can be viewed as an extension $\mathcal{F}_{\mathsf{Broadcast}}$ but for clarity is kept as a separate functionality. For now, the map $\mathsf{q} : [n] \to 2^{[m]}$ can be defined as $\mathsf{q}(i) = [m]$ for all $i \in [n]$ and encodes which shares should be received by $\mathcal{P}_i$ for all $i \in [n]$; in Section 3.3, it is shown that it is not necessary for all parties to receive all shares to realize this functionality, in certain situations. The functionality $\mathcal{F}_{\mathsf{Open}}$ is given in Figure 4.1.

Functionality $\mathcal{F}_{\mathsf{Open}}$

Initialize On input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)$ from each honest party and from $\mathcal{S}$ on behalf of corrupt parties, set `Abort` to false. Compute the error matrix N for $[\![\cdot]\!]$.

Send On input $(\texttt{Send}, x, j, sid)$ from $\mathcal{P}_i$, or from $\mathcal{S}$ if $i \in A$, send x to $\mathcal{P}_j$, or to $\mathcal{S}$ if $j \in A$.

Broadcast On input $(\texttt{Broadcast}, x, sid)$ from $\mathcal{P}_i$, or from $\mathcal{S}$ if $i \in A$, await $n-1$ further calls with input $(\texttt{Send}, x^j, j, sid)$ for all $j \in [n] \setminus (A \cup \{i\})$. If $x^i \neq x^j$ for any $i \neq j$, set `Abort` to true.

Open To One On input $(\texttt{Open}, i, id, sid)$ from all honest parties and $\mathcal{S}$, do the following:

1. For each $j \in [n] \setminus A$, await a vector of shares $[\![x]\!]_{\mathcal{P}_j}$ from honest party $\mathcal{P}_j$.
2. If $i \in [n] \setminus A$,
   a) Await a vector of shares $[\![x]\!]_{\mathcal{A}}$ from $\mathcal{S}$. If $N \cdot [\![x]\!] \neq \mathbf{0}$ then set `Abort` to true.[a]
   b) Set $x := \langle \boldsymbol{\lambda}, [\![x]\!] \rangle$ and send x to $\mathcal{P}_i$.

   If $i \in A$,
   a) Send $[\![x]\!]_{\mathcal{P} \setminus \mathcal{A}}$ to $\mathcal{S}$.
   b) Await a message OK or Abort from $\mathcal{S}$. If the message is OK then continue; otherwise send the message Abort to all honest parties, and then halt.

Open To All On input $(\texttt{Open}, 0, id, sid)$, the functionality does the following:

1. For each $i \in [n] \setminus A$, await a vector of shares $[\![x]\!]_{\mathcal{P}_i}$ from honest party $\mathcal{P}_i$.
2. Send $\{[\![x]\!]_k : \rho(k) \in [n] \setminus A \text{ and } k \in \mathsf{q}(A)\}$ to $\mathcal{S}$.
3. Await a message OK or Abort from $\mathcal{S}$. If the message is Abort, then send the message Abort to all honest parties, and then halt; otherwise, continue.
4. Await a set of vectors of shares $\{[\![x^i]\!]_k : \rho(k) \in A \text{ and } k \in \mathsf{q}([n] \setminus A)\}_{i \in [n] \setminus A}$ or a message Abort from $\mathcal{S}$. If the message is Abort, then send the message Abort to all honest parties, and then halt; otherwise, continue.
5. For each $i \in [n] \setminus A$, set $x^i = \langle \boldsymbol{\lambda}^i_{\mathsf{q}(i)}, [\![x^i]\!]_{\mathsf{q}(i)} \rangle$, solve $M_{\mathsf{q}(i)} \cdot \mathbf{x}^i = [\![x^i]\!]_{\mathsf{q}(i)}$ for $\mathbf{x}^i$ and send x^i to $\mathcal{P}_i$; if there are no solutions, send the message Abort to all honest parties, and then halt. If $\mathbf{x}^i \neq \mathbf{x}^j$ for any $i, j \in [n] \setminus A$ then set `Abort` to true.

Verify On input $(\texttt{Verify}, sid)$ from all honest parties and $\mathcal{S}$, await a message OK or Abort from $\mathcal{S}$. If the message is OK and `Abort` is false then send the message OK to all honest parties and continue; otherwise send the message Abort to all honest parties, and then halt.

[a] Note that for a full-threshold access structure, N is the zero matrix.

Figure 4.1: Opening Functionality, $\mathcal{F}_{\mathsf{Open}}$.

Notice that the honest parties send *all* of their shares to $\mathcal{F}_{\mathsf{Open}}$. This reflects the fact that $\mathcal{F}_{\mathsf{Open}}$ is essentially a subroutine to which the parties send their shares: the fact that they only actually "send" a subset of shares to other parties is acknowledged in the protocol realizing $\mathcal{F}_{\mathsf{Open}}$.

It is also important to note that the adversary may send erroneous share vectors to honest parties when providing input: if the adversary corrects this error later on, the parties should *not* abort, which is why the functionality does not flag for an abort to occur on receiving shares for honest parties as in input for a corrupt party. This makes the simulation more "natural" in the sense that $\mathcal{S}$ need not keep a record of whether or not it should abort after each input: it merely executes exactly as the honest parties would, aborting as appropriate.

4.3 Opening Protocol

The protocol Π_{Open} given in Figure 4.5 uses the results of Chapter 3 to define a sort of "authenticated opening" protocol. Note that although the hash function is not guaranteed to be hiding, parties only hash shares of secrets that are being made public, so no secret information is leaked when the hashes are revealed. Indeed, it would suffice for parties to send the reconstructed share vectors to each other and verify they are the same as what each computed itself: the hash function is used simply to amortize this cost. This also means it does not matter if the hash function is evaluated on "short" inputs (say, where the string is fewer than κ bits long) since the only thing that matters is that honest parties agree on what the output is (and will abort if not the case). This is one advantage of only requiring security with abort rather than full robustness.

4.3.1 Agreement Protocol

The agreement of parties on local secrets is dependent both on broadcasted elements and on reconstructed elements which are computed from local data and are expected (optimistically) to be the same for all honest parties. In the context of security with abort, "broadcast" can be achieved over point-to-point secure channels, as was discussed in Section 2.3.4; it is abstracted as the functionality $\mathcal{F}_{\mathsf{Broadcast}}$.

The functionality $\mathcal{F}_{\mathsf{Agreement}}$ is given in Figure 4.2.

Functionality $\mathcal{F}_{\mathsf{Agreement}}$

Initialize On input (`Initialize`,sid) from all parties, where sid is a new session identifier, set `Abort` to false.

Add To Agreement On input (`Agree`,x^i,sid) from $\mathcal{P}_i$ for all $i \in [n]$, or from $\mathcal{S}$ if $i \in A$, if $x^i \neq x^j$ for some $j \neq i$ then set `Abort` to true.

Verify On input (`Verify`,sid) from all parties and $\mathcal{S}$, await a message Abort or OK from $\mathcal{S}$. If the message is OK and `Abort` is false, send the message OK to all honest parties and continue; otherwise, send the message Abort to all honest parties, and then and halt.

Figure 4.2: Agreement Functionality, $\mathcal{F}_{\mathsf{Agreement}}$.

The protocol to realize $\mathcal{F}_{\mathsf{Agreement}}$ makes use of a hash function. When implementing the hash function in practice, while one could write that the parties store the string str as the protocol is executed (which may be of arbitrary length and appended throughout) and that they evaluate the hash function at the end; instead, in order to model real-world practice, in the protocol as soon as this string is larger than some length parameter, parties begin to evaluate. This can be expressed by writing that the hash function is initialized and then its state is updated using consecutive chunks of the string, i.e. by writing $str = str_1 \| \cdots \| str_t$ and to evaluate one executes $\mathcal{H}$.`Initialize`() and then a sequence of updates

$$\mathcal{H}.\texttt{Update}(str_1), \ldots, \mathcal{H}.\texttt{Update}(str_t)$$

and finally compute $h := \mathcal{H}.\texttt{Finalize}()$. This application programming interface (API) is given in Figure 4.3.

Hash Function Interface

Given an efficiently-computable function $H : \{0,1\}^* \to \{0,1\}^\kappa$, a stateful object $\mathcal{H}$ is defined to have the following three procedures. To evaluate H on an input $m = m_1 \| \cdots \| m_{t-1} \| Padd(m_t)$ where $Padd$ denotes a padding function, where each m_i is of the correct block size len, do the following:

Initialize When $\mathcal{H}$.`Initialize`() is called, set $state(IV)$ where IV is the initialization vector.

Update When $\mathcal{H}$.`Update`(m_i) is called, update $state$ according to the description of the hash function.

Output When $\mathcal{H}$.`Finalize`() is called, perform any finalization procedure prescribed by the hash function definition and then return $state$.

Figure 4.3: Hash Function Interface.

Finally, the protocol is given in Figure 4.4. It uses the algorithm given in Figure 3.2 to determine a map q.

Protocol $\Pi_{\mathsf{Agreement}}$

This protocol is realized in the $\mathcal{F}_{\mathsf{Broadcast}}$-hybrid model.

Initialize

1. The parties agree on a new session identifier sid and call $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Initialize}, sid)$.
2. The parties agree on and initialize a hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$ implemented by the stateful object $\mathcal{H}$ (see Figure 4.3) and each party initializes it by executing $\mathcal{H}.\texttt{Initialize}()$. They obtain a parameter len.
3. For each $i \in [n]$, party $\mathcal{P}_i$ locally sets $str^i := \varepsilon$ (the empty word).

Add To Agreement If each party holds x^i and they want to check that $x^i = x^j$ for all $i, j \in [n]$, each party $\mathcal{P}_i$ does the following:

1. Interpret x^i as a binary string and append it to the current string to check, $str^i := str^i \| x^i$, where $\|$ denotes concatenation of strings.
2. If $|str^i| > len$ then set s to be the first len bits of str^i, compute $\mathcal{H}.\texttt{Update}(s)$, and truncate the first len bits off str^i.

Verify To verify all values so far, each $\mathcal{P}_i \in \mathcal{P}$ does the following:

1. Execute $\mathcal{H}.\texttt{Update}(str^i)$ (multiple times, with padding if necessary) and then set $h^i := \mathcal{H}.\texttt{Finalize}()$.
2. Call $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Broadcast}, h^i, sid)$ and await the messages $\{h^j\}_{j \in [n] \setminus \{i\}}$ from other parties.
3. Call $\mathcal{F}_{\mathsf{Broadcast}}$ with input $(\texttt{Verify}, sid)$, and if it aborts, or if $h^j \neq h^i$ for any $j \in [n] \setminus \{i\}$, then (locally) output $\perp$, and then halt; otherwise, continue.

Figure 4.4: Agreement Protocol, $\Pi_{\mathsf{Agreement}}$.

Protocol Π_{Open}

This protocol is realized in the $\mathcal{F}_{\mathsf{Agreement}}$-hybrid model. If at any point a party receives the message Abort, it runs the subprotocol **Abort**.

Initialize

1. Agree on a session identifier sid, and then agree on the $\mathscr{Q}_2$ access structure, Γ, and an monotone span program (MSP) $[\![\cdot]\!]$ realizing it.
2. Execute the algorithm in Figure 3.2 to obtain the map q.
3. For each $i \in [n]$, each party computes $\boldsymbol{\lambda}^i$ as the lexicographically first recombination vector such that $\mathrm{supp}(\boldsymbol{\lambda}^i) \subseteq \mathsf{q}(i)$.
4. Call an instance of $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Initialize}, sid)$.

Send Party $\mathcal{P}_i$ sends a secret x to $\mathcal{P}_j$ over a secure channel.

Broadcast When $\mathcal{P}_i$ calls this procedure to broadcast a value x,

1. Party $\mathcal{P}_i$ sends the secret x to all other players over pair-wise secure channels; let x^j denote the value received by $\mathcal{P}_j$.
2. All parties call $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Agree}, x^j, sid)$.

Open To One To open a secret $[\![x]\!]$ to one party $\mathcal{P}_i$, the parties do the following:

1. Each $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ sends $[\![x]\!]_{\mathcal{P}_j}$ to $\mathcal{P}_i$, who concatenates local and received shares into a vector $[\![x]\!]$.
2. Party $\mathcal{P}_i$ computes $N \cdot [\![x]\!]$; if it is equal to $\mathbf{0}$, $\mathcal{P}_i$ (locally) outputs $s = \langle \boldsymbol{\lambda}^i, [\![x]\!] \rangle$, and otherwise runs **Abort**.

Open To All To open a secret $[\![x]\!]$ to all parties, each $\mathcal{P}_i \in \mathcal{P}$ does the following:

1. Retrieve from memory the recombination vector $\boldsymbol{\lambda}^i$.
2. For each $\mathcal{P}_j \in \mathcal{P}$, for each $k \in \mathsf{q}(j) \subseteq [m]$, if $\rho(k) = i$ then $\mathcal{P}_i$ sends $[\![x]\!]_k$ to $\mathcal{P}_j$ over an authenticated channel.
3. For each $k \in \mathsf{q}(i)$, wait to receive $[\![x]\!]_k$ from party $\mathcal{P}_{\rho(k)}$.
4. Concatenate local and received shares into a vector $[\![x]\!]^i_{\mathsf{q}(i)} \in \mathbb{F}^{|\mathsf{q}(i)|}$.
5. Solve $M_{\mathsf{q}(i)} \cdot \mathbf{x}^i = [\![x]\!]^i_{\mathsf{q}(i)}$ for $\mathbf{x}^i$. If there are no solutions, run **Abort**.
6. Call $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Agree}, \mathbf{x}^i, sid)$.
7. (Locally) output $x = \langle \boldsymbol{\lambda}^i_{\mathsf{q}(i)}, [\![x]\!]^i_{\mathsf{q}(i)} \rangle$.

Verify The parties call $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Verify}, sid)$. If the functionality sends the message Abort then the parties run **Abort**; otherwise they continue.

Abort If a party calls this subroutine, or if it receives a message Abort from any other party, it sends a message Abort to each other party over a secure channel, (locally) outputs $\perp$, and then halts.

Figure 4.5: Opening Protocol, Π_{Open}.

Theorem 4.1. *The protocol Π_{Open} universal composability (UC)-securely realizes the functionality $\mathcal{F}_{\mathsf{Open}}$ against a static, active adversary in the $\mathcal{F}_{\mathsf{Agreement}}$-hybrid model.*

Simulator $\mathcal{S}_{\mathsf{Open}}$

Initialize

1. Agree on a session identifier sid with $\mathcal{A}$, and then agree on the $\mathscr{Q}_2$ access structure, Γ, and an MSP $[\![\cdot]\!]$ realizing it, with $\mathcal{A}$; then call $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)$.
2. Execute the algorithm in Figure 3.2 to obtain the map q.
3. For each $i \in [n]$, compute $\boldsymbol{\lambda}^i$ as the lexicographically first recombination vector such that $\mathrm{supp}(\boldsymbol{\lambda}^i) \subseteq \mathsf{q}(i)$.
4. Await the call to $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Initialize}, sid)$ from $\mathcal{A}$ and initialize a local instance.

Send If $\mathcal{P}_i \in \mathcal{A}$ is sending a message to $\mathcal{P}_j \in \mathcal{P} \setminus \mathcal{A}$, then await x from $\mathcal{A}$ and then call $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Send}, x, j, sid)$. If $\mathcal{P}_j \in \mathcal{A}$ is awaiting a message from honest party $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{A}$, then await a message x from $\mathcal{F}_{\mathsf{Open}}$ and forward x to $\mathcal{A}$.

Broadcast

1. When $\mathcal{P}_i$ calls this procedure to broadcast a value ε,
   If $i \in [n] \setminus A$, await ε from $\mathcal{F}_{\mathsf{Open}}$ and forward this to $\mathcal{A}$.
   If $i \in A$, await a set of inputs $\{\varepsilon^i\}_{i \in [n] \setminus A}$ from $\mathcal{A}$, and then call $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Broadcast}, \varepsilon^i, sid)$ for any i followed by inputs $(\texttt{Send}, \varepsilon^i, i, sid)_{i \in [n] \setminus A}$.
2. Await the call to $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Agree}, x^j, sid)$ and execute it honestly with $\mathcal{A}$.

Open To One To open $[\![x]\!]$ to $\mathcal{P}_i$,

If $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{A}$,

1. Await a vector of shares $[\![x]\!]_{\mathcal{A}}$ from $\mathcal{A}$.
2. Send the vector $[\![x]\!]_{\mathcal{A}}$ to $\mathcal{F}_{\mathsf{Open}}$.

If $\mathcal{P}_i \in \mathcal{A}$,

1. Await the share vector $[\![x]\!]_{\mathcal{P} \setminus \mathcal{A}}$ from $\mathcal{F}_{\mathsf{Open}}$ and send $[\![x]\!]_{\mathcal{P} \setminus \mathcal{A}}$ to $\mathcal{A}$.
2. If $\mathcal{A}$ sends the message Abort then send the message Abort to $\mathcal{F}_{\mathsf{Open}}$; otherwise send the message OK.

Open To All

1. For each emulated honest party $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{A}$, retrieve from memory the recombination vector $\boldsymbol{\lambda}^i$.
2. Await a set of share vectors $\{[\![x]\!]_{\mathsf{q}(i) \cap \rho^{-1}([n] \setminus A)} : i \in [n] \setminus A\}$ from $\mathcal{F}_{\mathsf{Open}}$ and send these to $\mathcal{A}$.
3. Await a message set of vectors of shares $\{[\![x]\!]_{\mathsf{q}(i) \cap \rho^{-1}(A)} : i \in [n] \setminus A\}$ from $\mathcal{A}$. If the shares are not sent, or if $\mathcal{A}$ sends a message Abort, then send the message Abort to $\mathcal{F}_{\mathsf{Open}}$; otherwise, send the message OK and forward the share vectors to $\mathcal{F}_{\mathsf{Open}}$.
4. Concatenate local and received shares into a vector denoted by $\mathbf{s}^i_{\mathsf{q}(j)} \in \mathbb{F}^{|\mathsf{q}(i)|}$.

Simulator $\mathcal{S}_{\mathsf{Open}}$ (continued)

5. Solve $M_{\mathsf{q}(i)} \cdot \mathbf{x}^i = [\![x^i]\!]_{\mathsf{q}(i)}$ for $\mathbf{x}^i$. If there are no solutions, then send the message Abort to $\mathcal{A}$ (emulating an honest party's message) and halt; otherwise, continue.
6. Await the call to $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Agree}, \mathbf{x}^i, sid)$ from $\mathcal{A}$ and execute it honestly.
7. Set $x^i := \langle \boldsymbol{\lambda}^i_{\mathsf{q}(i)}, [\![x^i]\!]_{\mathsf{q}(i)} \rangle$.

Verify Await the call to $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\texttt{Verify}, sid)$ and execute it honestly. If $\mathcal{A}$ sends the message Abort then send the message Abort to $\mathcal{F}_{\mathsf{Open}}$ and halt. Otherwise, if `Abort` is true, then send the message Abort to $\mathcal{A}$ on behalf of an (emulated) honest party and send the message Abort to $\mathcal{F}_{\mathsf{Open}}$.

Figure 4.6: Simulator $\mathcal{S}_{\mathsf{Open}}$ for $\mathcal{F}_{\mathsf{Open}}$.

Proof. The simulator is given in Figure 4.6. It essentially acts as a relay between the real honest parties and $\mathcal{A}$, via $\mathcal{F}_{\mathsf{Open}}$. Thus there is nothing to do to extract inputs of corrupt parties.

The "inputs" of this protocol are share vectors that the environment provides to the honest parties ahead of time. Indeed, this is exactly what the functionality aims to achieve: the parties can generate secrets *outside* of this process, and check them *inside* it.

Note that while the *functionality* has an abort flag, which is slightly unconventional, this is merely a way of encoding that the adversary has behaved in such a way that means the functionality will eventually abort. The reason for modelling in this way is that the real protocol allows incorrect share vectors to be introduced at various points in the execution, and the parties can choose to complete the protocol without running the verification subroutine, which means that if the functionality were cut off without running it, the environment should observe invalid share vectors as were specified by the adversary.

□

4.4 Preprocessing Functionality

The functionality $\mathcal{F}_{\mathsf{Prep}}$, that extends the functionality $\mathcal{F}_{\mathsf{Open}}$, is given in Figure 4.7, which aims to capture all actively-secure LSSS-based MPC, that is, including the full-threshold SPDZ protocol and the variants we described in this paper for $\mathcal{Q}_2$ access structures.

Functionality $\mathcal{F}_{\mathsf{Prep}}$

This functionality extends $\mathcal{F}_{\mathsf{Open}}$. (See Section 2.2 for the definition of extension in this context.)

Mask On input $(\texttt{Mask}, i, id_r, sid)$, the functionality does the following:

1. If $i \in [n] \setminus A$, then sample $r \leftarrow \mathcal{U}(\mathbb{F})$; otherwise await the input r from $\mathcal{S}$.
2. Execute the macro $\texttt{Sample}(id_r)$.
3. Await a message Abort or OK from $\mathcal{S}$. If the message is OK and `Abort` is false, then for all $j \in [n] \setminus A$ send $[\![r]\!]_{\mathcal{P}_j}$ to $\mathcal{P}_j$ and send r to $\mathcal{P}_i$; otherwise, send the message Abort to all honest parties, and then halt.

Triples On input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$ the functionality does the following:

1. For $k = 1, \ldots, T$, the functionality does the following:
   a) Sample $a_k, b_k \leftarrow \mathcal{U}(\mathbb{F})$ and set $c_k := a_k \cdot b_k$.
   b) Execute the macro $\texttt{Sample}(id_{a_k})$, $\texttt{Sample}(id_{b_k})$ and $\texttt{Sample}(id_{c_k})$.
2. Await a message Abort or OK from $\mathcal{S}$. If the message is OK and `Abort` is false then for each $i \in [n] \setminus A$, send $([\![a_k]\!]_{\mathcal{P}_i}, [\![b_k]\!]_{\mathcal{P}_i}, [\![c_k]\!]_{\mathcal{P}_i})_{k=1}^{T}$ to $\mathcal{P}_i$; otherwise, send the message Abort to all honest parties, and then halt.

Internal procedure

Sample When an internal procedure calls $\texttt{Sample}(id_v)$, the functionality does the following:

1. Await shares $[\![v]\!]_{\mathcal{A}}$ from $\mathcal{S}$.
2. Retrieve v from memory, sample a vector $\mathbf{v} \leftarrow \mathcal{U}\left(\{\mathbf{v} \in \mathbb{F}^d : M_{\mathcal{A}} \cdot \mathbf{v} = [\![v]\!]_{\mathcal{A}} \wedge \langle \mathbf{t}, \mathbf{v} \rangle = v\}\right)$ to create $[\![v]\!] := M \cdot \mathbf{v}$, and then return this vector.

Figure 4.7: Preprocessing Functionality, $\mathcal{F}_{\mathsf{Prep}}$.

4.5 Arithmetic Black Box

The protocol Π_{Online} is given in Figure 4.8; then follows a proof that it UC-securely realizes the arithmetic black box $\mathcal{F}_{\mathsf{ABB}}$, given in Figure 2.14, in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.

Protocol Π_{Online}

This protocol is realized in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.

Initialize The parties do the following:

1. Agree on a session identifier sid.
2. Agree on the circuit to compute, with T multiplication gates and $M = \sum_{i \in [n]} M_i$ total inputs, where M_i is the number of inputs for $\mathcal{P}_i$.
3. Call an instance of $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)$.
4. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$ where T is the number of multiplication gates in the circuit. If $\mathcal{F}_{\mathsf{Prep}}$ sends the message Abort, then (locally) output $\perp$ and halt; otherwise continue.
5. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id_{r_k}, sid)$ for each $k \in M_i$, for each party $\mathcal{P}_i$. If $\mathcal{F}_{\mathsf{Prep}}$ sends the message Abort, then (locally) output $\perp$ and halt; otherwise continue.
6. Agree on a sharing of 1, $[\![1]\!]$.

Input For party $\mathcal{P}_i$ to provide input x,

1. The parties retrieve from memory the agreed mask $(r, [\![r]\!])$ where $\mathcal{P}_i$ holds r.
2. Party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, x - r, sid)$ so all parties obtain $x - r$.
3. The parties compute a new identifier id_x and set $[\![x]\!] := [\![r]\!] + (x - r) \cdot [\![1]\!]$.

Add To add secrets $[\![x]\!]$ and $[\![y]\!]$, the parties compute a new identifier id_z for the result and set $[\![z]\!] := [\![x]\!] + [\![y]\!]$.

Multiply To multiply secrets $[\![x]\!]$ and $[\![y]\!]$, the parties do the following:

1. Retrieve from memory one unused multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$.
2. Compute new identifiers id_r and id_s and set $[\![r]\!] := [\![x]\!] - [\![a]\!]$ and $[\![s]\!] := [\![y]\!] - [\![b]\!]$.
3. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_r, sid)$ and $(\texttt{Open}, 0, id_s, sid)$ to open r and s.
4. Compute a new identifier id_z and set $[\![z]\!] := r \cdot s \cdot [\![1]\!] + s \cdot [\![a]\!] + r \cdot [\![b]\!] + [\![c]\!]$.

Output to one To output a secret $[\![x]\!]$ to $\mathcal{P}_i$, the parties do the following:[a]

1. Retrieve from memory the agreed mask $(r, [\![r]\!])$ where $\mathcal{P}_i$ holds r.
2. Compute a new identifier id_ε and set $[\![\varepsilon]\!] := [\![x]\!] - [\![r]\!]$.
3. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_\varepsilon, sid)$ to open ε.
4. Party $\mathcal{P}_i$ computes $x := r + \varepsilon$.
5. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$. If $\mathcal{F}_{\mathsf{Prep}}$ sends the message OK, then $\mathcal{P}_i$ (locally) outputs x; otherwise, the parties (locally) output $\perp$ and halt.

Protocol Π_{Online} (continued)

Output to all To output a secret $[\![x]\!]$ to all parties, the parties do the following:

1. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$ and execute it honestly with $\mathcal{A}$. If $\mathcal{F}_{\mathsf{Prep}}$ sends the message OK, then continue; otherwise, (locally) output $\perp$ and halt.
2. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_x, sid)$.
3. Call $\mathcal{F}_{\mathsf{Prep}}$ input $(\texttt{Verify}, sid)$. If $\mathcal{F}_{\mathsf{Prep}}$ sends the message OK then all parties, then (locally) output x; otherwise, (locally) output $\perp$ and halt.

[a] An alternative method is to call $\mathcal{F}_{\mathsf{Prep}}$ with inputs $(\texttt{Verify}, sid)$ and then $(\texttt{Open}, i, id_x, sid)$.

Figure 4.8: Online Protocol, Π_{Online}.

Theorem 4.2. *The protocol Π_{Online} UC-securely realizes the functionality $\mathcal{F}_{\mathsf{ABB}}$ against a static, active adversary in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

Proof. The simulator is given in Figure 4.9 and the transcript is given in Figure 4.10.

Simulator $\mathcal{S}_{\mathsf{ABB}}$

Initialize Set `Abort` to true, call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Initialize}, \mathbb{F}, sid)$, and then do the following:

1. Agree on a session identifier sid with $\mathcal{A}$.
2. Agree on the circuit to compute with $\mathcal{A}$, with T multiplication gates and $M = \sum_{i\in[n]} M_i$ total inputs, where M_i is the number of inputs for $\mathcal{P}_i$.
3. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)$ from $\mathcal{A}$ and initialize a local instance.
4. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$ from $\mathcal{A}$ where T is the number of multiplication gates in the circuit, and execute it honestly with $\mathcal{A}$. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, then send the message Abort to $\mathcal{A}$ and halt.
5. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id_{r_k}, sid)$ for each $k \in M_i$, for each corrupt party $\mathcal{P}_i$ with $i \in A$, and execute it honestly with $\mathcal{A}$. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, then send the message Abort to $\mathcal{A}$ and halt.
6. Agree on a sharing of 1, $[\![1]\!]$, with $\mathcal{A}$.

Input For party $\mathcal{P}_i$ to provide input x,

If $i \in A$,

1. Retrieve from memory the mask $(r, [\![r]\!])$ generated in the local execution of $\mathcal{F}_{\mathsf{Prep}}$.
2. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, \varepsilon, sid)$ from $\mathcal{A}$.
   - Await the calls $(\texttt{Send}, \varepsilon^j, j, sid)$ from $\mathcal{A}$ for $j \in [n] \setminus A$; if $\varepsilon^j \neq \varepsilon^i$ for all j, set `Abort` to true and set $\varepsilon := \varepsilon^j$ for any j.
3. Set $[\![x]\!] := \varepsilon \cdot [\![1]\!] + [\![r]\!]$, set $x := \varepsilon + r$, and then compute a new identifier and call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Input}, i, id, x, sid)$.

Simulator $\mathcal{S}_{\mathsf{ABB}}$ (continued)

If $i \in [n] \setminus A$,

1. Retrieve from memory the agreed mask $(r, [\![r]\!])$ generated in the local execution of $\mathcal{F}_{\mathsf{Prep}}$.
2. Call the internal copy of $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, -r, sid)$ and send the appropriate outputs to $\mathcal{A}$.
3. Set $[\![x]\!] := [\![r]\!] - r \cdot [\![1]\!]$ and call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Input}, i, id, \bot, sid)$.

Add Compute a new identifier id_z, retrieve from memory the associated identifiers id_x and id_y, set $[\![z]\!] := [\![x]\!] + [\![y]\!]$, and call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Add}, id_x, id_y, id_z, sid)$.

Multiply

1. Retrieve from memory one unused multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$ generated in the execution of $\mathcal{F}_{\mathsf{Prep}}$.
2. Compute new identifiers id_r and id_s and set $[\![r]\!] := [\![x]\!] - [\![a]\!]$ and $[\![s]\!] := [\![y]\!] - [\![b]\!]$.
3. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_r, sid)$ and $(\texttt{Open}, 0, id_s, sid)$ from $\mathcal{A}$ and execute the procedures honestly.
4. Compute a new identifier id_z, set $[\![z]\!] := r \cdot s \cdot [\![1]\!] + s \cdot [\![a]\!] + r \cdot [\![b]\!] + [\![c]\!]$ and call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Multiply}, id_x, id_y, id_z, sid)$.

Output to one To open some secret with identifier id_x shared as $[\![x]\!]$ to a party $\mathcal{P}_i$,

If $i \in [n] \setminus A$,

1. Retrieve from memory the mask $(r, [\![r]\!])$ where $\mathcal{P}_i$ holds r.
2. Compute a new identifier id_ε and set $[\![\varepsilon]\!] := [\![x]\!] - [\![r]\!]$.
3. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_\varepsilon, sid)$ and shares $[\![\varepsilon]\!]_{\mathcal{A}}$ from $\mathcal{A}$.
4. Set $x := r + \langle \boldsymbol{\lambda}, [\![\varepsilon]\!] \rangle$.
5. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, or if $\texttt{Abort}$ is true, then send the message Abort to $\mathcal{F}_{\mathsf{ABB}}$; otherwise, continue.

If $i \in A$,

1. Retrieve from memory the mask $(r, [\![r]\!])$ where corrupt party $\mathcal{P}_i$ holds r.
2. Compute a new identifier id_ε and set $[\![\varepsilon]\!] := [\![x]\!] - [\![r]\!]$.
   - Call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Output}, i, id_x, sid)$ and await the output x.
   - Sample a vector $\mathbf{x} \leftarrow \mathcal{U}\left(\{\mathbf{x} \in \mathbb{F}^d : M \cdot \mathbf{x} = [\![\varepsilon]\!] \wedge \langle \mathbf{t}, \mathbf{x} \rangle = x - r\}\right)$ and set $[\![x - r]\!] = M \cdot \mathbf{x}$.
   - Send $[\![x - r]\!]_{\mathcal{P} \setminus \mathcal{A}}$ to $\mathcal{A}$.
3. Await a message Abort or OK from $\mathcal{A}$, and if the message is Abort then set $\texttt{Abort}$ to true. Then await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$. If the returned message is Abort or $\texttt{Abort}$ is true, then send the message Abort to $\mathcal{F}_{\mathsf{ABB}}$; otherwise, send the message OK to $\mathcal{F}_{\mathsf{ABB}}$.

Output to all

1. Await the call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$ from $\mathcal{A}$. If the returned message is Abort then send the message Abort to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{Prep}}$; otherwise, continue.
2. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, id_x, sid)$ from $\mathcal{A}$ and then do the following:

Simulator $\mathcal{S}_{\mathsf{ABB}}$ (continued)

- Call $\mathcal{F}_{\mathsf{ABB}}$ with input $(\texttt{Output}, 0, id, sid)$ to receive the output x.
- Sample a vector $\mathbf{x} \leftarrow \mathcal{U}\left(\{\mathbf{x} \in \mathbb{F}^d : M \cdot \mathbf{x} = [\![\varepsilon]\!] \wedge \langle \mathbf{t}, \mathbf{x} \rangle = x\}\right)$ and set $[\![x]\!] = M \cdot \mathbf{x}$.
- Send $[\![x]\!]_{\mathcal{P} \setminus \mathcal{A}}$ to $\mathcal{A}$.

3. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$ from $\mathcal{A}$ and execute it honestly with $\mathcal{A}$. If the returned message is Abort or $\texttt{Abort}$ is true, then send the message Abort to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{ABB}}$; otherwise, continue.

Figure 4.9: Simulator $\mathcal{S}_{\mathsf{ABB}}$ for $\mathcal{F}_{\mathsf{ABB}}$.

Procedure	From	To	Message
Initialize	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)_{i \in A}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
	$\mathcal{F}_{\mathsf{Prep}}$	$\mathcal{A}$	$([\![a_k]\!]_{\mathcal{A}}, [\![b_k]\!]_{\mathcal{A}}, [\![c_k]\!]_{\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$((\texttt{Mask}, i, id_{r_{i,k}}, sid)_{k=1}^{M_i})_{i \in [n]}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
	$\mathcal{F}_{\mathsf{Prep}}$	$\mathcal{A}$	$([\![r_{i,k}]\!]_{\mathcal{A}})_{k=1}^{M_i})_{i \in [n]}$
	$\mathcal{A}$	$\mathcal{S}$	$[\![1]\!]$
Input	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Broadcast}, \varepsilon, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
Add	n/a	n/a	n/a
Multiply	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Open}, 0, id_r, sid)$
	$\mathcal{S}$	$\mathcal{A}$	$\{[\![r]\!]_k : \rho(k) \in [n] \setminus A \text{ and } k \in \mathsf{q}(A)\}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Open}, 0, id_s, sid)$
	$\mathcal{S}$	$\mathcal{A}$	$\{[\![s]\!]_k : \rho(k) \in [n] \setminus A \text{ and } k \in \mathsf{q}(A)\}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
Output To All	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Verify}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Open}, 0, id, sid)$
	$\mathcal{S}$	$\mathcal{A}$	$\{[\![x]\!]_k : \rho(k) \in [n] \setminus A \text{ and } k \in \mathsf{q}(A)\}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Verify}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
Output To One	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Verify}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Open}, 0, id, sid)$
	$\mathcal{S}$	$\mathcal{A}$	$\{[\![\varepsilon]\!]_k : \rho(k) \in [n] \setminus A \text{ and } k \in \mathsf{q}(A)\}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	$(\texttt{Verify}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}$	Abort/OK

Figure 4.10: Transcript for Π_{Online}.

It is clear from the transcript that the only difficulty is to ensure the following: $\mathcal{S}$ manages to extract corrupt parties' inputs; that the broadcasted value ε by the simu-

lator when providing inputs for emulated honest parties is indistinguishable from the broadcast in a real execution; and that the simulator can send a share vector for the *correct* final output instead of the simulated output. The rest of the interactions are calls to $\mathcal{F}_{\mathsf{Prep}}$ and are identical to a real-world execution as $\mathcal{S}$ emulates a real instantiation of the functionality.

Firstly, the ability of $\mathcal{S}$ to extract is ensured by the fact that since $\mathcal{S}$ executed $\mathcal{F}_{\mathsf{Prep}}$ locally, it knows the mask and can compute the corrupt party's (implicit) input. If $\mathcal{A}$ "broadcasts" different values to different honest parties then they abort later in the protocol, so the simulator sets a flag and aborts when the check in the real protocol would occur. Secondly, the mask r is uniform and unknown to $\mathcal{A}$ (or indeed $\mathcal{Z}$) so the distributions of $x+r$ and r, where x is the real honest party's input, are perfectly indistinguishable. Thirdly, note that every share held by $\mathcal{A}$ is established from share vectors generated by $\mathcal{F}_{\mathsf{Prep}}$ (i.e. $\mathcal{S}$) and the agreed sharing of $[\![1]\!]$, which means that $\mathcal{S}$ knows all the shares $\mathcal{A}$ would hold if it behaved honestly. Now, since $\mathcal{A}$ is unqualified, a set of shares held by $\mathcal{A}$ for a given secret provides no information on the secret, which means $\mathcal{S}$ can always replace shares for emulated honest parties so that $\mathcal{A}$ sees the output of $\mathcal{S}$'s choosing; thus it can modify the share vectors to encode the correct outputs of $\mathcal{F}_{\mathsf{ABB}}$ instead of the simulated outputs.

Hence the simulation is perfect, and so no environment can distinguish between worlds. □

4.6 Reactive Computation

In some applications, additional forms of preprocessing are useful, such as random bits when using MPC for circuit garbling [LPSY15] (discussed in detail in Chapter 8). Moreover, the exact circuit to be computed may not be known ahead of time, so the execution of an arithmetic black-box does not model the desired computation. In order to evaluate continuously – called *reactive* computation – an extended form of $\mathcal{F}_{\mathsf{Prep}}$, denoted by $\mathcal{F}_{\mathsf{RPrep}}$, is given in Figure 4.11, and a protocol Π_{RPrep} realizing it is given in Figure 4.12. The functionality $\mathcal{F}_{\mathsf{RPrep}}$ is essentially the same as the functionality $\mathcal{F}_{\mathsf{MPC}}$ given in [LPSY15, KY18] and offers essentially all of the commands of $\mathcal{F}_{\mathsf{ABB}}$, but involves shares of secrets explicitly.

The purpose Π_{RPrep} here is only to show that it is possible to realize $\mathcal{F}_{\mathsf{RPrep}}$, as it is an essential ingredient for circuit garbling in Chapter 8. However, the method by which this is achieved in Π_{RPrep} – namely, by extending the black box $\mathcal{F}_{\mathsf{Prep}}$ – is not necessarily

the optimal way of obtaining $\mathcal{F}_{\mathsf{RPrep}}$.

Notice that unlike $\mathcal{F}_{\mathsf{ABB}}$, no procedure **Output** is given: instead, the notion of "opening" and later verifying is retained, which more closely resembles real-world execution of the preprocessing protocol. The procedure for generating a random shared bit via these steps was given in [DKL+13].

Functionality $\mathcal{F}_{\mathsf{RPrep}}$

This functionality extends $\mathcal{F}_{\mathsf{Prep}}$ in Figure 4.7.

Input On input $(\texttt{Input}, i, id, x, sid)$ from party $\mathcal{P}_i$, or $\mathcal{S}$ if $i \in A$, and $(\texttt{Input}, i, id, \perp, sid)$ from all other parties, execute $\texttt{Sample}(id)$ and for all $i \in [n] \setminus A$, send $[\![x]\!]_{\mathcal{P}_i}$ to $\mathcal{P}_i$.

Add On input $(\texttt{Add}, id_x, id_y, id_z, sid)$ from all honest parties and $\mathcal{S}$, await $[\![x]\!]_{\mathcal{P}_i}$ and $[\![y]\!]_{\mathcal{P}_i}$ from each honest party $i \in [n] \setminus A$, compute $[\![z]\!] := [\![x]\!]_{\mathcal{P}_i} + [\![y]\!]_{\mathcal{P}_i}$ and send $[\![z]\!]_{\mathcal{P}_i}$ to $\mathcal{P}_i$ for all $i \in [n] \setminus A$.

Multiply On input $(\texttt{Multiply}, id_x, id_y, id_z, sid)$ from all honest parties and $\mathcal{S}$, for all $i \in [n]$ await shares $[\![x]\!]_{\mathcal{P}_i}$ and $[\![y]\!]_{\mathcal{P}_i}$ from $\mathcal{P}_i$, or from $\mathcal{S}$ if $i \in A$, compute $z := \langle \boldsymbol{\lambda}, [\![x]\!] \rangle \cdot \langle \boldsymbol{\lambda}, [\![y]\!] \rangle$, execute $\texttt{Sample}(id_z)$, and for all $i \in [n] \setminus A$, send $[\![z]\!]_{\mathcal{P}_i}$ to $\mathcal{P}_i$.

Random Element On input $(\texttt{RElt}, id_r, sid)$, sample $r \leftarrow \mathcal{U}(\mathbb{F})$, execute $\texttt{Sample}(id_r)$ and await a message Abort or OK from $\mathcal{S}$. If the message is OK, then for all $i \in [n] \setminus A$ send $[\![r]\!]_{\mathcal{P}_i}$ to $\mathcal{P}_i$; otherwise, send the message Abort to all honest parties, and then halt.

Random Bit On input $(\texttt{RBit}, id_b, sid)$, sample $b \leftarrow \mathcal{U}(\{0,1\})$, execute $\texttt{Sample}(id_b)$, and await a message Abort or OK from $\mathcal{S}$. If the message is OK, then for all $i \in [n] \setminus A$ send $[\![b]\!]_{\mathcal{P}_i}$ to $\mathcal{P}_i$; otherwise, send the message Abort to all honest parties, (locally) output $\perp$, and then halt.

Figure 4.11: Reactive Preprocessing Functionality, $\mathcal{F}_{\mathsf{RPrep}}$.

Protocol Π_{RPrep}

This protocol is realized in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.

Initialize The parties do the following:

1. Agree on a session identifier sid.
2. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Initialize}, \Gamma, aux_i, sid)$.
3. Set $[\![1]\!]$ to be any sharing of 1.

$\mathcal{F}_{\mathsf{Prep}}$ subroutines

Send `[-]` For $\mathcal{P}_i$ to send x to $\mathcal{P}_j$, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Send}, x, j, sid)$.

Broadcast `[<]` To broadcast x, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, x, sid)$.

Open `[>]` To open a secret with identifier id to $\mathcal{P}_i$, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, i, id, sid)$.

Open `[*]` To open a secret with identifier id to all parties, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id, sid)$.

Verify The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$.

Masks To generate masks, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id, sid)$ several times, for each $i \in [n]$. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, then the parties abort.

Triples To generate T triples, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$ where T is the batch-size. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, then the parties abort.

Π_{Online} subroutines

Input For party $\mathcal{P}_i$ to provide input x,

1. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id, sid)$ so that they obtain $(r, [\![r]\!])$ where $\mathcal{P}_i$ holds r.
2. Party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, x - r, sid)$ so all parties obtain $x - r$.
3. The parties compute a new identifier id_x and set $[\![x]\!] := [\![r]\!] + (x - r) \cdot [\![1]\!]$.

Add To add secrets $[\![x]\!]$ and $[\![y]\!]$, the parties compute a new identifier id_z for the result and set $[\![z]\!] := [\![x]\!] + [\![y]\!]$.

Multiply To multiply secrets $[\![x]\!]$ and $[\![y]\!]$, the parties do the following:

1. Compute new identifiers id_a, id_b, and id_c and call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_a, id_b, id_c), sid)$ to obtain $([\![a]\!], [\![b]\!], [\![c]\!])$.
2. Compute new identifiers id_r and id_s and set $[\![r]\!] := [\![x]\!] - [\![a]\!]$ and $[\![s]\!] := [\![y]\!] - [\![b]\!]$.
3. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_r, sid)$ and $(\texttt{Open}, 0, id_s, sid)$ to open r and s.
4. Compute a new identifier id_z and set $[\![z]\!] := r \cdot s \cdot [\![1]\!] + s \cdot [\![a]\!] + r \cdot [\![b]\!] + [\![c]\!]$.

Additional commands

Random Bits To generate T shared bits, the parties do the following:

1. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{\frac{3}{2} \cdot T}, sid)$ to obtain $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])_{k=1}^{\frac{3}{2} \cdot T}$.
2. For each $k = T + 1$ to $\frac{3}{2} \cdot T$, relabel $[\![a_k]\!]$ as $[\![d_{2(k-T)-1}]\!]$ and $[\![b_k]\!]$ as $[\![d_{2(k-T)}]\!]$ and discard $[\![c_k]\!]$.

Protocol Π_{RPrep} (continued)

3. For each $k = 1$ to T,
   a) Compute new identifiers id_{r_k} and id_{s_k} and set $[\![r_k]\!] := [\![d_k]\!] - [\![a_k]\!]$ and $[\![s_k]\!] := [\![d_k]\!] - [\![b_k]\!]$.
   b) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_{r_k}, sid)$ and $(\texttt{Open}, 0, id_{s_k}, sid)$.
   c) Compute a new identifier id_{e_k} and set $[\![e_k]\!] := r_k \cdot s_k \cdot [\![1]\!] + s_k \cdot [\![a_k]\!] + r \cdot [\![b_k]\!] + [\![c_k]\!]$.
   d) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, 0, id_{e_k}, sid)$.
   e) Compute a new identifier id_{b_k} and set $[\![b_k]\!] := \frac{1}{2} \cdot (\frac{1}{\sqrt{e_k}} \cdot [\![d_k]\!] + 1)$.
4. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$.
5. If $\mathcal{F}_{\mathsf{Prep}}$ sent the message OK, then $\{[\![b_k]\!]\}_{k=1}^{T}$; otherwise, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, \text{Abort}, sid)$.

Random Element The parties $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_a, id_b, id_c), sid)$ to obtain $([\![a]\!], [\![b]\!], [\![c]\!])$ and output $[\![a]\!]$.

Figure 4.12: Reactive Preprocessing Protocol, Π_{RPrep}.

Theorem 4.3. *The protocol Π_{RPrep} UC-securely realizes the functionality $\mathcal{F}_{\mathsf{RPrep}}$ against a static, active adversary in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

Proof. The simulator is given in Figure 4.13 but the transcript is omitted as it is very similar to the transcript provided in the proof of Theorem 4.2.

Simulator $\mathcal{S}_{\mathsf{RPrep}}$

Initialize

1. Agree on a session identifier sid with $\mathcal{A}$.
2. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)$ from $\mathcal{A}$ on behalf of each corrupt party and send $(\texttt{Initialize}, \Gamma, sid)$ to $\mathcal{F}_{\mathsf{RPrep}}$.
3. Set $[\![1]\!]$ to be any sharing of 1.

$\mathcal{F}_{\mathsf{Prep}}$ subroutines

Send`[-]` Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Send}, x, j, sid)$ from $\mathcal{A}$, forward the message to $\mathcal{F}_{\mathsf{RPrep}}$, and relay any response back to $\mathcal{A}$.

Broadcast`[<]` Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, \varepsilon, sid)$ from $\mathcal{A}$, forward the message to $\mathcal{F}_{\mathsf{RPrep}}$, and relay any response back to $\mathcal{A}$.

Open`[>]` Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, id, i, sid)$ from $\mathcal{A}$, forward the message to $\mathcal{F}_{\mathsf{RPrep}}$, and relay any response back to $\mathcal{A}$.

Open`[⋈]` Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, id, sid)$ from $\mathcal{A}$, forward the message to $\mathcal{F}_{\mathsf{RPrep}}$, and relay any response back to $\mathcal{A}$.

Simulator $\mathcal{S}_{\mathsf{RPrep}}$ (continued)

Triples Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$ from $\mathcal{A}$, forward the message to $\mathcal{F}_{\mathsf{RPrep}}$, and relay any response back to $\mathcal{A}$.

Masks Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id, sid)$ from $\mathcal{A}$, forward the message to $\mathcal{F}_{\mathsf{RPrep}}$, and relay any response back to $\mathcal{A}$. Additionally, for any $i \in A$ store mask value r sent by $\mathcal{A}$ in the interaction.

Verify The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$.

Π_{Online} subroutines

Input If $\mathcal{P}_i$ is to provide input,

If $i \in A$,

1. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id, sid)$ and execute it using the local instance of $\mathcal{F}_{\mathsf{Prep}}$.
2. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, \varepsilon, sid)$, retrieve from memory the mask r, and send the command $(\texttt{Input}, i, \varepsilon + r, sid)$ to $\mathcal{F}_{\mathsf{RPrep}}$ and $(\texttt{Input}, i, \perp, sid)$ on behalf of all corrupt parties $\mathcal{P}_j$ where $j \in A \setminus \{i\}$.
3. Compute a new identifier id_x and set $[\![x]\!] \coloneqq [\![r]\!] + \varepsilon \cdot [\![1]\!]$, and when $\mathcal{F}_{\mathsf{RPrep}}$ executes $\texttt{Sample}(id_x)$, send $[\![x]\!]_{\mathcal{A}}$ to $\mathcal{F}_{\mathsf{RPrep}}$.

If $i \in [n] \setminus A$,

- Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id, sid)$ and execute it using the local instance of $\mathcal{F}_{\mathsf{Prep}}$ and send the message $(\texttt{Input}, i, \perp, sid)$ to $\mathcal{F}_{\mathsf{RPrep}}$ for each $i \in A$.
- Retrieve from memory the mask r and send $-r$ to $\mathcal{A}$ to emulate the broadcast via $\mathcal{F}_{\mathsf{Prep}}$.
- Compute a new identifier id_x and set $[\![x]\!] \coloneqq [\![r]\!] - r \cdot [\![1]\!]$.

Add To add x and y, the simulator computes $[\![z]\!] \coloneqq [\![x]\!] + [\![y]\!]$.

Multiply To multiply secrets $[\![x]\!]$ and $[\![y]\!]$,

1. Compute new identifiers id_a, id_b, and id_c and await a call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_a, id_b, id_c), sid)$ from $\mathcal{A}$.
2. Compute new identifiers id_r and id_s and set $[\![r]\!] \coloneqq [\![x]\!] - [\![a]\!]$ and $[\![s]\!] \coloneqq [\![y]\!] - [\![b]\!]$.
3. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, id_r, sid)$ and $(\texttt{Open}, id_s, sid)$ and execute the protocols honestly.
4. Compute a new identifier id_z and set $[\![z]\!] \coloneqq r \cdot s \cdot [\![1]\!] + s \cdot [\![a]\!] + r \cdot [\![b]\!] + [\![c]\!]$ and call $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Multiply}, id_x, id_y, id_z, sid)_{i \in A}$ and when $\mathcal{F}_{\mathsf{RPrep}}$ executes $\texttt{Sample}(id_)$, send the vector $[\![z]\!]_{\mathcal{A}}$.

Additional commands

Random Bits To generate T shared bits, the parties do the following:

1. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{\frac{3}{2} \cdot T}, sid)$ to obtain $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])_{k=1}^{\frac{3}{2} \cdot T}$.
2. For each $k = T + 1$ to $\frac{3}{2} \cdot T$, relabel $[\![a_k]\!]$ as $[\![d_{2(k-T)-1}]\!]$ and $[\![b_k]\!]$ as $[\![d_{2(k-T)}]\!]$ and discard $[\![c_k]\!]$.
3. For each $k = 1$ to T,

Simulator $\mathcal{S}_{\mathsf{RPrep}}$ (continued)

a) Set $[\![r_k]\!] := [\![d_k]\!] - [\![a_k]\!]$.
b) Set $[\![s_k]\!] := [\![d_k]\!] - [\![b_k]\!]$.
c) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, [\![r_k]\!], sid)$ and $(\texttt{Open}, [\![s_k]\!], sid)$.
d) Set $[\![e_k]\!] := r_k \cdot s_k \cdot [\![1]\!] + s_k \cdot [\![a_k]\!] + r \cdot [\![b_k]\!] + [\![c_k]\!]$.
e) Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Open}, [\![e_k]\!], sid)$.
f) Set $[\![b_k]\!] := \frac{1}{2} \cdot (\frac{1}{\sqrt{e_k}} \cdot [\![d_k]\!] + 1)$.

4. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Verify}, sid)$ and execute it honestly with $\mathcal{A}$. If $\mathcal{F}_{\mathsf{Prep}}$ returns the message Abort then send the message Abort to $\mathcal{F}_{\mathsf{RPrep}}$ and halt; otherwise, $\{[\![b_k]\!]\}_{k=1}^{T}$; otherwise, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid)$.

Random Element The parties $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_a, id_b, id_c), sid)$ to obtain $([\![a]\!], [\![b]\!], [\![c]\!])$ and output $[\![a]\!]$.

Figure 4.13: Simulator $\mathcal{S}_{\mathsf{RPrep}}$ for $\mathcal{F}_{\mathsf{RPrep}}$.

A subtle, technical point in the simulation is that because $\mathcal{F}_{\mathsf{Prep}}$ is being extended and the simulator just relays information between $\mathcal{A}$ and $\mathcal{F}_{\mathsf{RPrep}}$ for commands to $\mathcal{F}_{\mathsf{Prep}}$, the simulator does not know masks for inputs, or the secret values of the triples. However, the environment *does* know these secrets, as the final outputs of the honest parties include shares that will allow $\mathcal{Z}$ to reconstruct the secrets (since it will hold shares for all honest parties and $\mathcal{A}$). Thus in order to be able to simulate (specifically, to extract inputs from $\mathcal{A}$ to pass on to $\mathcal{F}_{\mathsf{RPrep}}$, and to provide messages on behalf of honest parties that do not reveal the fact that $\mathcal{S}$ is emulating their inputs), it is crucial that any masks and triples used in the real-world protocol to provide input and to multiply *in this realization* of $\mathcal{F}_{\mathsf{RPrep}}$ should *not* be output by the honest parties at the end of the whole execution.

This is achieved in the protocol by requiring that the procedures **Mask** and **Triples** be called every time **Input** and **Multiply** are called, instead of the parties generating these as a form of preprocessing. This ensures that the honest parties do *not* provide the shares of the masks and triples that are used up in the execution as part of their final output to the environment. This means the simulator can generate the shares of the masks and triples *itself* by running the local instance of $\mathcal{F}_{\mathsf{Prep}}$, and can thus extract inputs, and can generate a transcript as if from real honest parties without knowing their secret inputs. Note that if $\mathcal{F}_{\mathsf{RPrep}}$ is later used to realize $\mathcal{F}_{\mathsf{ABB}}$ using Π_{Online}, say, then triples and masks may be generated as part of preprocessing.

As for **Random Bits**, the transcript for this part is exactly the same as for **Multiply** as given in the proof of security for the protocol Π_{Online}. Finally, note that since the only

way parties obtain any shares throughout the whole protocol is to take linear combinations of shares generated by $\mathcal{F}_{\mathsf{Prep}}$ (or $\mathcal{F}_{\mathsf{RPrep}}$), $\mathcal{S}$ knows what the adversary would compute, if it were to behave honestly, at all times. This means that in the execution of **Random Bits** and of **Random Element**, $\mathcal{S}$ can provide $\mathcal{F}_{\mathsf{RPrep}}$ with the correct shares during the execution of $\texttt{Sample}(id_b)$ and $\texttt{Sample}(id_r)$, respectively.

The remainder of the simulation is trivial as the simulator merely relays information between $\mathcal{A}$ and $\mathcal{F}_{\mathsf{RPrep}}$.

□

4.7 Modelling SPDZ

The main focus of this thesis concerns MPC for $\mathcal{Q}_2$ access structures. However, the aim is to describe the functionalities obliviously to the access structure (which is particularly useful for the protocols in Chapter 8), so it is helpful to understand how a protocol for a full-threshold access structure can be used to realize these functionalities. The purpose of this section is to give high-level intuition for this realization.

As outlined in Section 2.5.3, authentication for SPDZ is achieved with an IT MAC. To initialize $\mathcal{F}_{\mathsf{Open}}$ in this context, in the command $(\texttt{Initialize}, \Gamma, [\![\cdot]\!], sid)$, the symbol $[\![\cdot]\!]$ is interpreted as the additive secret-sharing MSP *and* information regarding the MAC being used. More specifically, the parties agree on an "authenticated" LSSS,

$$[\![\cdot]\!] \coloneqq \left([\![\cdot]\!]^{\mathtt{A}}, [\![\gamma(\cdot)]\!]^{\mathtt{A}}\right),$$

where now $[\![x]\!]$ is interpreted as the pair $\left([\![x]\!]^{\mathtt{A}}, [\![\gamma(x)]\!]^{\mathtt{A}}\right)$. Then the initialization involves party $\mathcal{P}_i$ sending the message

$$\left(\texttt{Initialize}, \Gamma, \left([\![1]\!]^{\mathtt{A}}_{\mathcal{P}_i}, [\![\alpha]\!]^{\mathtt{A}}_{\mathcal{P}_i}\right), sid\right)$$

to $\mathcal{F}_{\mathsf{Open}}$.

Opening to One Party When secrets are opened, only the first component of the share is sent; i.e. $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_i}$ but not $[\![\gamma(x)]\!]^{\mathtt{A}}_{\mathcal{P}_i}$, as opening both secrets would reveal the global MAC key and prohibit further computation with active security. The error-detection matrix N for additive secret-sharing is the all-zeroes matrix in $\mathbb{F}^{1\times n}$, which means that a party to whom a secret is revealed never aborts in the full-threshold setting. The intuition for this is that authentication is established via the MACs and not by the LSSS as in the $\mathcal{Q}_2$ setting. This "optimistic" opening of secrets is exactly what happens

in SPDZ, where a distinction is made between *opening* secrets, and providing *output*: the correctness check is deferred to the verification stage.

Opening to All Parties The map $\mathsf{q} : [n] \to [m]$ (where $m = n$ for additive secret-sharing) is defined as $\mathsf{q}(i) = [n]$ for all $i \in [n]$.[1] While the *functionality* checks that the shares being sent correspond to a valid share vector in the general setting (by computing a preimage of $[\![x]\!]^{\mathtt{A}}_{\mathsf{q}(i)}$ under $M_{\mathsf{q}(i)}$), for additive secret-sharing every set of shares is a valid share vector since the matrix is invertible, and therefore in the *protocol* the parties do not need to emulate this check.

Verification In addition to calling $\mathcal{F}_{\mathsf{Agreement}}$ to verify consistency of broadcasts, the parties execute a procedure for checking MACs without revealing the global MAC key, allowing computation to continue even after this check is performed. This was first given in [DKL+13] and is presented in Figure 4.14 for completeness.

[1] By considering the codomain of q to be $[2 \cdot n]$ instead of $[n]$, q can be viewed as encoding the fact that parties send shares of the secret but do not send shares of the associated MAC.

Subprotocol Π_{MACCheck}

Parties use this subprotocol to check that a set of *secret* values $\{[\![v_k]\!]\}_{k=1}^{t}$ are correct given a set $\{[\![v_k]\!]\}_{k=1}^{t+1}$ where $[\![v_{t+1}]\!]$ is a random mask to be discarded (for example, this can be taken from an unused triple). This subprotocol assumes there are running instances of $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{Commit}}$. Recall $[\![v]\!] := ([\![v]\!]^{\mathtt{A}}, [\![\gamma(v)]\!]^{\mathtt{A}})$ and that the parties also hold $[\![\alpha]\!]^{\mathtt{A}}$.

MAC Check

1. Agree on a new session identifier sid' for a new instance of $\mathcal{F}_{\mathsf{CoinFlip}}$.
2. Call a new instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\mathtt{Initialize}, \mathbb{F}^{t+1}, sid')$.
3. Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\mathtt{RElt}, sid')$ to obtain $\{r_k\}_{k=1}^{t+1}$ and compute $[\![v]\!] := \sum_{k=1}^{t+1} r_k \cdot [\![v_k]\!]$ and agree on a new identifier, id_v.
4. Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathtt{Open}, 0, id_v, sid)$.
5. Compute a new identifier id_z.
6. Each party $\mathcal{P}_i$ does the following:
   a) Set
   $$[\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i} := [\![\alpha]\!]^{\mathtt{A}}_{\mathcal{P}_i} \cdot v - \sum_{k=1}^{t+1} r_k \cdot [\![\gamma(v_k)]\!]^{\mathtt{A}}_{\mathcal{P}_i}.$$
   b) Call $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathtt{Commit}, i, [\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i}, sid)$ and $(\mathtt{Commit}, j, \perp, sid)$ for all $j \in [n] \setminus \{i\}$.
7. Await identifiers $\{id_{[\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i}}\}_{i\in[n]}$ from $\mathcal{F}_{\mathsf{Commit}}$.
8. Call $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathtt{Open}, i, id_{[\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i}}, sid)$ for all $i \in [n]$. If $\mathcal{F}_{\mathsf{Commit}}$ sends the message Abort, then each party calls $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathtt{Broadcast}, \mathsf{Abort}, sid)$, (locally) outputs $\perp$, and then halts.
9. Each party computes $z = \sum_{i\in[n]} [\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i}$. If every party computes $z = 0$ then they continue with $\{[\![v_k]\!]\}_{k=1}^{t}$ and discard $[\![v_{t+1}]\!]$, and if any party computes $z \neq 0$ then that party calls $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathtt{Broadcast}, \mathsf{Abort}, sid)$, (locally) outputs $\perp$, and then halts.

Figure 4.14: MAC-Checking Subprotocol, Π_{MACCheck}.

4.7.1 Errors on MACs

As a slight digression, the security of the MAC checking procedure is now analysed. It is important in the protocols in Chapter 5 that the adversary should be allowed to introduce errors onto the shares of the secret *or* on the MACs. In [DPSZ12], it was only shown that errors on the secrets could be detected, but it is trivial to show that errors on either the secret or the MAC (or both) still cause the honest parties to abort with overwhelming probability in the statistical security parameter, as demonstrated in the following lemma.

Lemma 4.1. *If the adversary cheats on* either *the MAC or on the secret, then the protocol* Π_{MACCheck} *aborts with probability at least* $1 - |\mathbb{F}|^{-1}$.

Recall that in SPDZ it is assumed that the field is of size $O(2^\sigma)$, and if not then the parties generate several MACs for every secret as outlined in Section 2.5.3, and so the probability that cheating is undetected is negligible in the statistical security parameter.

Proof. Let $\mathcal{P}_i$ be the honest party. (Recall that in the full-threshold setting, the adversary can corrupt all parties but one.) Since the shares of z are committed to before being opened, the (rushing) adversary cannot wait for the honest party to send its share and in return send its negative. Indeed, the only way to cheat without detection is to introduce errors in such a way that $z = 0$ holds.

Suppose the adversary introduces errors $\{\varepsilon_k\}_{k=1}^{t+1}$ on the shares of $\{v_k\}_{k=1}^{t+1}$ and $\{\delta_k\}_{k=1}^{t+1}$ on the shares of $\{\gamma(v_k)\}_{k=1}^{t+1}$ so that

$$\sum_{j\in[n]} [\![v_k]\!]^{\mathtt{A}}_{\mathcal{P}_j} = v_k + \varepsilon_k \qquad \text{and} \qquad \sum_{j\in[n]} [\![\gamma(v_k)]\!]^{\mathtt{A}}_{\mathcal{P}_j} = \gamma(v_k) + \delta_k.$$

Then since the random coefficients $\{r_k\}_{k=1}^{t+1}$ are not known before Π_{MACCheck} is executed, if the protocol does not abort then these errors must satisfy

$$0 = \alpha \cdot \left(v + \sum_{k=1}^{t+1} r_k \cdot \varepsilon_k \right) - \left(\gamma(v) + \sum_{k=1}^{t+1} r_k \cdot \delta_k \right) = \alpha \cdot \left(\sum_{k=1}^{t+1} r_k \cdot \varepsilon_k \right) - \left(\sum_{k=1}^{t+1} r_k \cdot \delta_k \right).$$

This means that choosing the correct errors is equivalent to guessing the MAC key, which can only be done correctly with probability at most $|\mathbb{F}|^{-1}$. □

4.7.2 $\mathcal{F}_{\mathsf{Prep}}$ with MACs

For realizing $\mathcal{F}_{\mathsf{Prep}}$ with MACs, it is necessary to alter the sampling procedure as follows.

Sample When an internal procedure calls `Sample`(id_v), the functionality does the following:

1. Await shares $[\![v]\!]^{\mathtt{A}}_{\mathcal{A}}$ and $[\![\gamma(v)]\!]^{\mathtt{A}}_{\mathcal{A}}$ from $\mathcal{S}$.
2. Retrieve v and α from memory, sample shares $\{[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}, [\![\gamma(x)]\!]^{\mathtt{A}}_{\mathcal{P}_i}\}_{i\in[n]\setminus A} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $v = \sum_{i\in[n]} [\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}$ and $\alpha \cdot v = \sum_{i\in[n]} [\![\gamma(v)]\!]^{\mathtt{A}}_{\mathcal{P}_i}$, and (locally) return the pair $([\![v]\!]^{\mathtt{A}}, [\![\gamma(v)]\!]^{\mathtt{A}})$.

Notice that this procedure does not allow errors to be introduced on MACs. This is because the simulator always has the opportunity to abort after **Sample** is executed (as

shown in the description of $\mathcal{F}_{\mathsf{Prep}}$). One could alternatively define $\mathcal{F}_{\mathsf{Prep}}$ to allow errors to be introduced during **Sample**, and then argue that any time Π_{MACCheck} is executed later will abort if the adversary introduced errors on the MACs. This is the approach that is taken in Section 5.4.

4.7.3 Viewing MACs as Part of the MSP

An alternative way to interpret the initialization of $\mathcal{F}_{\mathsf{Open}}$ with IT MACs is as an MSP with secret matrix M as follows:

$$\begin{array}{c} \mathcal{P}_1 \\ \mathcal{P}_2 \\ \vdots \\ \mathcal{P}_n \\ \hline \mathcal{P}_1 \\ \vdots \\ \mathcal{P}_{n-1} \\ \mathcal{P}_n \end{array} \left(\begin{array}{cccc|ccc} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ \hline 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \\ \alpha & \alpha & \cdots & \alpha & -1 & \cdots & -1 \end{array}\right) \cdot \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \\ \mathbf{x}_{n+1} \\ \vdots \\ \mathbf{x}_{2n-1} \end{pmatrix} = \begin{pmatrix} [\![x]\!]^{\mathtt{A}}_{\mathcal{P}_1} \\ \vdots \\ [\![x]\!]^{\mathtt{A}}_{\mathcal{P}_n} \\ [\![\gamma(x)]\!]^{\mathtt{A}}_{\mathcal{P}_1} \\ \vdots \\ [\![\gamma(x)]\!]^{\mathtt{A}}_{\mathcal{P}_n} \end{pmatrix}$$

with target vector $\mathbf{t} = (1, \ldots, 1, 0, \ldots, 0)^\top \in \mathbb{F}^{2n-1}$. Here, $\mathbf{x} \in \mathbb{F}^{2n-1}$ is the randomness vector satisfying $\langle \mathbf{x}, \mathbf{t} \rangle = x$, and the vector on the right is the complete vector of shares. The (secret) error-detecting matrix N can be written as

$$N = \begin{pmatrix} \alpha & \alpha & \cdots & \alpha & -1 & \cdots & -1 \end{pmatrix}.$$

However, interpreting the LSSS in this way interferes with other aspects of the description of $\mathcal{F}_{\mathsf{Open}}$. For instance, if a secret is opened to one party, this party cannot perform the error-detection procedure (premultiplying the share vector by N) because the matrix N is secret.

Chapter 5

Outsourcing MPC preprocessing

This chapter is based on work published at IMACC 2017 under the title When It's All Just Too Much: Outsourcing MPC-Preprocessing *[SSW17] and was joint work with Peter Scholl and Nigel Smart. A protocol generalizing these results, to allow preprocessing generated by a set of parties under a $\mathcal{Q}_2$ access structure, has also been given, employing the error-detection results from Chapter 3.*

This chapter Since the preprocessing is by far the costliest part of multi-party computation (MPC), it is natural to wonder if this work may be outsourced. The benefits are clear: for example, low-powered devices without access to strong entropy sources (which are required for cryptography) can simply *receive* the preprocessed data and use it to execute the cheap online protocol. It is shown in this chapter that there is a very natural way to achieve this for SPDZ preprocessing. In fact, any form of secret-sharing that makes use of linear message authentication codes (MACs) or another form of linear authentication for a linear secret-sharing scheme (LSSS) is amenable to the transformation described here.

5.1 Overview

The idea is very simple: one set of parties, which will be denoted by $\mathcal{R}$, produces preprocessing – that is, the correlated randomness of Beaver's circuit randomization technique (see Section 2.5.3) – and sends it to a second set of parties, which will be denoted by $\mathcal{Q}$, so that the latter set can realize the arithmetic black box $\mathcal{F}_{\mathsf{ABB}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model by executing the protocol Π_{Online} given in Chapter 4. The strength of the universal com-

posability (UC) framework comes to the fore in this chapter, as the secure realization of $\mathcal{F}_{\mathsf{ABB}}$ for the parties in $\mathcal{Q}$ is immediate.

The motivation behind outsourcing is that while the preprocessing phase requires expensive public-key cryptography (PKC) in the full-threshold setting, the online computation is much less costly. Thus a set of low-powered devices can outsource the heavy cost of preprocessing and then run the low-cost online phase to evaluate circuits. Indeed, it makes sense for protocols in the preprocessing model to be designed with a view to outsourcing in this way, as without outsourcing it may be more efficient to evaluate the circuit directly.

The idea of the protocols in this chapter is that the parties executing the preprocessing will send their data over a network to the parties that want to use it. The challenging part is to prove in the UC framework that the second set of parties will abort if any of the parties producing preprocessing cheat when sending their data. Beyond the resharing itself, transferring the preprocessing requires essentially no additional communication cost because an inexpensive checking procedure can be performed. The randomness required to verify the correctness of the outsourced data can also be outsourced, if indeed the use case is low-powered clients without access to good sources of entropy.[1] Following the protocol Π_{Online}, whenever parties in $\mathcal{Q}$ require a preprocessed data-item, they can request $\mathcal{R}$ to provide one. This may be necessary if the low-powered devices have little memory, but this data could equally well be sent all at once.

All that is required is a methodology for translating $[\![\cdot]\!]^{\mathcal{R}}$ sharings into $[\![\cdot]\!]^{\mathcal{Q}}$ sharings with active security. The principal idea of the protocol is for each party $\mathcal{R}_i$ in $\mathcal{R}$ to reshare their share and send the shares to the parties in $\mathcal{Q}$. Two different scenarios will be considered:

- from a $\mathscr{Q}_2$ access structure to a $\mathscr{Q}_2$ access structure; and
- from full-threshold to full-threshold.

Because the preprocessed data from the source set of parties is shared using an LSSS with a form of authentication, there is a low-cost method for the parties in $\mathcal{Q}$ to verify that no errors were introduced during resharing: in the case of a $\mathscr{Q}_2$ access structure on the parties in $\mathcal{R}$ this is achieved by the self-authentication of the LSSS; in the

[1] Low-powered devices often generate random data by using a pseudorandom function (PRF) to extend a short seed, since high-entropy sources may not be available. Doing so drains power resources, which is a crucial factor for protocols for battery-powered clients.

full-threshold setting where authentication is achieved through the use of information-theoretic (IT) MACs, if parties cheat in what they send then the MAC will be incorrect with high probability, and so the standard MAC verification procedure may be performed to check this. The main differences between these two scenarios is caused by this difference in the method of authentication; the rest of the protocol is essentially the same. An optimization for the full-threshold to full-threshold case will also be explored in which parties in $\mathcal{R}$ do not need to communicate with *all* parties in $\mathcal{Q}$.

In the literature, there are several works that show how to outsource preprocessing from a set of low-powered clients, typically through the use of a third party. In his thesis, Yan Huang [Hua12] explained how to use a "partially" trusted third party to generate preprocessed data for two computing parties to garble and evaluate circuits. Demmler et al. [DSZ14] investigated how to generate preprocessed data on low-powered devices cheaply through the use of hardware tokens. In their protocol known as *Chameleon*, Riazi et al. [RWT+18] also looked at how to use a third party to generate correlated randomness for both garbled circuits and LSSS-based MPC. The work in this chapter can be seen as using MPC to replace the third party.

5.2 Preliminaries

5.2.1 Network

The complete set of parties is denoted by $\mathcal{P}$ and parties are indexed by $[n]$. The set $\mathcal{P}$ is considered to be the union of two parts, $\mathcal{R}$ and $\mathcal{Q}$, indexed by R and Q respectively (so $R, Q \subseteq [n]$ are not necessarily disjoint). To avoid confusion, parties in $\mathcal{R}$ will always be indexed by the letter i, and parties in $\mathcal{Q}$ by the letter j. The variables $n^{\mathcal{R}}$ and $n^{\mathcal{Q}}$ denote the number of parties in $\mathcal{R}$ and $\mathcal{Q}$, respectively. The set $A \subseteq R \cup Q$ denotes the indexing set of corrupt parties in the complete network.

It is assumed that there is a complete network of authenticated channels amongst the parties in $\mathcal{R}$, and amongst the parties in $\mathcal{Q}$, and that each party in $\mathcal{R}$ is connected via a secure channel to each party in $\mathcal{Q}$. This last network assumption is replaced with the notion of a *secure cover* in the full-threshold to full-threshold case, defined and discussed in Section 5.4.

5.2.2 Preprocessing Functionality

The process of "outsourcing" MPC preprocessing is defined as a protocol realizing the functionality $\mathcal{F}_{\mathsf{Prep}}$ for a set of parties $\mathcal{Q}$ given the functionality $\mathcal{F}_{\mathsf{Prep}}$ provided to a set of parties $\mathcal{R}$. In the language of the UC framework, a protocol $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ will be said to realize $\mathcal{F}_{\mathsf{Prep}}$ for $\mathcal{Q}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, where in the protocol the oracle $\mathcal{F}_{\mathsf{Prep}}$ is used to generate preprocessing for $\mathcal{R}$. To avoid ambiguity, the oracle is called $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ and the functionality the outsourcing protocol will realize is called $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$.

Much of the remainder of this thesis deals with how to generate preprocessing efficiently for a set of parties in a $\mathscr{Q}_2$ access structure. For full-threshold MPC, $\mathcal{F}_{\mathsf{Prep}}$ can more-or-less be realized by the SPDZ protocol as described in the original paper [DPSZ12] and outlined in Section 2.5.3. This chapter demonstrates why modelling of $\mathcal{F}_{\mathsf{Prep}}$ in terms of concrete realization of the "authenticated dictionary" via linear authentication of secrets in an LSSS as described in Chapter 4 is useful: if $\mathcal{F}_{\mathsf{Prep}}$ were expressed in terms of identifiers and not in terms of an LSSS then there would be no way for parties to manipulate the shares, which is necessary in order for the "resharing" to take place. Because the shares are explicit, the access structure on $\mathcal{Q}$ is determined entirely by the LSSS used in the resharing.

The focus here is restricted to input and output masks and Beaver triples, rather than other forms of correlated randomness such as shared squares and shared bits, as the security of resharing these other kinds follows immediately from the security of the first kinds.

5.2.3 Types of Secret-Sharing

Secrets shared amongst parties in $\mathcal{R}$ are denoted by $[\![\cdot]\!]^{\mathcal{R}}$, which is notation that encompasses all the information regarding the monotone span program (MSP) used. That is, there is a matrix $M^{\mathcal{R}} \in \mathbb{F}^{m^{\mathcal{R}}\times d^{\mathcal{R}}}$, a target vector $\mathbf{t}^{\mathcal{R}}$, and a row map $\rho^{\mathcal{R}}$. Analogous notation is used for secrets shared amongst parties in $\mathcal{Q}$.

5.3 Outsourcing $\mathscr{Q}_2$ to $\mathscr{Q}_2$

Before the protocol is given, first the resharing and verification procedures are described and their correctness is justified. Then the protocol $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ for outsourcing is presented, followed by a proof that it can be used to realize $\mathcal{F}_{\mathsf{Prep}}$ for the parties in $\mathcal{Q}$.

5.3.1 Correctness

Resharing

The protocol involves sharings in $\mathcal{Q}$ of sharings in $\mathcal{R}$: in the protocol, parties in $\mathcal{Q}$ will hold vectors $[\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}} \in \mathbb{F}^{m^{\mathcal{R}} \cdot m^{\mathcal{Q}} \times 1}$ as

$$[\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}} = \begin{pmatrix} [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_1} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_{n^{\mathcal{Q}}}} \end{pmatrix} = \begin{pmatrix} [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{1} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{m^{\mathcal{Q}}} \end{pmatrix} = \begin{pmatrix} [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_1}]\!]^{\mathcal{Q}}_{1} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_{n^{\mathcal{R}}}}]\!]^{\mathcal{Q}}_{1} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_1}]\!]^{\mathcal{Q}}_{m^{\mathcal{Q}}} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_{n^{\mathcal{R}}}}]\!]^{\mathcal{Q}}_{m^{\mathcal{Q}}} \end{pmatrix} = \begin{pmatrix} [\![[\![v]\!]^{\mathcal{R}}_{1}]\!]^{\mathcal{Q}}_{1} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}_{m^{\mathcal{R}}}]\!]^{\mathcal{Q}}_{1} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}_{1}]\!]^{\mathcal{Q}}_{m^{\mathcal{Q}}} \\ \vdots \\ [\![[\![v]\!]^{\mathcal{R}}_{m^{\mathcal{R}}}]\!]^{\mathcal{Q}}_{m^{\mathcal{Q}}} \end{pmatrix}.$$

Given a matrix $A \in \mathbb{F}^{m^{\mathcal{R}} \times k}$ with $k \geq 1$, the vector $A^\top \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}$ is defined as $[\![A^\top \cdot [\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}$; i.e. $(\mathbf{1}^\top \otimes A^\top) \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}$ where $\mathbf{1} \in \mathbb{F}^{m^{\mathcal{Q}} \times 1}$.

Each party reshares every component of the share vector they hold as a $[\![\cdot]\!]^{\mathcal{Q}}$ sharing and distributes the shares. More concretely, party $\mathcal{R}_i$ computes $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}$ and sends the shares to the parties in $\mathcal{Q}$. To obtain a single sharing of the same secret, each party $\mathcal{Q}_j$ in $\mathcal{Q}$ writes the shares $\left\{[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\right\}_{i \in R}$ as a share vector $[\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$, and then the parties in $\mathcal{Q}$ compute

$$\boldsymbol{\lambda}^\top \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}} = [\![\boldsymbol{\lambda}^\top \cdot [\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}} = [\![v]\!]^{\mathcal{Q}}$$

where party $\mathcal{Q}_j$ computes $[\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} = [\![\boldsymbol{\lambda}^\top \cdot [\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$.

Verifying a Resharing

To reduce communication costs for the parties in $\mathcal{Q}$, verification of the resharing is performed by computing a random linear combination of secrets and performing a single check. To minimize the amount of randomness parties in $\mathcal{Q}$ need to generate, either these parties can execute the checks individually instead of in batches, or they can use a trusted source of randomness such as a blockchain, lottery or random beacon[2], or they can receive randomness from the parties in $\mathcal{R}$ via a sort of "outsourced $\mathcal{F}_{\mathsf{CoinFlip}}$" by initializing an agreement functionality $\mathcal{F}_{\mathsf{Agreement}}$ and doing the following:

1. Each party $\mathcal{R}_i$ in $\mathcal{R}$ samples r_i and sends a commitment to r_i to all $\mathcal{Q}_j$ in $\mathcal{Q}$.

[2] See, for example, the Interoperable Randomness Beacon project [KBPB19].

2. When all commitments have been received, the parties in $\mathcal{R}$ send the decommitments.
3. Each party in $\mathcal{Q}$ fixes $r := \sum_{i \in R} r_i$.
4. The parties in $\mathcal{Q}$ call $\mathcal{F}_{\mathsf{Agreement}}$ with input $(\mathtt{Agree}, r, sid)$.

The naïve way of resharing would be for each party $\mathcal{R}_i$ to compute an additive sharing $[\![v]\!]^{\mathtt{A}}_{\mathcal{R}_i} := \langle \boldsymbol{\lambda}_{\mathcal{R}_i}, [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i} \rangle$, to compute a sharing $[\![[\![v]\!]^{\mathtt{A}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}$ and distribute shares to parties in $\mathcal{Q}$, who could then set $[\![v]\!]^{\mathcal{Q}} := \sum_{i=1}^{n^{\mathcal{R}}} [\![[\![v]\!]^{\mathtt{A}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}$. However, in the $\mathcal{Q}_2$ setting the error-detection in the $[\![\cdot]\!]^{\mathcal{R}}$ sharing is lost; in the protocol presented, the parties in $\mathcal{R}$ instead reshare *every component* of the share vector they hold, and then the parties in $\mathcal{Q}$ recombine as outlined above, but now they can additionally compute a sharing of the error vector $\boldsymbol{\varepsilon} \in \mathbb{F}^{m^{\mathcal{R}} - d^{\mathcal{R}}}$ for the secret as follows:

$$N^{\mathcal{R}} \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}} = [\![N^{\mathcal{R}} \cdot [\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}} = [\![\boldsymbol{\varepsilon}]\!]^{\mathcal{Q}} \in \mathbb{F}^{m^{\mathcal{Q}} \cdot (m^{\mathcal{R}} - d^{\mathcal{R}})}$$

where $N^{\mathcal{R}}$ is the cokernel of the MSP matrix M, as described in Chapter 3. To verify that this vector is $\mathbf{0}$, the parties in $\mathcal{Q}$ combine the (secret-shared) entries of the error vector into a single field element by taking a random linear combination, and check it is equal to zero. This check can be further amortized by combining error vectors from several resharings together in a random linear combination first. The subprotocol for verification is given in Figure 5.1.

Subprotocol $\Pi_{\mathsf{ErrorCheck}}$

Error Check The parties check the correctness of a set $\{[\![v_k]\!]^{\mathcal{Q}}\}_{k=1}^{t}$ with errors $\{[\![\boldsymbol{\varepsilon}^k]\!]^{\mathcal{Q}}\}_{k=1}^{t}$:

1. Compute a new session identifier sid and call a new instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\mathtt{Initialize}, \mathbb{F}, sid)$.
2. Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\mathtt{RElt}, sid)$ a total of $(m^{\mathcal{R}} - d^{\mathcal{R}}) \cdot t$ times to obtain $\left\{ \{r_{k,l}\}_{l=1}^{m^{\mathcal{R}} - d^{\mathcal{R}}} \right\}_{k=1}^{t}$.
3. For each $k \in [t]$, retrieve from memory the shares of $[\![\boldsymbol{\varepsilon}^k]\!]^{\mathcal{Q}}$ and let $[\![\boldsymbol{\varepsilon}^k_l]\!]^{\mathcal{Q}}$ be the sharing of the l^{th} component of $\boldsymbol{\varepsilon}^k$.
4. Compute $[\![\varepsilon]\!]^{\mathcal{Q}} := \sum_{k=1}^{t} \sum_{l=1}^{m^{\mathcal{R}} - d^{\mathcal{R}}} r_{k,l} \cdot [\![\boldsymbol{\varepsilon}^k_l]\!]^{\mathcal{Q}}$ and agree on a new identifier id_{ε}.
5. Call $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ with input $(\mathtt{Open}, 0, id_{\varepsilon}, sid_{\mathcal{Q}})$ followed by $(\mathtt{Verify}, sid_{\mathcal{Q}})$. If $\varepsilon \neq 0$ or $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ returned the message Abort, then send the message Abort to all other parties, (locally) output $\perp$, and then halt; otherwise continue.

Figure 5.1: Error-Checking Subprotocol, $\Pi_{\mathsf{ErrorCheck}}$.

The security proof later relies on the assumption that the protocol $\Pi^{\mathcal{R} \to \mathcal{Q}}_{\mathsf{Outsource}}$ aborts if one or more invalid $[\![\cdot]\!]^{\mathcal{R}}$ sharings are generated or if one or more invalid $[\![\cdot]\!]^{\mathcal{Q}}$ sharings are generated, and hence the following lemma is required.

Lemma 5.1. *The probability that $\boldsymbol{\varepsilon}^k \neq \mathbf{0}$ for some $k \in [t]$ but that $\varepsilon = 0$ is at most $1/|\mathbb{F}|$. The probability that $N^{\mathcal{Q}} \cdot [\![v_k]\!]^{\mathcal{Q}} \neq \mathbf{0}$ for some $k \in [t]$ but that $N^{\mathcal{Q}} \cdot [\![\varepsilon]\!]^{\mathcal{Q}} = \mathbf{0}$ is at most $1/|\mathbb{F}|$.*

Proof. When ε is opened in the call to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ with input $(\mathtt{Open}, 0, id_\varepsilon, sid)$, error-detection is performed, meaning that once the resharing has taken place, the adversary cannot change the value of ε.

Since the multipliers used in the linear combination are not known before the resharing, it only holds that $\varepsilon = 0$ if the adversary chooses to add in error vectors $\{\boldsymbol{\delta}^k\}_{k=1}^t = \{(\boldsymbol{\delta}^k_l)_{l=1}^{m^{\mathcal{R}}-d^{\mathcal{R}}}\}_{k=1}^t$ so that $\sum_{k=1}^t \sum_{l=1}^{m^{\mathcal{R}}-d^{\mathcal{R}}} r_{k,l} \cdot \boldsymbol{\delta}^k_l = 0$. This is equivalent to fixing all $\{\boldsymbol{\delta}^k_l\}$ but one and then guessing the correct way to fix the final component, which can be done with probability at most $1/|\mathbb{F}|$, the size of the set from which the multipliers were sampled.

For the second probability, note that again the adversary must guess the random multipliers ahead of time to generate one or more invalid $[\![\cdot]\!]^{\mathcal{Q}}$ sharings and correct them in the combination. The probability of this is again at most $1/|\mathbb{F}|$ by the same argument. □

For simplicity, the field is assumed $O(2^\sigma)$ so that this probability is negligible in the statistical security parameter. When this is not the case, similar repetition techniques as for sacrifice and MACs can be used, as described in Section 2.5.3.

5.3.2 Security

The goal is to show that if the set of parties $\mathcal{R} \cup \mathcal{Q}$ is provided the functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ and $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ and executes the protocol $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$, then this "looks the same" to the parties in $\mathcal{Q}$ as a functionality $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$.

Protocol $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$

This protocol is realized in the $\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}, \mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$-hybrid model and makes use of the subprotocol $\Pi_{\mathsf{ErrorCheck}}$ (where $\mathcal{F}_{\mathsf{CoinFlip}}$ is used).

Initialize

1. The parties agree on a session identifier, sid; the parties in $\mathcal{Q}$ agree on a session identifier $sid_{\mathcal{Q}}$ and the parties in $\mathcal{R}$ agree on a session identifier $sid_{\mathcal{R}}$.
2. The parties in $\mathcal{Q}$ call an instance of $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Initialize}, \Gamma^{\mathcal{Q}}, [\![\cdot]\!]^{\mathcal{Q}}, sid_{\mathcal{Q}})$. This instance is denoted by $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$.
3. The parties in $\mathcal{R}$ call an instance of $\mathcal{F}_{\mathsf{Prep}}$ with input $(\mathtt{Initialize}, \Gamma^{\mathcal{R}}, [\![\cdot]\!]^{\mathcal{R}}, sid_{\mathcal{R}})$. This instance is denoted by $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$. The parties in $\mathcal{R}$ send the error-detecting matrix $N^{\mathcal{R}}$ to the parties in $\mathcal{Q}$ and any recombination vector $\boldsymbol{\lambda}^{\mathcal{R}}$.[a]

Protocol $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ (continued)

Mask To compute a mask for $\mathcal{Q}_j$, the parties do the following:

1. The parties in $\mathcal{R}$ compute new identifiers id_a, id_b and id_c and call $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple},(id_a,id_b,id_c),sid_{\mathcal{R}})$ to obtain some $[\![a]\!]^{\mathcal{R}}$ and relabel id_a as $id_{r^{(j)}}$ and the shares as $[\![r^{(j)}]\!]^{\mathcal{R}}$.
2. The parties in $\mathcal{R}$ execute $\texttt{FeedValue}(id_{r^{(j)}})$.
3. Each $\mathcal{R}_i \in \mathcal{R}$ sends their share vector $[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}_i}$ to $\mathcal{Q}_j$.
4. Party $\mathcal{Q}_j$ computes $N^{\mathcal{R}} \cdot [\![r^{(j)}]\!]^{\mathcal{R}}$ and if it is equal to $\mathbf{0}$ then it computes $r^{(j)} := \langle \boldsymbol{\lambda}^{\mathcal{R}}, [\![r^{(j)}]\!]^{\mathcal{R}} \rangle$; otherwise $\mathcal{Q}_j$ calls $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ with input $(\texttt{Broadcast},\mathsf{Abort},sid_{\mathcal{Q}})$, (locally) outputs $\perp$, and then halts.

Triple To compute T triples for parties in $\mathcal{Q}$, the parties do the following:

1. Parties in $\mathcal{R}$ compute new identifiers id_{a_k}, id_{b_k} and id_{c_k} for $k \in [T]$ and call $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple},(id_{a_k},id_{b_k},id_{c_k})^T_{k=1},sid_{\mathcal{R}})$ to obtain triples $([\![a_k]\!]^{\mathcal{R}},[\![b_k]\!]^{\mathcal{R}},[\![c_k]\!]^{\mathcal{R}})^T_{k=1}$.
2. For each $v \in \{a_k,b_k,c_k\}^T_{k=1}$ the parties in $\mathcal{R}$ execute $\texttt{FeedValue}(id_v)$.

Feed Value When the parties call $\texttt{FeedValue}(id_v)$, they do the following:

1. Each $\mathcal{R}_i \in \mathcal{R}$ does the following:
   a) For each $k \in \rho^{-1}(\{i\})$,
      i. Compute a sharing $[\![[\![v]\!]^{\mathcal{R}}_k]\!]^{\mathcal{Q}}$.
      ii. For each $j \in Q$, send $[\![[\![v]\!]^{\mathcal{R}}_k]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$ to $\mathcal{Q}_j$ over a secure channel.
2. Each $\mathcal{Q}_j \in \mathcal{Q}$ does the following:
   a) Concatenate the shares $\{[\![[\![v]\!]^{\mathcal{R}}_k]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\}^{m^{\mathcal{R}}}_{k=1}$ into a share vector $[\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$.
   b) Set $[\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} := \langle \boldsymbol{\lambda}^{\mathcal{R}}, [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} \rangle$.
   c) Store a vector of secret-shared values $[\![\boldsymbol{\epsilon}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} := N^{\mathcal{R}} \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} \in \mathbb{F}^{m^{\mathcal{Q}} \cdot (m^{\mathcal{R}} - d^{\mathcal{R}}) \times 1}$.

Error Check To check the correctness of a set $\{[\![v_k]\!]^{\mathcal{Q}}\}^t_{k=1}$, the parties execute $\Pi_{\mathsf{ErrorCheck}}$. If the parties did not abort, then they (locally) output shares of all secrets.

$\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ subroutines

The parties call $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ with the appropriate input.

[a] For example, they can take the first recombination vector for the first minimally-qualified set, where both orderings are lexicographic.

Figure 5.2: Outsourcing Protocol, $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$.

Theorem 5.1. *The protocol $\Pi^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ UC-securely realizes the functionality $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ with statistical security parameter $\sigma := \log|\mathbb{F}|$ against a static, active adversary in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$, $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$-hybrid model.*

Proof of Theorem 5.1. The simulator is given in Figure 5.3.

Simulator $\mathcal{S}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$

Initialize The simulator and adversary agree on a session identifier sid and then the simuator does the following:

1. The simulator agrees on a session identifier with $\mathcal{A}$.
2. Await the call to $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Initialize}, \mathcal{Q}, [\![\cdot]\!]^{\mathcal{Q}}, sid_{\mathcal{Q}})$ from $\mathcal{A}$ for each $j \in \boldsymbol{Q} \cap A$ and forward the commands to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$.
3. Await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Initialize}, \mathcal{R}, [\![\cdot]\!]^{\mathcal{R}}, sid_{\mathcal{R}})$ from $\mathcal{A}$ for each $i \in \boldsymbol{R} \cap A$ and initialize a local instance to be the oracle $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$.

Mask When the parties are to produce a mask for $\mathcal{Q}_j$ in $\mathcal{Q}$,

1. Await a call to $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_a, id_b, id_c), sid_{\mathcal{R}})$ and execute the procedure from the local instace of $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with $\mathcal{A}$ to obtain some $[\![r^{(j)}]\!]^{\mathcal{R}}$ with identifier $id_{r^{(j)}}$.
2. Execute $\texttt{SFeedValue}([\![r^{(j)}]\!]^{\mathcal{R}})$ with $\mathcal{A}$.
3. If $j \in \boldsymbol{Q} \setminus A$, then await a set of shares $[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}$ from $\mathcal{A}$; if $j \in \boldsymbol{Q} \cap A$, then send $[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}$ to $\mathcal{A}$.
4. Send the message $(\texttt{Mask}, j, id_{r^{(j)}}, sid_{\mathcal{Q}})$ to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ and then send $[\![j^{()}]\!]_{\mathcal{A}}$ to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ when it executes $\texttt{Sample}(id_{r^{(j)}})$. If $j \in \boldsymbol{Q} \setminus A$, and if $N \cdot [\![r^{(j)}]\!] = \boldsymbol{0}$ then send the message Abort to $\mathcal{A}$ and to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$, and otherwise continue. If $j \in \boldsymbol{Q} \cap A$, then await a message Abort or OK; forward this message to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$, and if the message is Abort then halt; otherwise, continue.
5. Execute **Error Check**.

Triples

1. Await the call to $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid_{\mathcal{R}})$ from $\mathcal{A}$ and execute the procedure from the local instance of $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with $\mathcal{A}$ to obtain $\{[\![a_k]\!]^{\mathcal{R}}, [\![b_k]\!]^{\mathcal{R}}, [\![c_k]\!]^{\mathcal{R}}\}_{k=1}^{T}$.
2. For each $v \in \{a_k, b_k, c_k\}_{k=1}^{T}$, execute $\texttt{SFeedValue}(id_v)$ with $\mathcal{A}$.
3. Execute **Error Check**.

Feed Value The macro $\texttt{SFeedValue}(id_v)$, simulating $\texttt{FeedValue}()$, is defined as follows:

1. For each $i \in \boldsymbol{R} \setminus A$, sample a (complete) share vector $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}$ and send $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}$ to $\mathcal{A}$.
2. For each $j \in \boldsymbol{Q} \setminus A$,
   a) For each $i \in \boldsymbol{R} \cap A$, await a share vector $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$ from $\mathcal{A}$.
   b) Combine $\{[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\}_{i\in\boldsymbol{R}}$ into a share vector $[\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$ and set $[\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} := \boldsymbol{\lambda}^{\mathcal{R}} \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$.
   c) Set $[\![\boldsymbol{\varepsilon}]\!]^{\mathcal{Q}}_{\mathcal{P}_j} := N^{\mathcal{Q}} \cdot [\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$.

   Finally, sample a vector $\mathbf{v} \leftarrow \mathcal{U}\left(\{\mathbf{v} \in \mathbb{F}^{d^{\mathcal{Q}}} : M^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}} \cdot \mathbf{v} = [\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}\}\right)$, set $[\![v]\!]^{\mathcal{Q}} := M^{\mathcal{Q}} \cdot \mathbf{v}$, return the share vector $[\![v]\!]^{\mathcal{Q}}$ locally, and send $[\![v]\!]^{\mathcal{Q}}_{\mathcal{A}}$ to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ when it executes $\texttt{Sample}(id_v)$.

Error Check The simulator awaits the call $(\texttt{RElt}, sid)$ to $\mathcal{F}_{\mathsf{CoinFlip}}$ and then executes $\Pi_{\mathsf{ErrorCheck}}$ honestly with $\mathcal{A}$; if honest parties would abort then send the message Abort to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$.

$\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$ subroutines

Relay commands between $\mathcal{A}$ and $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$.

Figure 5.3: Simulator $\mathcal{S}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ for $\mathcal{F}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$.

The transcript produced during the protocol execution in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{Open}}^{\mathcal{Q}}$, $\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$-hybrid world is given in Figure 5.4.

Procedure	From	To	Message
Initialize	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Commit}}$	$(\texttt{Initialize}, \mathcal{P}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}^{\mathcal{Q}}$	$(\texttt{Initialize}, \Gamma^{\mathcal{Q}}, [\![\cdot]\!]^{\mathcal{Q}}, sid_{\mathcal{Q}})_{j \in Q \cap A}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$(\texttt{Initialize}, \Gamma^{\mathcal{R}}, [\![\cdot]\!]^{\mathcal{R}}, sid_{\mathcal{R}})_{i \in R \cap A}$
Mask	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$(\texttt{Triple}, (id_a, id_b, id_c), sid_{\mathcal{R}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$([\![a]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![b]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![c]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	Abort/OK
	$\mathcal{S}$	$\mathcal{A}$	$[\![[\![a]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{S}$	$[\![[\![\widetilde{a}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}$
Triples	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid_{\mathcal{R}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$([\![a_k]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![b_k]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![c_k]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	Abort/OK
	$\mathcal{S}$	$\mathcal{A}$	$([\![[\![a_k]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}, [\![[\![b_k]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}, [\![[\![c_k]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{S}$	$([\![[\![\widetilde{a_k}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}, [\![[\![\widetilde{b_k}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}, [\![[\![\widetilde{c_k}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}})_{k=1}^{T}$
Error Check	$\mathcal{A}$	$\mathcal{F}_{\mathsf{CoinFlip}}$	$(\texttt{RElt}, sid)$
	$\mathcal{F}_{\mathsf{CoinFlip}}$	$\mathcal{A}$	$\{\{r_{k,l}\}_{k=1}^{T}\}_{l=1}^{m^{\mathcal{R}}-d^{\mathcal{R}}}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}^{\mathcal{Q}}$	$(\texttt{Open}, 0, id_{\varepsilon}, sid_{\mathcal{Q}})_{j \in Q \setminus A}$
	$\mathcal{S}$	$\mathcal{A}$	Abort/OK
	$\mathcal{A}$	$\mathcal{S}$	Abort/OK

Figure 5.4: Transcript for $\Pi_{\mathsf{Outsource}}^{\mathcal{R}\to\mathcal{Q}}$.

The honest parties in both $\mathcal{R}$ and $\mathcal{Q}$ have no inputs as the preprocessed data is determined by the random tapes. Additionally, in the ideal world, the honest parties in $\mathcal{R}$ have no input, no output, and do nothing in the execution of the functionality $\mathcal{F}_{\mathsf{Prep}}^{\mathcal{Q}}$, and thus can be perfectly emulated by the simulator. Moreover, all calls to the functionalities are distributed identically in both worlds as the simulator emulates these oracles honestly.

It only remains to show that the reshares sent from $\mathcal{S}$ to $\mathcal{A}$ are consistent with the final outputs of honest parties in $\mathcal{Q}$ in the ideal world (as chosen by $\mathcal{F}_{\mathsf{Prep}}^{\mathcal{Q}}$) despite $\mathcal{S}$ having to sample share vectors to provide input to $\mathcal{F}_{\mathsf{Prep}}^{\mathcal{Q}}$ when the procedure **Sample** is called. The difficulty comes from the fact that $\mathcal{S}$ does *not* see the reshares $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}$, and yet must somehow deduce a set of shares $[\![[\![v]\!]^{\mathcal{R}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}$ for honest parties to send to $\mathcal{F}_{\mathsf{Prep}}^{\mathcal{Q}}$.

The idea is that it does not matter what shares the simulator chooses. Since at least one honest party in $\mathcal{R}$ reshares its share vector, the final shares held by honest parties

are not known to $\mathcal{A}$ in the protocol execution. Thus, provided the final share vector of shares output by honest parties in the ideal world $[\![v']\!]_{\mathcal{Q}\setminus\mathcal{A}}$ concatenated with the shares the adversary would compute $[\![v]\!]_{\mathcal{Q}\cap\mathcal{A}}$ forms a valid share vector, the distribution of shares output by the parties in both worlds is the same.

To see this explicitly, recall that in the simulation, $\mathcal{S}$ samples

$$\mathbf{v} \leftarrow \mathcal{U}\left(\left\{\mathbf{v}\in\mathbb{F}^{d^{\mathcal{Q}}} : M^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}\cdot\mathbf{v} = [\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}\right\}\right),$$

sets $[\![v]\!]^{\mathcal{Q}} \coloneqq M^{\mathcal{Q}}\cdot\mathbf{v}$, and sends $[\![v]\!]^{\mathcal{Q}}_{\mathcal{A}}$ to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ when it executes $\texttt{Sample}(id_v)$. Next, the functionality uses these shares $[\![v]\!]_{\mathcal{A}}$ to sample a vector

$$\mathbf{v}' \leftarrow \mathcal{U}\left(\left\{\mathbf{v}\in\mathbb{F}^{d^{\mathcal{Q}}} : M^{\mathcal{Q}}_{\mathcal{A}}\cdot\mathbf{v} = [\![v]\!]_{\mathcal{A}}\right\}\right)$$

and then computes $[\![v']\!] \coloneqq M^{\mathcal{Q}}\cdot\mathbf{v}'$. The final share vector lies in the space given by

$$\begin{aligned}
&M^{\mathcal{Q}}\cdot\left(M^{\mathcal{Q}^{-1}}_{\mathcal{A}}\cdot\left(M^{\mathcal{Q}}\cdot\left({M^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}}^{-1}\cdot[\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}\right)\right)_{\mathcal{A}}\right)\\
=&M^{\mathcal{Q}}\cdot\left(M^{\mathcal{Q}^{-1}}_{\mathcal{A}}\cdot\left(M^{\mathcal{Q}}_{\mathcal{A}}\cdot\left(\mathbf{v}+\ker(M^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}})\right)\right)\right)\\
=&M^{\mathcal{Q}}\cdot\left(\mathbf{v}+\ker(M^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}})+\ker(M^{\mathcal{Q}}_{\mathcal{A}})\right)\\
=&M^{\mathcal{Q}}\cdot\mathbf{v}+M^{\mathcal{Q}}\cdot\ker(M^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}})+M^{\mathcal{Q}}\cdot\ker(M^{\mathcal{Q}}_{\mathcal{A}})
\end{aligned}$$

where $M^{\mathcal{Q}^{-1}}$ here denotes the preimage, *not* the inverse (which may not be well-defined). Vectors in the second summand are non-zero only on components owned by corrupt parties. Thus the total share vector generated by $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ may differ on corrupt parties' shares by a share vector supported by rows owned by $\mathcal{Q}\cap\mathcal{A}$, but this is still a valid share vector. Furthermore, vectors in the third summand are non-zero only on components owned by honest parties, and therefore the share vector plausibly shares any secret from the point of view of $\mathcal{A}$, which in the real world corresponds to the fact that there is at least one honest party in $\mathcal{R}$ that generates a share vector, so the shares of honest parties are unknown to $\mathcal{A}$.

Finally, observe that if $\mathcal{A}$ does not faithfully reshare the vector $[\![v]\!]_{\mathcal{R}\cap\mathcal{A}}$, then by Lemma 5.1, the honest parties abort with overwhelming probability in $\sigma = \log|\mathbb{F}|$. Additionally, if the share vector $[\![[\![v]\!]_{\mathcal{R}\cap\mathcal{A}}]\!]_{\mathcal{Q}\setminus\mathcal{A}}$ is invalid, then this is detected when opening the secret $[\![\varepsilon]\!]^{\mathcal{Q}}$. In either of these cases, $\mathcal{S}$ sends the message Abort to $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Prep}}$ and so $\mathcal{Z}$ does not see the erroneous share vectors. □

5.4 Outsourcing Full-Threshold to Full-Threshold

To avoid overloading the notation used for shares $[\![\cdot]\!]^{\mathcal{Q}}$ with a superscript $\mathtt{A}$ as has been used previously to indicate the use of additive secret-sharing, in this section, all secret sharing is assumed to be additive.

In the $\mathscr{Q}_2$ case, every party in $\mathcal{R}$ was required to reshare their shares under the LSSS of the parties in $\mathcal{Q}$. However additive resharing can be much more flexible: each party in $\mathcal{R}$ can reshare its summand additively to some proper subset of parties in $\mathcal{Q}$, and if all parties in $\mathcal{Q}$ compute the sum of all shares they receive, then they will still obtain an additive sharing of the original secret. In this case, secrecy (against the environment) will hold if and only if at least one honest party in $\mathcal{R}$ reshares to at least one honest party in $\mathcal{Q}$. The intuition behind this requirement is that if it does not hold, then every share held by parties in $\mathcal{R}$ is either known by $\mathcal{A}$ (if the party is corrupt), or each honest party in $\mathcal{R}$ sends only to corrupt parties in $\mathcal{Q}$, so $\mathcal{A}$ can reconstruct the share. Thus, knowing all shares in $\mathcal{R}$, $\mathcal{A}$ (and hence the distinguisher $\mathcal{Z}$) can compute the secret. This is problematic as it means the simulator cannot emulate the final outputs of real honest parties without $\mathcal{Z}$ observing a difference with high probability. Conversely, if this requirement *does* hold then at "worst" $\mathcal{A}$ obtains all reshares but one; then since all reshares are sampled uniformly at random so that they sum to shares in $\mathcal{R}$, and these shares are themselves sampled so that they sum to the secret, this set of shares is indistinguishable from a uniformly randomly sampled set. This trust assumption is formalized with the definition of *secure cover* $\{\mathcal{Q}^{(i)}\}_{i\in R}$ of $\mathcal{Q}$, which defines to which parties in $\mathcal{Q}$ the party $\mathcal{R}_i$ reshares.

Definition 5.1 (Secure Cover)**.** A set $\{\mathcal{Q}^{(i)}\}_{i\in R}$ of (not necessarily disjoint) subsets of $\mathcal{Q}$ is called a *secure cover* if the following conditions hold:

1. For every $i \in R$, $\mathcal{Q}^{(i)} \neq \varnothing$.
2. The set $\{\mathcal{Q}^{(i)}\}_{i\in R}$ is a cover for $\mathcal{Q}$, i.e. $\mathcal{Q} = \bigcup_{i\in R} \mathcal{Q}^{(i)}$.
3. Every party $\mathcal{Q}_j$ in $\mathcal{Q}^{(i)}$ is connected to the party $\mathcal{R}_i \in \mathcal{R}$ via a *secure* channel.
4. There is *at least one* pair $(\mathcal{R}_i, \mathcal{Q}^{(i)})$ where $\mathcal{R}_i$ is honest in $\mathcal{R}$, and $\mathcal{Q}^{(i)}$ contains *at least one* honest party in $\mathcal{Q}$.

The set $\{\boldsymbol{Q}^{(i)}\}_{i\in R}$ is defined to be the corresponding cover on indexing set $\boldsymbol{Q}$, and similarly for R. While it is always possible to define a secure cover by setting $\mathcal{Q}^{(i)} := \mathcal{Q}$

for all $i \in R$, there may be better (i.e. more communication-efficient) ways of defining it, such as in the following ways:

- If $\mathcal{R} \subseteq \mathcal{Q}$ then for each $i \in R$, one can define $\mathcal{Q}^{(i)}$ to be any subset of $\mathcal{Q}$ containing $\mathcal{R}_i$ and ensure that $\bigcup_{i \in R} \mathcal{Q}^{(i)} = \mathcal{Q}$. In this case, since at least one party $\mathcal{R}_i$ in $\mathcal{R}$ is honest, it is also honest in $\mathcal{Q}$, and so $\mathcal{R}_i$ is the honest party in $\mathcal{R}$ that "sends" the data to itself as the honest party in $\mathcal{Q}$.
- If $\mathcal{R}$ and $\mathcal{Q}$ are disjoint but each party in $\mathcal{Q}$ believes that some proper subset of parties in $\mathcal{R}$ contains at least one honest party then the cover can potentially be created respecting this knowledge, provided the trusted sets form a cover.

One can also consider the set of parties in $\mathcal{R}$ connected to each party in $\mathcal{Q}$. Let $\mathcal{R}^{(j)}$ denote the set of parties in $\mathcal{R}$ which are connected to party $\mathcal{Q}_j \in \mathcal{Q}$. Since $\mathcal{Q}^{(i)} \neq \varnothing$ for all $i \in R$, for every $i \in R$ there is at least one $j \in Q$ such that $\mathcal{R}^{(j)} \ni \mathcal{R}_i$, and hence $\left\{\mathcal{R}^{(j)}\right\}_{j \in Q}$ is a cover for $\mathcal{R}$. The last two properties are symmetrical, which means that in fact $\left\{\mathcal{R}^{(j)}\right\}_{j \in Q}$ is a secure cover for $\mathcal{R}$.

5.4.1 Modified Preprocessing Functionality

The functionality $\mathcal{F}_{\mathsf{Prep}}$ must be modified in a few superficial ways. The altered functionality is denoted by $\overline{\mathcal{F}}_{\mathsf{Prep}}$ and is given in Figure 5.5, and the reasons for the modifications are described below.

It is very important to notice that honest parties in $\mathcal{Q}$ use shares they receive in an entirely deterministic manner, and, as such, if some party $\mathcal{Q}_j \in \mathcal{Q}$ is honest but receives shares from *only* corrupt parties in $\mathcal{R}$, then $\mathcal{A}$ has complete control over this party's share. For this reason, hereafter such parties are deemed "effectively" corrupt and are called *effectively-corrupt honest parties*, which are added to the set of corrupt parties $\mathcal{A}$, which becomes the extended adversary set $\overline{\mathcal{A}}$. This means the functionality $\mathcal{F}_{\mathsf{Prep}}$ must be modified to allow for this form of corruption, although it does not significantly change the functionality except that the simulator will always choose the messages output by $\mathcal{F}_{\mathsf{Prep}}$ to these parties. Note that this type of corruption is not the same as passive corruption as $\mathcal{A}$ will *not* learn the inputs of such parties since the random masks used for inputs are generated by all parties in $\mathcal{R}$. In terms of the secure cover, an effectively-corrupt honest party is an honest party $\mathcal{Q}_j$ in $\mathcal{Q}$ for which $\mathcal{R}^{(j)} \setminus \mathcal{A} = \varnothing$.

In order to make the outsourcing more efficient, it is not necessary for parties to execute Π_{MACCheck} to verify the correctness of secrets sent across the network: instead, it

suffices for these checks to be combined with checks executed during the online phase of the protocol. Despite this, the ability of the simulator to cause the functionality to abort may be retained as the adversary in the real world can cause honest parties to abort at any time during the protocol execution. However, during **Sample**, the functionality now allows the simulator to introduce errors on secrets and so the honest parties in $\mathcal{Q}$ may end up with incorrect share vectors. It will be shown in the simulation that such errors can be "carried through" from the real execution into the ideal world. The online phase of the protocol remains secure since introducing errors during resharing is equivalent to introducing errors during the execution of the online phase, which is detected in the execution of Π_{MACCheck}.

Functionality $\overline{\mathcal{F}}_{\mathsf{Prep}}$

This functionality is identical to $\mathcal{F}_{\mathsf{Prep}}$ in Figure 4.7 except for the following modifications: Whenever outputs are to be sent to effectively-corrupt honest parties, allow the adversary to choose the outputs they receive.

Initialize Accept a cover $\{\mathcal{Q}^{(i)}\}_{i\in R}$ from all parties and $\mathcal{S}$ and set

$$\overline{\mathcal{A}} := \mathcal{A} \cup \left\{\mathcal{Q}_j \in \mathcal{Q} : \mathcal{R}^{(j)} \setminus \mathcal{A} = \varnothing\right\}.$$

Additionally, the simulator is allowed to choose shares of the MAC key not only for corrupt parties, but also for effectively-corrupt honest parties.

Sample In addition to accepting the vector of shares $([\![v]\!]^{\mathcal{Q}}_{\mathcal{A}}, [\![\gamma(v)]\!]^{\mathcal{Q}}_{\mathcal{A}})$, the functionality also receives shares for effectively-corrupt honest parties, and then does the following:

1. Await shares $[\![v]\!]^{\mathcal{Q}}_{\overline{\mathcal{A}}}$ and $[\![\gamma(v)]\!]^{\mathcal{Q}}_{\overline{\mathcal{A}}}$ and errors ε_v and $\varepsilon_{\gamma(v)}$ from $\mathcal{S}$.
2. Retrieve v and α from memory, sample shares $\{[\![v]\!]^{\mathcal{Q}}_{\mathcal{P}_i}, [\![\gamma(x)]\!]^{\mathcal{Q}}_{\mathcal{P}_i}\}_{i\in[n]\setminus\overline{A}} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $v = \sum_{i\in[n]} [\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \varepsilon_v$ and $\alpha \cdot v = \sum_{i\in[n]} [\![\gamma(v)]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \varepsilon_{\gamma(v)}$, and (locally) return $([\![v]\!]^{\mathtt{A}}, [\![\gamma(v)]\!]^{\mathtt{A}})$.

Figure 5.5: Modified Preprocessing Functionality, $\overline{\mathcal{F}}_{\mathsf{Prep}}$.

5.4.2 Correctness

Resharing

To reshare, each party $\mathcal{R}_i$ in $\mathcal{R}$ additively shares their share amongst the parties in $\mathcal{Q}^{(i)}$, i.e. samples $\{[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{j\in\boldsymbol{Q}^{(i)}} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{j\in\boldsymbol{Q}^{(i)}} [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} = [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}$. Then each party $\mathcal{Q}_j$ in $\mathcal{Q}$ sums all the shares they receive: $[\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} := \sum_{i\in R^{(j)}} [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$. Correctness holds by observing that $\bigcup_{i\in R} R \times \boldsymbol{Q}^{(i)} = \bigcup_{j\in\boldsymbol{Q}} R^{(j)} \times \boldsymbol{Q}$ so the limits can be switched below

to show that

$$\sum_{i\in R} [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i} = \sum_{i\in R}\sum_{j\in Q^{(i)}} [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} = \sum_{j\in Q}\sum_{i\in R^{(j)}} [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} = \sum_{j\in Q} [\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}.$$

During initialization, the global MAC key α is reshared as above. Then for any secret x that is shared as a pair $([\![x]\!]^{\mathcal{R}}, [\![\gamma(x)]\!]^{\mathcal{R}})$, both $[\![x]\!]^{\mathcal{R}}$ and $[\![\gamma(x)]\!]^{\mathcal{R}}$ are reshared. Thus the parties in $\mathcal{Q}$ hold secrets shared under the same global MAC key α.

Verification of Resharing

In contrast to the protocol outsourcing to a set of parties under an $\mathscr{Q}_2$ access structure, the protocol presented here does *not* involve an "error-checking" procedure. This is because if the parties in $\mathcal{R}$ are dishonest in sending their data, any MAC-checking procedure in the *online* phase will abort with high probability. (The checking procedure involves verifying the correctness of the MACs and was given in Figure 4.14 in Section 4.7.) This makes the protocol more straightforward, but the security analysis a little more complicated: shares must now be consistently erroneous in the ideal and real worlds. It will be shown that it is indeed possible to simulate.

A subtle point to be aware of is that while in the proof in the original SPDZ paper [DPSZ12] errors were permitted on secrets but *not* MACs, the adversary now has freedom to introduce errors on either. Thus a minor tweak to the proof of security of the online phase (regarding the MAC check) is required, but was dealt with in detail in Section 4.7 and so it is not explained here.

5.4.3 Security

The protocol is given in Figure 5.6 and the goal is to realize the functionality $\overline{\mathcal{F}}_{\mathsf{Prep}}$. The access structures $\Gamma_{\mathcal{R}}$ and $\Gamma_{\mathcal{Q}}$ are full-threshold. The algorithm for finding a secure cover is discussed in Section 5.5 and is used in the initialization of the protocol, but for now it suffices to know that the parties can find a cover.

Theorem 5.2. *The protocol $\overline{\Pi}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ UC-securely realizes the functionality $\overline{\mathcal{F}}_{\mathsf{Prep}}$ with perfect security against a static, active adversary in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{Commit}}$, $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$, $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$-hybrid model, assuming a secure cover of $\mathcal{Q}$ is given.*

Protocol $\overline{\Pi}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$

Initialize

1. The parties agree on a session identifier, sid; the parties in $\mathcal{Q}$ agree on a session identifier $sid_{\mathcal{Q}}$ and the parties in $\mathcal{R}$ agree on a session identifier $sid_{\mathcal{R}}$.
2. The parties execute the algorithm in Figure 5.10 to obtain a secure cover, $\{\mathcal{Q}^{(i)}\}_{i\in R}$.
3. The parties compute a new identifier id_a, then each $\mathcal{R}_i$ samples $[\![a]\!]^{\mathcal{R}}_{\mathcal{R}_i} \leftarrow \mathcal{U}(\mathbb{F})$ and they an instance of $\mathcal{F}_{\mathsf{Prep}}$ where $\mathcal{R}_i$ provides input $\left(\texttt{Initialize}, \Gamma_{\mathcal{R}}, \left([\![1]\!]^{\mathcal{R}}_{\mathcal{R}_i}, [\![a]\!]^{\mathcal{R}}_{\mathcal{P}_i}\right), sid_{\mathcal{R}}\right)$. This instance is denoted by $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$.
4. The parties in $\mathcal{R}$ execute $\texttt{FeedValue}(id_a)$.
5. The parties in $\mathcal{Q}$ call an instance of $\mathcal{F}_{\mathsf{Open}}$ where $\mathcal{Q}_j$ provides input $\left(\texttt{Initialize}, \Gamma_{\mathcal{Q}}, \left([\![1]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}, [\![a]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\right), sid_{\mathcal{Q}}\right)$. This instance is denoted by $\mathcal{F}^{\mathcal{Q}}_{\mathsf{Open}}$.

Mask To compute a mask for $\mathcal{Q}_j$, the parties do the following:

1. The parties in $\mathcal{R}$ compute new identifiers id_a, id_b and id_c and call $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_a, id_b, id_c), sid_{\mathcal{R}})$ to obtain some $[\![a]\!]^{\mathcal{R}}$ and relabel id_a as $id_{r^{(j)}}$ and the shares as $[\![r^{(j)}]\!]^{\mathcal{R}}$.
2. The parties in $\mathcal{R}$ execute $\texttt{FeedValue}(id_{r^{(j)}})$.
3. Each $\mathcal{R}_i$ sends their share $[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}_i}$ to $\mathcal{Q}_j$.

Triple To compute T triples for parties in $\mathcal{Q}$, the parties do the following:

1. Parties in $\mathcal{R}$ compute new identifiers id_{a_k}, id_{b_k} and id_{c_k} for $k \in [T]$ and call $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})^T_{k=1}, sid_{\mathcal{R}})$ to obtain triples $([\![a_k]\!]^{\mathcal{R}}, [\![b_k]\!]^{\mathcal{R}}, [\![c_k]\!]^{\mathcal{R}})^T_{k=1}$.
2. For each $v \in \{a_k, b_k, c_k\}^T_{k=1}$ the parties in $\mathcal{R}$ execute $\texttt{FeedValue}(id_v)$.

Feed Value When the parties call $\texttt{FeedValue}(id_v)$, where id_v is the identifier for a sharing $[\![v]\!]^{\mathcal{R}}$, they do the following on $[\![v]\!]^{\mathcal{R}}$ and then on $[\![\gamma(v)]\!]^{\mathcal{R}}$:

1. Each $\mathcal{R}_i \in \mathcal{R}$ does the following:
   a) Sample a sharing $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}$.
   b) For each $j \in \boldsymbol{Q}^{(i)}$, send $[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$ to $\mathcal{Q}_j$ over a secure channel.
2. Each $\mathcal{Q}_j \in \mathcal{Q}$ does the following:
   a) Await shares $\{[\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{i\in R^{(j)}}$.
   b) (Locally) output $[\![v]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} \coloneqq \sum_{i\in R^{(j)}} [\![[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$.

Figure 5.6: Optimized Outsourcing Protocol, $\overline{\Pi}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$.

Proof. The simulator is given in Figure 5.7.

Simulator $\overline{\mathcal{S}}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$

Initialize

1. Agree on a session identifiers sid, $sid_{\mathcal{Q}}$ and $sid_{\mathcal{R}}$ with $\mathcal{A}$.
2. Execute the algorithm in Figure 5.10 to obtain a secure cover, $\{\mathcal{Q}^{(i)}\}_{i\in R}$.
3. Compute a new identifier id_α and then on behalf of emulated honest parties, sample $\{[\![\alpha]\!]^{\mathcal{R}}_{\mathcal{R}_i}\}_{i\in R\setminus\overline{A}} \leftarrow \mathcal{U}(\mathbb{F})$; then await the call to $\mathcal{F}_{\mathsf{Prep}}$ with input $\left(\texttt{Initialize},\Gamma_{\mathcal{R}},\left([\![1]\!]^{\mathcal{R}}_{\mathcal{R}_i},[\![\alpha]\!]^{\mathcal{R}}_{\mathcal{P}_i}\right),sid_{\mathcal{R}}\right)$ from $\mathcal{A}$ and initialize a local instance to be the oracle $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$. Let $[\![\alpha]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\overline{\mathcal{A}}}$ be the shares of the MAC key held by (emulated) honest parties.
4. Execute the simulated routine $\texttt{SFeedValue}(id_\alpha)$ with $\mathcal{A}$ to obtain $\{[\![\widetilde{\alpha}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\}_{j\in\boldsymbol{Q}}$ and Δ_α.
5. Await the call to $\mathcal{F}_{\mathsf{Open}}$ with input $\left(\texttt{Initialize},\Gamma_{\mathcal{Q}},\left([\![1]\!]^{\mathcal{Q}}_{\mathcal{Q}_j},[\![\alpha']\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\right),sid_{\mathcal{Q}}\right)$ for $j\in\mathcal{Q}\cap\mathcal{A}$ from $\mathcal{A}$, redefine $[\![\widetilde{\alpha}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} \coloneqq [\![\alpha']\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$ for all $j\in\boldsymbol{Q}\cap A$, and then call $\overline{\mathcal{F}}_{\mathsf{Prep}}$ with input $\left(\texttt{Initialize},\Gamma_{\mathcal{Q}},\left([\![1]\!]^{\mathcal{Q}}_{\mathcal{Q}_j},[\![\widetilde{\alpha}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}\right),sid_{\mathcal{Q}}\right)$ for $j\in\boldsymbol{Q}\cap\overline{A}$.

Mask To generate a mask for $\mathcal{Q}_j$,

1. Compute new identifiers id_a, id_b and id_c and call $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple},(id_a,id_b,id_c),sid_{\mathcal{R}})$ to obtain some $[\![a]\!]^{\mathcal{R}}$ and relabel id_a as $id_{r^{(j)}}$ and the shares as $[\![r^{(j)}]\!]^{\mathcal{R}}$.
2. Execute $\texttt{SFeedValue}(id_{r^{(j)}})$ with $\mathcal{A}$ to obtain $[\![\widetilde{r^{(j)}}]\!]^{\mathcal{Q}}$, $\Delta_{r^{(j)}}$, $[\![\widetilde{\gamma(r^{(j)})}]\!]^{\mathcal{Q}}$, and $\Delta_{\gamma(r^{(j)})}$.
3. If $j\in\boldsymbol{Q}\setminus\overline{A}$,
   - Await shares $[\![\widetilde{r'^{(j)}}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\overline{\mathcal{A}}}$ from $\mathcal{A}$.
   - Set $\delta_{r^{(j)}} \coloneqq \sum_{i\in R\cap\overline{A}}[\![\widetilde{r'^{(j)}}]\!]^{\mathcal{R}}_{\mathcal{R}_i} - [\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}_i}$.
   - Retrieve from memory $[\![\widetilde{r^{(j)}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and $\Delta_{r^{(j)}}$ and send $[\![\widetilde{r^{(j)}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and $\Delta_{r^{(j)}}-\delta_{r^{(j)}}$ to $\overline{\mathcal{F}}_{\mathsf{Prep}}$.
   - Retrieve from memory $[\![\widetilde{\gamma(r^{(j)})}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and $\Delta_{\gamma(r^{(j)})}$ and send these to to $\overline{\mathcal{F}}_{\mathsf{Prep}}$.

   If $j\in\boldsymbol{Q}\cap\overline{A}$,
   - Send $[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\overline{\mathcal{A}}}$ to $\mathcal{A}$.
   - Send $r^{(j)}$ to $\overline{\mathcal{F}}_{\mathsf{Prep}}$.
   - Retrieve from memory $[\![r^{(j)}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and $\Delta_{r^{(j)}}$ and send these to $\overline{\mathcal{F}}_{\mathsf{Prep}}$.
   - Retrieve from memory $[\![\gamma(r^{(j)})]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and $\Delta_{\gamma(r^{(j)})}$ and send these to $\overline{\mathcal{F}}_{\mathsf{Prep}}$.

Triples

1. Parties in $\mathcal{R}$ compute new identifiers id_{a_k}, id_{b_k} and id_{c_k} for $k\in[T]$ and call $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ with input $(\texttt{Triple},(id_{a_k},id_{b_k},id_{c_k})^T_{k=1},sid_{\mathcal{R}})$ to obtain triples $([\![a_k]\!]^{\mathcal{R}},[\![b_k]\!]^{\mathcal{R}},[\![c_k]\!]^{\mathcal{R}})^T_{k=1}$.
2. For each $v\in\{a_k,b_k,c_k\}^T_{k=1}$, execute $\texttt{SFeedValue}(id_v)$ with $\mathcal{A}$ to obtain $[\![\tilde{v}]\!]^{\mathcal{Q}}$, Δ_v, $[\![\widetilde{\gamma(v)}]\!]^{\mathcal{Q}}$, and $\Delta_{\gamma(v)}$.
   Then call $\overline{\mathcal{F}}_{\mathsf{Prep}}$ with input $(\texttt{Triple},(id_{a_k},id_{b_k},id_{c_k})^T_{k=1},sid_{\mathcal{Q}})$, and when $\overline{\mathcal{F}}_{\mathsf{Prep}}$ executes $\texttt{Sample}(id_v)$ for each $v\in\{a_k,b_k,c_k\}^T_{k=1}$, send $[\![\tilde{v}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and Δ_v followed by $[\![\widetilde{\gamma(v)}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\overline{\mathcal{A}}}$ and $\Delta_{\gamma(v)}$.

Feed Value For the simulated macro $\texttt{SFeedValue}(id_v)$, where id_v is the identifier for a sharing $[\![v]\!]^{\mathcal{R}}$, the simulator does the following on $[\![v]\!]^{\mathcal{R}}$ and then on $[\![\gamma(v)]\!]^{\mathcal{R}}$:

Simulator $\overline{\mathcal{S}}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ (continued)

- For each $i \in R \setminus \overline{A}$,
  - Sample $\{[\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{j\in Q^{(i)}} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{j\in Q^{(i)}} [\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} = [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}$.
  - Send $\{[\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{j\in Q^{(i)}\cap\overline{A}}$ to $\mathcal{A}$.
  - Set $\Delta_i := 0$.
- For each $i \in R \cap \overline{A}$,
  - If $\mathcal{R}_i$ sends only to honest parties,
    - Await a set $\{[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{j\in Q^{(i)}}$ from $\mathcal{A}$.
    - Set $[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} := [\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$ for all $j \in Q^{(i)}$.
    - Set $\Delta_i := \left(\sum_{j\in Q^{(i)}} [\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\right) - [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}$.
  - If $\mathcal{R}_i$ sends to at least one corrupt party,
    - Await a (possibly empty) set $\{[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{j\in Q^{(i)}\setminus\overline{A}}$ from $\mathcal{A}$.
    - Sample reshares $\{[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\}_{j\in Q^{(i)}\cap\overline{A}}$ subject to

$$\sum_{j\in Q^{(i)}\cap\overline{A}} [\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + \sum_{j\in Q^{(i)}\setminus\overline{A}} [\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} = [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}.$$

    - Set $\Delta_i := 0$.
- For each $j \in Q \setminus \overline{A}$, compute $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} = \sum_{i\in R^{(j)}\cap\overline{A}} [\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + \sum_{i\in R^{(j)}\setminus\overline{A}} [\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$.
- For each $j \in Q \cap \overline{A}$, compute $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} = \sum_{i\in R^{(j)}\cap\overline{A}} [\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + \sum_{i\in R^{(j)}\setminus\overline{A}} [\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$.
- Compute the error $\Delta := \sum_{i\in R} \Delta_i$.
- (Locally) output $([\![w]\!]^{\mathcal{Q}}, \Delta)$.

Figure 5.7: Simulator $\overline{\mathcal{S}}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$ for $\overline{\mathcal{F}}_{\mathsf{Prep}}$.

Since the cover is secure, for every secret shared amongst $\mathcal{R}$ and reshared amongst $\mathcal{Q}$ there is a share held by an honest party in $\mathcal{R}$ for which at least one reshare is held by an honest party in $\mathcal{Q}$. This means that every set of shares for every secret during the execution of the protocol is statistically indistinguishable from uniform, which means that throughout the protocol execution the environment cannot learn the value of any secret. For example, consider that a secret a sampled by $\mathcal{S}$ during the execution of **Triples** in $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ is not the same as the value a' sampled by $\overline{\mathcal{F}}_{\mathsf{Prep}}$ during the corresponding execution in the ideal world (with high probability). The point is that this difference cannot be observed by the environment.

However, after the execution, the environment learns all shares of parties in $\mathcal{Q}$. Since $\mathcal{S}$ honestly executes $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$ locally, the MACs will be correct according to the MAC key sampled by $\mathcal{S}$, but the secrets and MACs held by parties in $\mathcal{Q}$ at the end of the execution should be correct with respect to the MAC key generated by $\overline{\mathcal{F}}_{\mathsf{Prep}}$. This is not a problem because in the definition of $\overline{\mathcal{F}}_{\mathsf{Prep}}$, whatever shares are sent to it by $\mathcal{S}$

Procedure	From	To	Message
Initialize	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$(\texttt{Initialize}, \Gamma^{\mathcal{R}}, [\![a]\!]^{\mathcal{R}}_{\mathcal{R}_i}, sid_{\mathcal{R}})_{i \in R \cap A}$
	$\mathcal{S}$	$\mathcal{A}$	$[\![[\![a]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{S}$	$[\![[\![\widetilde{a}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}^{\mathcal{Q}}$	$(\texttt{Initialize}, \Gamma^{\mathcal{Q}}, [\![\widetilde{a}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}, sid_{\mathcal{Q}})_{j \in Q \cap A}$
Mask	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$(\texttt{Triple}, (id_a, id_b, id_c), sid_{\mathcal{R}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$([\![a]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![b]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![c]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$([\![\gamma(a)]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![\gamma(b)]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![\gamma(c)]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	Abort/OK
	$\mathcal{S}$	$\mathcal{A}$	$[\![[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}$ (N.B. $r^{(j)} := a$)
	$\mathcal{S}$	$\mathcal{A}$	$[\![[\![\gamma(r^{(j)})]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{S}$	$[\![[\![\widetilde{r^{(j)}}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{S}$	$[\![[\![\widetilde{\gamma(r^{(j)})}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}$
	$\mathcal{S}$	$\mathcal{A}$	$[\![r^{(j)}]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}$
Triples	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid_{\mathcal{R}})$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$([\![a_k]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![b_k]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![c_k]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	$([\![\gamma(a_k)]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![\gamma(b_k)]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}, [\![\gamma(c_k)]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$	Abort/OK
	$\mathcal{S}$	$\mathcal{A}$	$([\![[\![a_k]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}, [\![[\![b_k]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}, [\![[\![c_k]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}})_{k=1}^{T}$
	$\mathcal{S}$	$\mathcal{A}$	$([\![[\![\gamma(a_k)]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}, [\![[\![\gamma(b_k)]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}}, [\![[\![\gamma(c_k)]\!]^{\mathcal{R}}_{\mathcal{R}\setminus\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\cap\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{S}$	$([\![[\![\widetilde{a_k}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}, [\![[\![\widetilde{b_k}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}, [\![[\![\widetilde{c_k}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}})_{k=1}^{T}$
	$\mathcal{A}$	$\mathcal{S}$	$([\![[\![\widetilde{\gamma(a_k)}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}, [\![[\![\widetilde{\gamma(b_k)}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}}, [\![[\![\widetilde{\gamma(c_k)}]\!]^{\mathcal{R}}_{\mathcal{R}\cap\mathcal{A}}]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\mathcal{A}})_{k=1}^{T}$

Figure 5.8: Transcript for $\overline{\Pi}^{\mathcal{R}\to\mathcal{Q}}_{\mathsf{Outsource}}$.

on behalf of corrupt and effectively-corrupt honest parties, $\overline{\mathcal{F}}_{\mathsf{Prep}}$ will create sharings of secrets with MACs corresponding to its *own* MAC key. The only job of the simulator is to ensure the same *errors* are introduced in the ideal world as in the real (hybrid) world.

Note that the adversary can cheat during initialization of the MAC key by calling $\mathcal{F}_{\mathsf{Prep}}^{\mathcal{R}}$ with one set of MAC key shares, then introducing errors in the execution of $\texttt{FeedValue}(a)$, and then sending different secrets to $\mathcal{F}_{\mathsf{Open}}$. In the simulation, the simulator just takes the shares provided as input to $\mathcal{F}_{\mathsf{Open}}$ and passes these on to $\overline{\mathcal{F}}_{\mathsf{Prep}}$ on behalf of corrupt parties, and then, since $\overline{\mathcal{F}}_{\mathsf{Prep}}$ expects shares from effectively-corrupt honest parties, the simulator also passes on the shares generated during the execution of $\texttt{SFeedValue}()$. There is therefore no difference between the MAC shares held by corrupt and effectively-corrupt parties in $\mathcal{Q}$ in the real world (with respect to $\mathcal{F}_{\mathsf{Open}}$) and the ideal world (with respect to $\overline{\mathcal{F}}_{\mathsf{Prep}}$).

Since there are no computational assumptions being exploited in the simulation,

the only strategy the environment can employ is to introduce errors and see how they affect the final outputs of all parties, since this means the simulator must correctly and consistently pass errors through into the ideal world. It only remains to argue that errors can be carried through from the real world to the ideal world, since this ensures that Π_{MACCheck} aborts with the correct distribution if it is executed later.

The only way to observe such errors is to inspect distributions on the *sum* of sets of shares in $\mathcal{Q}$, since any such set missing one share or more is indistinguishable from uniform. By the time the macro `FeedValue()` is called, $\mathcal{S}$ has a complete share vector $[\![v]\!]^{\mathcal{R}}$ produced by the local execution of $\mathcal{F}^{\mathcal{R}}_{\mathsf{Prep}}$. With each share, $\mathcal{S}$ behaves differently depending on the secure cover. The only time the simulator considers the adversary to have contributed an error is when all shares of a corrupt party are sent to honest parties. The intuition is that if a corrupt party in $\mathcal{Q}$, receiving from at least one corrupt party in $\mathcal{R}$, introduces an error and sends it to $\overline{\mathcal{F}}_{\mathsf{Prep}}$, this is equivalent to the corrupt sender in $\mathcal{R}$ sending a different reshare to the corrupt party in $\mathcal{Q}$. In other words, no error is "committed" to unless the adversary sends all reshares of a given share only to honest parties. The correctness of simulation regarding the errors the simulator computes will now be explained in detail.

Note that in the simulation, there are three "types" of reshare:

1. $\hat{v}$: honest to honest and honest to corrupt;
2. $\tilde{v}$: corrupt to honest and corrupt to corrupt;
3. $\bar{v}$: corrupt to corrupt, but *simulated*.

For each $j \in Q \cap \overline{A}$, $\mathcal{S}$ fixes $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} := \sum_{i \in R^{(j)} \setminus \overline{A}} [\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + \sum_{i \in R^{(j)} \cap \overline{A}} [\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$. Since the simulator does not receive the reshares of corrupt parties in R to corrupt parties in Q, this share $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$ is defined using the "guesses" $[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$'s not the "real" $[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}$'s that the adversary computes (or would compute).

Let $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q} \cap \overline{\mathcal{A}}}$ denote the shares subsequently sent from $\mathcal{S}$ to $\overline{\mathcal{F}}_{\mathsf{Prep}}$ and $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q} \setminus \overline{\mathcal{A}}}$ denote the shares sampled by $\overline{\mathcal{F}}_{\mathsf{Prep}}$, which are sent to (real) honest parties. Consider the variable defined as the sum of the shares of (real) honest parties in $\mathcal{Q}$, i.e. the sum of the shares $[\![w]\!]^{\mathcal{Q}}_{\mathcal{P}_j}$ for $j \in Q \setminus \overline{A}$, with the sum of the shares $[\![\tilde{w}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j}$ for $j \in \mathcal{Q} \cap \overline{\mathcal{A}}$ generated by the adversary that are not known to $\mathcal{S}$. Then observe that

$$\sum_{j \in Q \cap \overline{A}} [\![\tilde{w}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} + \sum_{j \in Q \setminus \overline{A}} [\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} = \sum_{j \in Q \cap \overline{A}} [\![\tilde{w}]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} + \left(w + \Delta - \sum_{j \in Q \cap \overline{A}} [\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}_j} \right)$$

$$
\begin{aligned}
&= \sum_{j\in Q\cap\overline{A}}\left(\cancel{\sum_{i\in R^{(j)}\setminus\overline{A}}[\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}} + \sum_{i\in R^{(j)}\cap\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\right) + w + \Delta \\
&\qquad - \sum_{j\in Q\cap\overline{A}}\left(\cancel{\sum_{i\in R^{(j)}\setminus\overline{A}}[\![[\![\hat{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}} + \sum_{i\in R^{(j)}\cap\overline{A}}[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\right) \\
&= \sum_{j\in Q\cap\overline{A}}\sum_{i\in R^{(j)}\cap\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} - \sum_{j\in Q\cap\overline{A}}\sum_{i\in R^{(j)}\cap\overline{A}}[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + w + \Delta \\
&= \sum_{i\in R\cap\overline{A}}\sum_{j\in Q^{(i)}\cap\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} - \sum_{i\in R\cap\overline{A}}\sum_{j\in Q^{(i)}\cap\overline{A}}[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + w + \Delta \\
&= \sum_{i\in R\cap\overline{A}}\sum_{j\in Q^{(i)}\cap\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} - \sum_{\substack{i\in R\cap\overline{A}:\\ Q^{(i)}\cap\overline{A}\neq\varnothing}}\sum_{j\in Q^{(i)}\cap\overline{A}}[\![[\![\bar{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + w + \Delta \\
&= \sum_{i\in R\cap\overline{A}}\sum_{j\in Q^{(i)}\cap\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} - \sum_{\substack{i\in R\cap\overline{A}:\\ Q^{(i)}\cap\overline{A}\neq\varnothing}}\left([\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i} - \sum_{j\in Q^{(i)}\setminus\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j}\right) + w \\
&\qquad + \sum_{\substack{i\in R\cap\overline{A}:\\ Q^{(i)}\cap\overline{A}=\varnothing}}\left(\sum_{j\in Q^{(i)}\setminus\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} - [\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i}\right) \\
&= \sum_{i\in R\cap\overline{A}}\sum_{j\in Q^{(i)}\cap\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} - \sum_{i\in R\cap\overline{A}}[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i} + \sum_{i\in R\cap\overline{A}}\sum_{j\in Q^{(i)}\setminus\overline{A}}[\![[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i}]\!]^{\mathcal{Q}^{(i)}}_{\mathcal{Q}_j} + w \\
&= \sum_{i\in R\cap\overline{A}}[\![\tilde{v}]\!]^{\mathcal{R}}_{\mathcal{R}_i} - \sum_{i\in R\cap\overline{A}}[\![v]\!]^{\mathcal{R}}_{\mathcal{R}_i} + w.
\end{aligned}
$$

This shows that the shares sampled by the functionality for honest parties during this procedure, namely the (vector of) shares $[\![w]\!]^{\mathcal{Q}}_{\mathcal{Q}\setminus\overline{\mathcal{A}}}$, sum with the shares the adversary would compute to differ from the secret by precisely the error the adversary introduced on the shares when resharing.

When masks are generated for parties in $\mathcal{Q}$, there are two opportunities for parties in $\mathcal{R}$ to produce errors: first, in the resharing during `FeedValue()`, and second, when the parties in $\mathcal{R}$ open the mask to the party in $\mathcal{Q}$. However, since at least one party in $\mathcal{R}$ is honest, the environment can only compute the difference between $\sum_{j\in Q}[\![r]\!]^{\mathcal{Q}}_{\mathcal{P}_j}$ and $\sum_{i\in R}[\![r]\!]^{\mathcal{R}}_{\mathcal{P}_i}$ to observe any errors. This means that in the simulation it is sufficient to "imply" an error in opening by *subtracting* the error introduced in opening from the error introduced in resharing. In other words, the environment cannot distinguish between

$$\left(\varepsilon + \sum_{i\in R}[\![r]\!]^{\mathcal{R}}_{\mathcal{P}_i},\quad \delta + r\right) \qquad \text{and} \qquad \left(\varepsilon - \delta + \sum_{i\in R}[\![r]\!]^{\mathcal{R}}_{\mathcal{P}_i},\quad r\right).$$

□

5.5 Probabilistically Choosing a Secure Cover

In order to cut the costs of resharing, the parties can attempt to agree on a secure cover by a probabilistic process. In this section, it will be assumed that $\mathcal{R}$ and $\mathcal{Q}$ are under threshold access structures but that they will execute full-threshold protocols, for simplicity.

Let $t^{\mathcal{R}}$ and $t^{\mathcal{Q}}$ be the number of corrupt parties in $\mathcal{R}$ and $\mathcal{Q}$ respectively. Let $\varepsilon^{\mathcal{R}} := t^{\mathcal{R}}/n^{\mathcal{R}}$ and $\varepsilon^{\mathcal{Q}} := t^{\mathcal{Q}}/n^{\mathcal{Q}}$ be the associated ratios. To help with the analysis and for efficiency and load-balancing reasons, it will be assumed that each party in $\mathcal{R}$ sends to the same number of parties $\ell \geq \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ in $\mathcal{Q}$. Any assignment of sets to parties in $\mathcal{R}$ which covers $\mathcal{Q}$ where $\ell = t^{\mathcal{Q}} + 1$ is secure since every party in $\mathcal{R}$ necessarily sends to at least one honest party in $\mathcal{Q}$.

Figure 5.9 shows an example of a load-balanced topology when $n^{\mathcal{Q}} \approx 2n^{\mathcal{R}}$. Note that it is not necessarily the case that each party in $\mathcal{Q}$ receives the same number of shares, even though each party in $\mathcal{R}$ is required to reshare to the same number of parties in $\mathcal{Q}$.

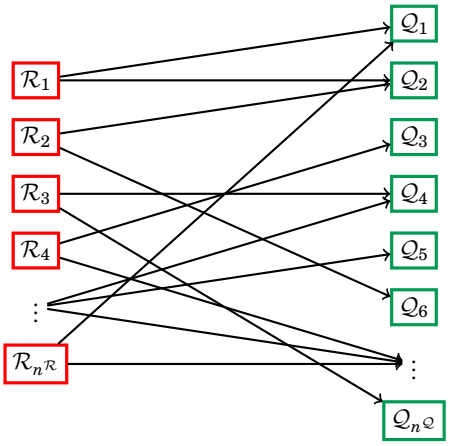


Figure 5.9: Load-Balanced Topology.

An algorithm for creating a cover probabilistically is given in Figure 5.10. The high-level idea is the following:

1. Randomly assign parties in $\mathcal{Q}$ to each party in $\mathcal{R}$ so that each party in $\mathcal{R}$ has exactly $\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ parties in $\mathcal{Q}$ assigned to it. For ease of exposition, it is assumed that $n^{\mathcal{R}} | n^{\mathcal{Q}}$.
2. For each party in $\mathcal{R}$, assign random parties in $\mathcal{Q}$ until each party in $\mathcal{R}$ is assigned a total of ℓ parties.

Note that in Step 12, the assignment can be made by cycling through only the entries $M_{i,j}$ of row i for which $M_{i,j} = 0$ to reduce the expected number of loops, but, since the

random oracle is evaluated locally, this would not significantly improve runtime unless there is a large number of parties.

Algorithm for Computing a Secure Cover

Recall that $M_{i,j}$ denotes the $(i,j)^{\text{th}}$ entry of M and M_i denotes the i^{th} row. The macro `Shuffle()` denotes the Knuth Shuffle in Figure 2.1, Section 2.1.4.

Input R, Q, and ℓ. For the purposes of this algorithm, $R := [n^{\mathcal{R}}]$ and $Q := [n^{\mathcal{Q}}]$ and are distinguished by using i to index R and j to index Q.

Output A cover, $\{\mathcal{Q}^{(i)}\}_{i \in R}$, where each set has cardinality ℓ.

Method

1. Call an instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \{0,1\}^{\kappa} sid)$.
2. Call an instance of $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, Q, sid)$.
3. Let $M \in \mathbb{F}^{n^{\mathcal{R}} \times n^{\mathcal{Q}}}$.
4. For each $i \in R$,
   a) For each $j \in Q$,
      i. Set $M_{i,j} := 0$.
   b) End For.
5. End For.
6. Set $count := 1$.
7. Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid)$ to obtain $seed_Q$.
8. Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid)$ to obtain $seed_\pi$.
9. Set $\pi := \texttt{Shuffle}(seed_\pi, n^{\mathcal{Q}})$.
10. For each $i \in R$,
    a) For each j where $\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil \cdot (i-1) < j \leq \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil \cdot i$,
       i. Set $M_{i,\pi(j)} := 1$.
    b) End For.
11. End For.
12. For each $i \in R$,
    a) While $\mathsf{HW}(M_i{}^{\top}) < \ell$,
       i. Do
          A. Call $\mathcal{F}_{\mathsf{RO}}$ with input $(seed_Q \| count, sid)$ to obtain j.
          B. Set $count := count + 1$.
       ii. Until $M_{i,j} = 0$
       iii. Set $M_{i,j} := 1$.
    b) End While.
13. End For.
14. For each $i \in R$,
    a) For each $j \in Q$,
       i. If $M_{i,j} = 1$ then

Algorithm for Computing a Secure Cover (continued)

A. Set $\mathcal{Q}^{(i)} := \mathcal{Q}^{(i)} \cup \{\mathcal{Q}_j\}$.
ii. End If.
b) End For.
15. End For.
16. Output $\{\mathcal{Q}^{(i)}\}_{i \in R}$.

Figure 5.10: Algorithm for Computing a Secure Cover.

Correctness

The algorithm allows different parties in $\mathcal{Q}$ to receive from different numbers of parties in $\mathcal{R}$, whilst parties in $\mathcal{R}$ always send to the same number of parties in $\mathcal{Q}$. One of the reasons for doing this is that it lends itself better to analysis of probabilities below. Let X be the event that every good party in $\mathcal{R}$ is assigned only dishonest parties in Step 10, and let Y be the event that every good party in $\mathcal{R}$ is assigned only dishonest parties in Step 12. Since these events are independent, the probability that the algorithm produces a secure cover is given by

$$1 - \Pr[X \wedge Y] = 1 - \Pr[X] \cdot \Pr[Y].$$

Computing $\Pr[X]$ The first probability is the number of ways of choosing $\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ parties from the $t^{\mathcal{Q}}$ corrupt parties divided by the number of ways of choosing $\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ parties from all $n^{\mathcal{Q}}$ parties:

$$\frac{\binom{t^{\mathcal{Q}}}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}}}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}$$

Thus the first $\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ corrupt parties in $\mathcal{Q}$ have been assigned. Then the probability that the next honest party in $\mathcal{R}$ is also assigned only corrupt parties from the remaining $t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ corrupt parties, out of the $n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ remaining parties in $\mathcal{Q}$, is

$$\frac{\binom{t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}$$

and this continues until all the $n^{\mathcal{R}} - t^{\mathcal{R}} - 1$ honest parties in $\mathcal{R}$ have been assigned parties in $\mathcal{Q}$.

Computing $\Pr[Y]$ Each party in $\mathcal{R}$ has been assigned $\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ parties in $\mathcal{Q}$ so that each party in $\mathcal{Q}$ has been assigned to exactly one party in $\mathcal{R}$. In Step 12, each party in $\mathcal{R}$ is independently assigned a random set of $\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ more parties in $\mathcal{Q}$. For a given party in $\mathcal{R}$, this is the number of ways of choosing $\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ dishonest parties from the remaining $n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ parties in $\mathcal{Q}$ such that they too are all dishonest – i.e. they are from the $t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil$ remaining dishonest parties. Thus

$$\Pr[Y] = \left(\frac{\binom{t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}} \right)^{n^{\mathcal{R}} - t^{\mathcal{R}}}.$$

Putting these together, the probability that the algorithm produces a secure cover is given by:

$$1 - \left(\frac{\binom{t^{\mathcal{Q}}}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil} \cdot \binom{t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil} \cdot \cdots \cdot \binom{t^{\mathcal{Q}} - (n^{\mathcal{R}} - t^{\mathcal{R}} - 1)\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}}}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil} \cdot \binom{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil} \cdot \cdots \cdot \binom{n^{\mathcal{Q}} - (n^{\mathcal{R}} - t^{\mathcal{R}} - 1)\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}} \right) \cdot \left(\frac{\binom{t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}} \right)^{n^{\mathcal{R}} - t^{\mathcal{R}}}.$$

Simplified, this gives

$$1 - \frac{t^{\mathcal{Q}}! \cdot (n^{\mathcal{Q}} - (n^{\mathcal{R}} - t^{\mathcal{R}})\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil)!}{n^{\mathcal{Q}}! \cdot (t^{\mathcal{Q}} - (n^{\mathcal{R}} - t^{\mathcal{R}})\lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil)!} \cdot \left(\frac{\binom{t^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}} \right)^{n^{\mathcal{R}} - t^{\mathcal{R}}}. \tag{5.1}$$

To see what happens in the extremal case where all parties but one are corrupt in each of $\mathcal{R}$ and $\mathcal{Q}$, set $t^{\mathcal{Q}} = n^{\mathcal{Q}} - 1$ and $t^{\mathcal{R}} = n^{\mathcal{R}} - 1$. Then:

$$\begin{aligned} 1 - \frac{(n^{\mathcal{Q}} - 1)! \cdot (n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil)!}{(n^{\mathcal{Q}})! \cdot (n^{\mathcal{Q}} - 1 - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil)!} \cdot \frac{\binom{(n^{\mathcal{Q}} - 1) - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}}{\binom{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{\ell - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}} \\ = 1 - \frac{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil}{n^{\mathcal{Q}}} \cdot \frac{n^{\mathcal{Q}} - \ell}{n^{\mathcal{Q}} - \lceil n^{\mathcal{Q}}/n^{\mathcal{R}} \rceil} = \frac{\ell}{n^{\mathcal{Q}}}. \end{aligned}$$

This agrees with the intuition that when ℓ is equal to $n^{\mathcal{Q}}$, i.e. each party in $\mathcal{R}$ sends to every party in $\mathcal{Q}$, the cover is secure with probability 1. Since this probability is linear in ℓ, for outsourcing full-threshold to full-threshold, the only way of achieving statistical security σ is to have an intractible number of parties, and hence in this situation it is recommended for all parties in $\mathcal{R}$ to reshare to all parties in $\mathcal{Q}$.

However, with lower thresholds the probability of a secure cover from the probabilistic algorithm increases. When ℓ is at least $t^{\mathcal{Q}} + 1$, then every party in $\mathcal{R}$ necessarily sends to at least one honest party. The probability that this algorithm produces a secure

cover grows with the number of parties. For example, if $\mathcal{R}$ is a $(5,2)$-threshold and $\mathcal{Q}$ a $(50,25)$-threshold, then setting $\ell = 23$ gives a secure cover with probability at least $1-2^{-80}$, instead of the 25 parties required to guarantee the cover is secure.

Thus it is not immediately clear that this probabilistic approach offers significant advantage, as it only works when the number of parties is comparatively large. However, if a set of parties $\mathcal{R}$ has an access structure $\Gamma^{\mathcal{R}}$ with $n^{\mathcal{R}}/2 < t^{\mathcal{R}} < n^{\mathcal{R}}$ and runs SPDZ then, for example, if $\mathcal{R}$ is $(20,15)$-threshold but runs a full-threshold protocol, and $\mathcal{Q}$ is $(50,25)$-threshold, then $\ell = 13$ gives a secure cover, and thus the amount of communication is halved.

5.6 Communication Complexity

The costs associated with verifying the correctness of resharing are negligible compared to the amount of preprocessed data that is transferred in circuits of large enough size. The salient factor, then, is the cost of resharing.

The two sets of parties are connected with bilateral secure channels, giving the complete bipartite graph between them. This topology requires $n^{\mathcal{R}} \cdot n^{\mathcal{Q}}$ secure connections, besides the additional internal networks in $\mathcal{R}$ and $\mathcal{Q}$, and is shown in Figure 5.11.

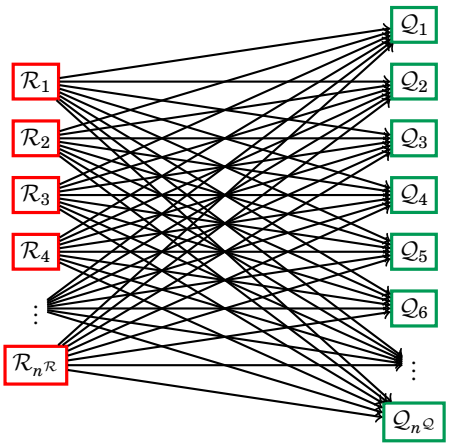


Figure 5.11: Complete Bipartite Graph.

For the $\mathcal{Q}_2$ outsourcing protocol, the communication cost is that of each party in $\mathcal{R}$ resharing their shares under the LSSS for $\mathcal{Q}$. The finite-field elements are sent over the $n^{\mathcal{R}} \cdot n^{\mathcal{Q}}$ secure channels, and the total communication cost (besides the verification step) is $m^{\mathcal{R}} \cdot m^{\mathcal{Q}}$. If the access structures are threshold access structures, then Shamir's secret-

sharing scheme can be used and then this is $n^{\mathcal{R}} \cdot n^{\mathcal{Q}}$, which is a linear communication cost per party in $\mathcal{R}$.

For the full-threshold case, without the optimization of probabilistically defining a secure cover, the complete bipartite graph of secure channels is required, and again the communication cost is $n^{\mathcal{R}} \cdot n^{\mathcal{Q}}$.

While the use case of outsourced preprocessing is primarily for situations in which clients cannot perform preprocessing themselves, it would be informative to compare the runtime of protocols executed in the "standard" manner with protocols in which outsourcing is performed, to see if there are performance gains even when computing parties are able to execute preprocessing. Unfortunately, this is difficult to do without concrete instantiations due to the many variable factors in hardware and protocol choice. This would be an interesting avenue of further research.

As a final note to conclude this chapter, observe that protocols for generating preprocessing for parties under a $\mathscr{Q}_2$ access structure often assume a *multiplicative* LSSS – as indeed is required for the protocol that will be presented in Chapter 6. (See Section 2.4.4 for the definition of multiplicative LSSS.) However, Π_{Online}, which is realized in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, makes use *only* of the linear property of the LSSS, since the costly multiplication step is resolved in the preprocessing phase. Consequently, the LSSS used for parties in $\mathcal{Q}$ need not be multiplicative, which means a broader class of LSSS can be used when resharing via the outsourcing protocol presented. This is useful because it potentially allows a more efficient LSSSs to be used (i.e. one with fewer total shares) to realize the access structure on $\mathcal{Q}$. Further discussion on this is given in Chapter 7.

Chapter 6

$\mathscr{Q}_2$ MPC for Small Numbers of Parties

This chapter is based on work published at SCN'18 under the title Reducing Communication Channels in MPC *[KRSW18] and was joint work with Marcel Keller, Dragoș Rotaru and Nigel Smart.*

The functionality $\mathcal{F}_{\mathsf{Rand}}$ from that paper has been separated into two separate functionalities $\mathcal{F}_{\mathsf{RSS}}^{\mathsf{R}}$ and $\mathcal{F}_{\mathsf{RZS}}$. They have been proved secure in the universal composability (UC) framework.

What was previously an optimization to the main protocol in the publication now replaces the "un"-optimized variant, and the functionality now defines a more "full" preprocessing phase from what was described in the published version, so that now input and output masks are generated in addition to Beaver triples. The "opening" protocol of the original work has been replaced with the generic functionality $\mathcal{F}_{\mathsf{Open}}$ from Chapter 4, which is taken from [SW19], resulting in a tidier proof of the actively-secure protocol.

This chapter This chapter focuses on two aspects of the communication cost associated with multi-party computation (MPC): the total amount of data sent over the network, and the number of channels the parties must maintain throughout circuit evaluation.

More specifically, the focus is on an actively-secure, efficient protocol for $\mathscr{Q}_2$ access structures with computational security, making use of the error-detection properties of the linear secret-sharing scheme (LSSS) discussed in Chapter 3. The resulting protocol offers significant advantages in terms of communication cost when compared to the historical, mainly information-theoretic (IT) protocols in this setting. The protocol uses replicated secret-sharing so the focus of this chapter is primarily on a protocol for a small number of parties as replicated secret-sharing requires $\Omega(2^n/\sqrt{n})$ (i.e.

exponentially-many) shares for threshold schemes.

6.1 Overview

General, as opposed to threshold, access structures are practically interesting in situations where different groups of parties play different organizational roles. For example, in a financial application, one may have a computation performed amongst a number of banks and regulators; the required access structures for collaboration between the banks and the regulators may not be a straightforward threshold.

In the classical results of Chaum et al. [CCD88a] and Ben-Or et al. [BGW88], parties were assumed to be connected by a complete network of secure channels. These results for honest-majority threshold access structures were extended to arbitrary access structures by Hirt and Maurer [HM97] and Beaver and Wool [BW98], in which case the two necessary and sufficient conditions become $\mathscr{Q}_2$ and $\mathscr{Q}_3$ respectively, as discussed in Section 2.6.

Another line of work considered computationally-bounded adversaries, starting with [GMW87]. There, parties are connected by a complete network of authenticated channels; it was shown that actively-secure protocols are possible in the (n,t)-threshold setting when $t < n/2$ (i.e. with an honest majority), and active security *with abort* when only one party is honest. Generally speaking, such computationally-secure protocols are less efficient than the information-theoretic protocols as they require some form of public-key cryptography (PKC); however, computational assumptions enable computation in situations when it is provably impossible to do so using only IT primitives.

In recent years there has been considerable progress in practical MPC by mixing the computational and IT approaches. For example, the VIFF [DGKN09], BDOZ [BDOZ11], SPDZ [DPSZ12], Tiny-OT [NNOB12], and ABY [DSZ15] protocols are in the preprocessing model and defer PKC to the preprocessing phase, and then use information-theoretic primitives in the online phase.

It is therefore logical to wonder what advancements might be made in the $\mathscr{Q}_2$ setting by sacrificing IT security. Recently, Araki et al. [AFL+16] gave an efficient passively-secure MPC evaluation of the advanced encryption standard (AES) circuit (a common benchmark) for a $(3,1)$-threshold access structure. This was then generalized to an actively-secure protocol by Furukawa et al. [FLNW17]. Both protocols require a preprocessing phase making use of symmetric-key cryptographic primitives only; thus the

preprocessing is much more efficient than for the full-threshold protocols mentioned above.

The passively-secure protocol of [AFL+16] is very cheap for a number of reasons. Firstly, the preprocessing phase is only used to produce additive sharings of zero. Pseudorandom zero-sharings (PRZSs) can be easily produced non-interactively after a one-time setup phase using symmetric key primitives, as will be shown in Section 6.3. Secondly, the network is not assumed to be complete: each party only sends data to one other party via a secure channel, and only receives data from the third party via a secure channel. Thirdly, parties only need to transmit one finite-field element per multiplication. Compared to IT protocols in the same setting, the protocol requires that each party holds two finite-field elements per share, as opposed to using an ideal secret-sharing scheme, such as Shamir's, in which each party need only hold one finite-field element per secret.

The underlying protocol of Araki et al., bar the use of the PRZSs, is highly reminiscent of the Sharemind system [BLW08], which also assumes a $(3,1)$-threshold access structure. Since both [AFL+16] and [BLW08] are based on replicated secret-sharing, they are also closely related to the "MPC-Made-Simple" approach of Maurer [Mau06]. Thus, for the case of this specific access structure, the work of [AFL+16] can be seen as using cryptographic assumptions to optimize the information-theoretic approach of [Mau06].

The actively-secure successor to [AFL+16] by Furakawa et al. [FLNW17] uses the passively-secure protocol (over an incomplete network of secure channels) to run a preprocessing phase that produces Beaver triples. These are then consumed in the online phase, by using the triples to *verify* the passively-secure multiplication of actual secrets. This is different from the traditional methodology in which Beaver triples are used to *perform* the multiplication as describe in Section 2.5.3, which allows the online phase to be executed over authenticated channels rather than secure channels.

The question addressed in this chapter is whether the approach outlined in [AFL+16], [BLW08] and [FLNW17] is particularly tied to the $(3,1)$-threshold access structure, or whether it generalizes to other access structures. The protocol presented in this chapter shows that they do indeed generalize to any $\mathscr{Q}_2$ access structure.

6.2 Preliminaries

This chapter concerns the evaluation of an arithmetic circuit over a finite field $\mathbb{F}_q$, where q is a prime power. Most of the notation used in this chapter was defined in Chapter 2. The network requirements are best understood in the context of the protocols themselves and so the descriptions are deferred until later.

6.2.1 Replicated Secret-Sharing

The protocols in this chapter make use of replicated secret-sharing, which was defined in Section 2.4.3 but is explained in the context of MPC here. Recall that the set Δ^+ of maximally unqualified sets of parties, and its structure, is important for replicated secret-sharing; it is notationally simpler to consider the set of complements of maximally unqualified sets, which is denoted by $\nabla := \{\mathcal{G} \in 2^{\mathcal{P}} : \mathcal{P} \setminus \mathcal{G} \in \Delta^+\}$. Note that in general it is not true that the set ∇ is equal to the set of minimally qualified sets, though there are cases for which they do coincide (for example, $\left(n, \frac{n-1}{2}\right)$-threshold access structures where n is odd). In replicated secret-sharing, a secret x is shared as an additive sum $x = \sum_{\mathcal{G} \in \nabla} x^{\mathtt{R}}_{\mathcal{G}}$, where party $\mathcal{P}_i$ is handed $x^{\mathtt{R}}_{\mathcal{G}}$ if and only if $i \in \mathcal{G}$.

It is clear that if Δ is $\mathscr{Q}_2$, then so is any subset. In particular, the set of maximally unqualified sets Δ^+ is also $\mathscr{Q}_2$. In fact, if Δ^+ is $\mathscr{Q}_2$ then Δ is $\mathscr{Q}_2$. Hence, for the set of complements ∇ it holds that if $\mathcal{G}_1, \mathcal{G}_2 \in \nabla$ then $\mathcal{G}_1 \cap \mathcal{G}_2 \neq \varnothing$. A set ∇ for which this property holds was called a *quorum system* by Beaver and Wool [BW98].

Recall that a sharing of a secret x is denoted by the vector $[\![x]\!]$, and $[\![x]\!]_{\mathcal{P}_i}$ is the vector of shares that party $\mathcal{P}_i$ holds. For replicated secret-sharing, additionally $[\![x]\!]^{\mathtt{R}}$ is used to denote the share vector proper, whereas $x^{\mathtt{R}}_{\mathcal{G}}$ is used to denote the summand of the secret corresponding to the set $\mathcal{G} \in \nabla$. Recall from Section 2.4.4 that if an access structure is $\mathscr{Q}_2$ then replicated secret-sharing is multiplicative, i.e. given two secret-shared values $[\![a]\!]$ and $[\![b]\!]$, an additive sharing of the product $a \cdot b$ can be computed as a linear combination of the shares the parties hold; i.e. for each $i \in [n]$ there exists a vector $\boldsymbol{\mu}_i \in \mathbb{F}^{|\rho^{-1}(i)|^2}$ such that

$$a \cdot b = \sum_{i=1}^{n} \langle \boldsymbol{\mu}_i, [\![a]\!]_{\mathcal{P}_i} \otimes [\![b]\!]_{\mathcal{P}_i} \rangle.$$

The fact that by local computations the parties each obtain one summand of the product is the reason that it is possible to build an MPC protocol for any $\mathscr{Q}_2$ access structure secure against passive adversaries and with IT security (which is exactly the protocol of Hirt and Maurer [HM97]).

Example 6.1, below, will be used throughout this chapter to demonstrate the savings which can result from the protocol presented later, and also to examine the communication channels required.

Example 6.1. Consider the following set of maximally unqualified sets for a six-party access structure

$$\Delta^+ = \Big\{\{2,5,6\},\{3,5,6\},\{4,5,6\},\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{1,6\},\\ \{2,3\},\{2,4\},\{3,4\}\Big\}.$$

Here the set ∇ is

$$\nabla = \Big\{\{1,3,4\},\{1,2,4\},\{1,2,3\},\{3,4,5,6\},\{2,4,5,6\},\{2,3,5,6\},\\ \{2,3,4,6\},\{2,3,4,5\},\{1,4,5,6\},\{1,3,5,6\},\{1,2,5,6\}\Big\}.$$

6.2.2 Redundancy

If there is a party $\mathcal{P}_i$ that is in a qualified set only if some other party $\mathcal{P}_j$ is in the qualified set then the access structure is said to contain *redundancy* and $\mathcal{P}_i$ is said to be *redundant*.

In the terminology of access structures, given an access structure Γ, a party $\mathcal{P}_i$ is considered *redundant* if there exists another party $\mathcal{P}_j$ ($j \neq i$) such that for any $\mathcal{U} \in \Delta$, $\mathcal{P}_i \notin \mathcal{U}$ implies $\mathcal{P}_j \notin \mathcal{U}$. In terms of replicated secret sharing, a party $\mathcal{P}_i$ is redundant if there is another party $\mathcal{P}_j$ that holds every share $\mathcal{P}_i$ holds (and possibly more).

For example, in the access structure defined by

$$\Delta^+ := \{\{1,2\},\{1,3,4\},\{2,3,4\}\}$$

$\mathcal{P}_3$ (or, equivalently, $\mathcal{P}_4$) is redundant; once removed, the access structure becomes

$$\Delta^+ := \{\{1,2\},\{1,3\},\{2,3\}\}.$$

Note that if any party is omitted from all sets in Δ^+ then it is present in all sets in ∇ and hence every party, but this party, is redundant, which makes the MPC protocol trivial: the omitted party can simply perform the entire computation itself and output the result to all parties.

The idea is that $\mathcal{P}_i$ learns information only when $\mathcal{P}_j$ learns it, so if an adversary corrupts $\mathcal{P}_j$ it learns as much as when it corrupts $\mathcal{P}_j$ *and* $\mathcal{P}_i$. For MPC, this means that

preprocessing can be outsourced to a non-redundant subset of parties and the data re-distributed according to the original set of parties. For the protocol given in this chapter this outsourcing is particularly cheap as replicated secret-sharing is used so the resharing cost is simply that of forwarding shares to the redundant party.

6.3 Computational Random Sharings

Improving on the protocols in [BW98] and [Mau06] seems to require sacrificing the IT security for computational assumptions. In this section protocols for random sharings are given, which are key components of the MPC protocol. The focus is on computational security in the random oracle model, which leads to a very efficient protocol: any number of (pseudo)random secrets can be generated from a one-time setup phase. Discussion of the generation of random secrets with IT security is given in Section 7.3. As the one-time setup cost potentially[1] grows exponentially with the number of parties, the IT method is more efficient for a large number of parties as the asymptotic cost is linear.

Specifically, this section gives protocols that UC-realize the functionalities $\mathcal{F}_{\mathsf{RSS}}^{\mathsf{R}}$ in Figure 6.1 and $\mathcal{F}_{\mathsf{RZS}}$ in Figure 6.2, which provide parties with pseudorandom secret-sharings (PRSSs) and pseudorandom zero-sharings (PRZSs), respectively. The idea is for different sets of parties to agree on keys that are appended to every call to a random oracle to obtain correlated randomness that can be used to generate random sharings[2].

UC proofs for random sharings were given in [DGKN09], but the functionalities were not defined. Since they are a key component of the MPC protocols, full proofs are given here.

[1] The cost depends on the access structure.

[2] The random oracle here can be instantiated with a keyed pseudorandom function (PRF) rather than a hash function.

Functionality $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$

Initialize On input $(\texttt{Initialize}, \Gamma, sid)$ from all honest parties and $\mathcal{S}$, where sid is a new session identifier, $\mathcal{P}$ is the set of parties, and Γ is an access structure, set ∇ to be the set of complements of sets in Δ^+ and await further messages.

Sharing of Secret On input $(\texttt{SecretSharing}, id, sid)$ from all honest parties and $\mathcal{S}$, the functionality does the following:

1. If id is a new identifier, sample a set $\{r^{\mathbb{R}}_{\mathcal{G}} : G \in \nabla\} \leftarrow \mathcal{U}(\mathbb{F})$.
2. For each $\mathcal{G} \in \nabla$, for each $\mathcal{P}_i \in \mathcal{G}$, send $r^{\mathbb{R}}_{\mathcal{G}}$ to $\mathcal{P}_i$ or to $\mathcal{S}$ if $\mathcal{P}_i \in \mathcal{A}$.

Figure 6.1: Functionality for Secret-Sharings of Random Secrets for Replicated Secret-Sharing, $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$.

Notice that the adversary cannot choose its own shares. This reflects the fact that with replicated shares generated by PRF keys, $\mathcal{A}$ does not have a choice in what values the shares take. Of course, this does not preclude it from ignoring these inputs later on, but the validity of doing so is dealt with outside the execution of $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ (specifically, by $\mathcal{F}_{\mathsf{Open}}$ in the protocol later on).

Functionality $\mathcal{F}_{\mathsf{RZS}}$

Initialize On input $(\texttt{Initialize}, \mathcal{P}, sid)$ from all parties, where sid is a new session identifier and $\mathcal{P}$ is the set of parties, await further messages.

Sharing of Zero On input $(\texttt{ZeroSharing}, id, \mathcal{P}', sid)$ from all parties in a set $\mathcal{P}' \subseteq \mathcal{P}$, where id is a new identifier,

1. Sample $\{[\![t]\!]^{\mathbb{A}}_{\mathcal{P}_i}\}_{i\in[n']} \leftarrow \mathcal{U}(\mathbb{F})$ subject to the constraint that $\sum_{i\in[n']} [\![t]\!]^{\mathbb{A}}_{\mathcal{P}_i} = 0$.
2. For each $i \in [n']$, send $[\![t]\!]^{\mathbb{A}}_{\mathcal{P}_i}$ to $\mathcal{P}_i$, or to $\mathcal{S}$ if $i \in A$.

Figure 6.2: Functionality for Secret-Sharings of Zero, $\mathcal{F}_{\mathsf{RZS}}$.

6.3.1 PRSSs

The protocol $\Pi^{\mathbb{R}}_{\mathsf{RSS}}$ used to realize $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ is given in Figure 6.3.

Protocol $\Pi^{\mathbb{R}}_{\mathsf{RSS}}$

This protocol is realized in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.

Initialize

1. The parties agree on a session identifier sid and a computational security parameter, κ.
2. The parties call an instance of $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, \mathbb{F}, sid)$.
3. Each set of parties $\mathcal{G} \in \nabla$ agree on a session identifier $sid_{\mathcal{G}}$ and send the message $(\texttt{Initialize}, \{0,1\}^{2\cdot\kappa}, sid_{\mathcal{G}})$ to an instance of $\mathcal{F}_{\mathsf{CoinFlip}}$; let this instance be denoted by $\mathcal{F}^{\mathcal{G}}_{\mathsf{CoinFlip}}$.
4. Each set of parties $\mathcal{G} \in \nabla$ call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid_{\mathcal{G}})$ to obtain $k_{\mathcal{G}}$.

Sharing of Secret

1. The parties compute a new identifier, id_r.
2. For each $\mathcal{G} \in \nabla$, the parties in $\mathcal{G}$ call $\mathcal{F}_{\mathsf{RO}}$ with input $(id_r \| k_{\mathcal{G}}, sid)$ and set the value they receive as output as $r^{\mathbb{R}}_{\mathcal{G}}$.
3. Each party $\mathcal{P}_i$ concatenates the shares $\{r^{\mathbb{R}}_{\mathcal{G}} : \mathcal{G} \ni \mathcal{P}_i\}$ into a share vector $[\![r]\!]^{\mathbb{R}}_{\mathcal{P}_i}$.

Figure 6.3: Protocol for Secret-Sharings of Random Secrets using Replicated Secret-Sharing, $\Pi^{\mathbb{R}}_{\mathsf{RSS}}$.

Theorem 6.1. *The protocol $\Pi^{\mathbb{R}}_{\mathsf{RSS}}$ UC-securely realizes the functionality $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ against a static, active, computationally-bounded adversary in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.*

Proof. The simulator is given in Figure 6.4.

Simulator $\mathcal{S}^{\mathbb{R}}_{\mathsf{RSS}}$

Initialize

1. Agree on a session identifier, sid, with $\mathcal{A}$.
2. Await the call to $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, \mathbb{F}, sid)$ from $\mathcal{A}$ and send the command $(\texttt{Initialize}, \Gamma, sid)$ to $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$.
3. For each $\mathcal{G} \in \nabla$ where $\mathcal{G} \cap \mathcal{A} \neq \varnothing$, agree on a session identifier $sid_{\mathcal{G}}$ with $\mathcal{A}$ and await the call to $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \{0,1\}^{2\cdot\kappa}, sid_{\mathcal{G}})$ and execute it honestly with $\mathcal{A}$, storing the outputs locally.
4. For each $\mathcal{G} \in \nabla$ where $\mathcal{G} \cap \mathcal{A} \neq \varnothing$, await the call to $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid_{\mathcal{G}})$ and execute it honestly, sending the output $k_{\mathcal{G}}$ to $\mathcal{A}$.

Simulator $\mathcal{S}^{\mathbb{R}}_{\mathsf{RSS}}$ (continued)

Sharing of Secret

1. Compute a new identifier, id_r.
2. Call $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_r, sid)$ and then do the following:
   - Await the set of shares $\{r^{\mathbb{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \wedge \mathcal{G} \cap \mathcal{A} \neq \emptyset\}$ from $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$.
   - For each $\mathcal{G} \in \nabla$ where $\mathcal{G} \cap \mathcal{A} \neq \emptyset$, await the call to $\mathcal{F}_{\mathsf{RO}}$ with input (s, sid) from $\mathcal{A}$.
   - If $s = id_r \| k_{\mathcal{G}}$ then send $r^{\mathbb{R}}_{\mathcal{G}}$ to $\mathcal{A}$.
   - If s has been queried before, return the same output that was sent before.
   - Otherwise sample $t \leftarrow \mathcal{U}(\mathbb{F})$ and send t to $\mathcal{A}$.
3. (No simulation is required for this step.)

Figure 6.4: Simulator $\mathcal{S}^{\mathbb{R}}_{\mathsf{RSS}}$ for $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$.

The only problem occurs if $\mathcal{A}$ queries the random oracle *before* the set of seeds $\{k_{\mathcal{G}}\}_{\mathcal{G} \in \nabla : \mathcal{G} \cap \mathcal{A} \neq \emptyset\}}$ are agreed on, and thus before $\mathcal{S}$ can program the random oracle by replacing the call with an output from $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$.

The adversary is computationally bounded, so the number of possible queries to the random oracle is bounded by some polynomial function in κ, i.e. $\mathrm{poly}(\kappa)$. The seeds are of length $2 \cdot \kappa$ bits, so the chance that the adversary queries the random oracle on an input before the seed agreement phase on a query that must be programmed with output from $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ is $\mathrm{poly}(\kappa) \cdot (2 \cdot 2^{2 \cdot \kappa})^{-1} < 2^{-\kappa}$ by the Birthday Bound (see the proof of Theorem 2.3). Thus no environment can distinguish except with negligible probability in the computational security parameter. □

6.3.2 PRZSs

In order to rerandomize additive sharings, which is required for proving the security of the MPC protocol defined later, parties can add a uniformly-random sharing of zero. Such sharings are provided by the functionality $\mathcal{F}_{\mathsf{RZS}}$ given in Figure 6.2. A protocol Π_{RZS} realizing $\mathcal{F}_{\mathsf{RZS}}$ in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model is given in Figure 6.5.

This protocol was first described by Gilboa and Ishai [GI99], but instead of using random oracles, there they altered the security notion to define a class of protocols that involved agreeing on a random seed and then extending it locally. The reason for this change in security definition is that there is some "short explanation" – namely, the seeds – that "explains" the final random string that is given as output by any set of parties, which means that simulation is not possible. Intuitively, one can see this as $\mathcal{S}$ having to find a seed that produces the uniform output of $\mathcal{F}_{\mathsf{RZS}}$, breaking the security of

the PRF, in order to send $\mathcal{A}$ information so that the environment would not know that the random outputs of honest parties were not generated by evaluating a PRF. This was the same problem encountered by Boyle et al. [BCGI18], again avoided by carefully choosing the definition of security for the primitive. Since the protocol assumes the use of random oracles in other places (for example, in realizing efficient UC commitments), its use here does not change the set of security assumptions.

Protocol Π_{RZS}

This protocol is realized in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{RO}}$-hybrid model.

Initialize

1. Agree on a session identifier sid.
2. Call an instance of $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, \mathbb{F}, sid)$.
3. Each unordered pair of parties $(\mathcal{P}_i, \mathcal{P}_j)$ (i.e. for all $(i,j) \in [n]^2$ such that $(i < j)$) does the following:
   a) Call an instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \{0,1\}^{2\cdot\kappa}, sid_{i,j})$; let this instance be denoted by $\mathcal{F}^{i,j}_{\mathsf{CoinFlip}}$.
   b) Call $\mathcal{F}^{i,j}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid_{i,j})$ and set the returned output as $k_{i,j}$.
   c) Call $\mathcal{F}^{i,j}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid_{i,j})$ and set the returned output as $k_{j,i}$.

Sharing of Zero To obtain a sharing of zero amongst parties indexed by $X \subseteq [n]$, for each $i \in X$,

1. The parties compute a new identifier id_r.
2. Each party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{RO}}$ with input $(id_r \| k_{i,j}, sid)$ and set the returned output as $r_{i,j}$.
3. Each party $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{RO}}$ with input $(id_r \| k_{j,i}, sid)$ and set the returned output as $r_{j,i}$.
4. Party $\mathcal{P}_i$ sets
$$[\![r]\!]^{\mathtt{A}}_{\mathcal{P}_i} := \sum_{j \in X \setminus \{i\}} r_{i,j} - r_{j,i}.$$

Figure 6.5: Protocol for Secret-Sharings of Zero, Π_{RZS}.

Theorem 6.2. *The protocol Π_{RZS} UC-securely realizes the functionality $\mathcal{F}_{\mathsf{RZS}}$ against a static, active, computationally-bounded adversary in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model.*

Proof. The simulator is given in Figure 6.6. For correctness, observe that

$$\begin{aligned}
\sum_{i\in[n]\setminus A} [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \sum_{i\in A}\sum_{j\neq i}(r_{i,j} - r_{j,i}) &= \sum_{i\in[n]\setminus A} [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \sum_{i\in A}\left(\sum_{j\neq i} r_{i,j} - \sum_{j\neq i} r_{j,i}\right) \\
&= \sum_{i\in[n]\setminus A} [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \sum_{i\in A}\left(\sum_{j\neq i} r_{i,j} - \sum_{j\neq i}(r_{i,j} - t_{i,j})\right) \\
&= \sum_{i\in[n]\setminus A} [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \sum_{i\in A}\sum_{j\neq i} t_{i,j} = \sum_{i\in[n]\setminus A} [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} + \sum_{i\in A} [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} = 0
\end{aligned}$$

(where for simplicity it is assumed that $X = [n]$).

Simulator $\mathcal{S}_{\mathsf{RZS}}$

Initialize

1. Agree on a session identifier, sid, with $\mathcal{A}$.
2. Await the call to $\mathcal{F}_{\mathsf{RO}}$ with input $(\texttt{Initialize}, \mathbb{F}, sid)$ from $\mathcal{A}$ and initialize a local instance.
3. Each unordered pair of parties $(\mathcal{P}_i, \mathcal{P}_j)$ (i.e. for all $(i,j) \in [n]^2$ such that $(i < j)$), where $i \in A$ or $j \in A$, do the following:
   a) If $i \in A$ or $j \in A$, await a call to an instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \{0,1\}^{2\cdot\kappa}, sid_{i,j})$ and initialize a copy locally. This instance is denoted by $\mathcal{F}^{i,j}_{\mathsf{CoinFlip}}$.
   b) Await the call to $\mathcal{F}^{i,j}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid_{i,j})$, execute it honestly, set the returned output as $k_{i,j}$ and send this to $\mathcal{A}$.
   c) Await the call to $\mathcal{F}^{i,j}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid_{i,j})$, execute it honestly, set the returned output as $k_{j,i}$ and send this to $\mathcal{A}$.

Sharing of Zero

1. Compute a new identifier id_r, call $\mathcal{F}_{\mathsf{RZS}}$ with input $(\texttt{SecretSharing}, id_r, sid)$ and await a share vector $[\![t]\!]^{\mathtt{A}}_{\mathcal{A}}$ in response.
2. For each $i \in X \cap A$, await the call to $\mathcal{F}_{\mathsf{RO}}$ with input $(id_r \| k_{i,j}, sid)$ for each $j \in X$, execute it honestly, set the returned output as $r_{i,j}$ and send this to $\mathcal{A}$.
3. For each $i \in X \cap A$, await the call to $\mathcal{F}_{\mathsf{RO}}$ with input $(id_r \| k_{j,i}, sid)$ for each $j \in X$ and do the following:
   - Sample a set $\{t_{i,j}\}_{j \in X} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{j \neq i} t_{i,j} = [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}$.
   - Set $r_{j,i} := r_{i,j} - t_{i,j}$.
   - Send $r_{j,i}$ to $\mathcal{A}$.
4. (No simulation is required for this step.)

Figure 6.6: Simulator $\mathcal{S}_{\mathsf{RZS}}$ for $\mathcal{F}_{\mathsf{RZS}}$.

The only messages in the transcript are from the random oracle, $(r_{i,j}, r_{j,i})$. Each $r_{j,i}$ can be viewed as $-t_{i,j}$ encrypted via the one-time pad $r_{i,j}$. Thus the only information learnt from such a pair is $t_{i,j}$. Since the $t_{i,j}$'s are uniformly random subject to the contraint that $\sum_{j \neq i} t_{i,j} = [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}$, and the $[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}$'s are uniformly random subject to the constraint that they sum to 0, the distribution of the simulated view is identical to the real view of $\mathcal{A}$.

A similar argument as for the proof of Theorem 6.1 can be used to show that the environment cannot distinguish between worlds by querying the random oracle except with negligible advantage. □

6.3.3 Communication Complexity

The total theoretical communication cost (in bits) for realizing the seed-agreement is given in Table 6.1, where $\mathsf{SC}(E_{\mathsf{BC}})$ and $\mathsf{AC}(E_{\mathsf{BC}})$ denote a complete network of secure and authenticated channels, respectively. The cost of instantiating the broadcast is not included. The communication cost of commitments comes from the instantiation of $\mathcal{F}_{\mathsf{Commit}}$ via Π_{Commit} in Figure 2.7. Furthermore, it is assumed that the functionality $\mathcal{F}_{\mathsf{CoinFlip}}$ defined in Figure 2.8 is realized using the protocol Π_{CoinFlip} in Figure 2.9. Thus in total, the cost (for each party in each pair, and for every party for every set in $\mathcal{G} \in \nabla$) is that of a commitment, which is an element of $\{0,1\}^{2\cdot\kappa}$, and the decommitment, which is a message that consists of the seed in $\{0,1\}^{2\cdot\kappa}$ with the nonce in $\{0,1\}^{\kappa}$.

Procedure	Number of bits	Channels
PRZS key commitments	$2\cdot\kappa\cdot n\cdot(n-1)$	$\mathsf{AC}(E_{\mathsf{BC}})$
Opening commitments	$3\cdot\kappa\cdot n\cdot(n-1)$	$\mathsf{SC}(E_{\mathsf{BC}})$
PRSS key commitments	$2\cdot\kappa\cdot\sum_{\mathcal{G}\in\nabla}\lvert\mathcal{G}\rvert\cdot(\lvert\mathcal{G}\rvert-1)$	$\mathsf{AC}(E_{\mathsf{BC}})$
Opening commitments	$3\cdot\kappa\cdot\sum_{\mathcal{G}\in\nabla}\lvert\mathcal{G}\rvert\cdot(\lvert\mathcal{G}\rvert-1)$	$\mathsf{SC}(E_{\mathsf{BC}})$

Table 6.1: Total communication cost to realize $\mathcal{F}^{\mathsf{R}}_{\mathsf{RSS}}$ and $\mathcal{F}_{\mathsf{RZS}}$.

6.4 Converting Additive to Replicated

Replicated secret-sharing is always multiplicative if the access structure is $\mathscr{Q}_2$, as discussed in Section 6.2. The real cost in computing a passive multiplication of secrets shared using replicated secret-sharing is that of converting the additive sharing of the product back into a replicated sharing. Indeed, at its heart, the protocol of Araki et al. [ABF+17] is an efficient method of turning an additive sharing into a replicated sharing.

This is analogous to the main communication cost in the BGW protocol, where the task is to convert what is essentially an additive sharing (modulo the Lagrange interpolation coefficients) back to a Shamir sharing of the secret. Given a procedure that performs this conversion, it is straightforward to give a subprotocol in which parties multiply two secrets. For now, it suffices to understand that it is desirable to have a methodology for doing this; the specifics of the multiplication are dealt with in detail in the following section.

From a high level, additive secret-sharing and replicated secret-sharing look very similar: to share a secret, the dealer additively splits the secret into several shares and

distributes them amongst the parties; indeed, additive secret-sharing is exactly replicated secret-sharing for a full-threshold access structure. In this section, this similarity is exploited to give a more communication efficient (but only computationally-secure) method of converting additive sharings to replicated sharings; first, for comparison, the "standard" (i.e. IT) method is described.

6.4.1 Information-Theoretic Conversion

The subprotocol given in Figure 6.7 is part of Maurer's protocol [Mau06] and is similar to the conversion in the protocol of Beaver and Wool [BW98] but uses replicated secret-sharing rather than disjunctive normal form (DNF)-based secret-sharing. It is expressed in its general form that turns an additive secret-sharing of a secret into a secret under *any other* LSSS.

Subprotocol $\Pi_{\mathtt{AToAny}}$

Parties hold $[\![v]\!]^{\mathtt{A}}$ and will convert to a sharing under another LSSS, denoted by $[\![\cdot]\!]$.

Additive to Replicated

1. Each $\mathcal{P}_i$ creates a sharing $[\![[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]$ of the value $[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}$.
2. Each $\mathcal{P}_i$ acts as the dealer in the LSSS and for each $j \neq i$ sends $[\![[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]_{\mathcal{P}_j}$ to $\mathcal{P}_j$ over a secure channel.
3. Each $\mathcal{P}_i$ computes $[\![v]\!]_{\mathcal{P}_i} := \sum_{j\in[n]} [\![[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_j}]\!]_{\mathcal{P}_i}$.
4. Parties (locally) output $[\![v]\!]$.

Figure 6.7: Protocol to Convert Additive Shares to Shares Under Any LSSS, $\Pi_{\mathtt{AToAny}}$.

Correctness

To see that the protocol is correct, observe that by linearity of the LSSS,

$$\sum_{i=1}^{n} [\![[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]^{\mathtt{R}} = [\![\sum_{i=1}^{n} [\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]^{\mathtt{R}} = [\![v]\!]^{\mathtt{R}}.$$

Security

Security holds by the IT security of the LSSS.

Communication Complexity

In order to reshare $[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}$, each party $\mathcal{P}_i$ sends a total of

$$\sum_{\mathcal{G}\in\nabla:\mathcal{G}\ni\mathcal{P}_i}(|\mathcal{G}|-1)+\sum_{\mathcal{G}\in\nabla:\mathcal{G}\not\ni\mathcal{P}_i}|\mathcal{G}|=\sum_{\mathcal{G}\in\nabla}|\mathcal{G}|-\sum_{\mathcal{G}\in\nabla:\mathcal{G}\ni\mathcal{P}_i}1$$

finite-field elements, and communicates with every other party over a point-to-point secure channel. Thus the total amount of data sent across all parties in a single resharing is

$$\sum_{i=1}^{n}\left(\sum_{\mathcal{G}\in\nabla}|\mathcal{G}|-\sum_{\mathcal{G}\in\nabla:\mathcal{G}\ni\mathcal{P}_i}1\right)=(n-1)\cdot\sum_{\mathcal{G}\in\nabla}|\mathcal{G}|$$

finite-field elements, across $n\cdot(n-1)$ secure channels (assuming a non-redundant access structure). These secure channels are enumerated as uni-directional secure channels, reflecting the fact that good security practice dictates that parties should have different secret keys securing communication in different directions.

For the running example, Example 6.1, this translates to sending $(6-1)\cdot 41 = 205$ finite-field elements over $6\cdot 5 = 30$ secure channels. Note that the same finite-field element will be sent to multiple parties (every set of parties $\mathcal{G}\in\nabla$ obtains a share common to them all), but these elements are counted as distinct when analysing communication costs.

6.4.2 Computational Conversion

The goal is to turn an additive sharing $[\![v]\!]^{\mathtt{A}}$ into a replicated sharing $[\![v]\!]^{\mathtt{R}}$ (with passive security) more efficiently than in the IT protocol above. The protocol is given in Figure 6.9 but the method is first outlined here.

To do this conversion, once at the beginning of the protocol, each party is assigned a set $\mathcal{G}_i\in\nabla$ where $i\in\mathcal{G}_i$ so that no two parties are assigned the same set. (Section 6.7 describes how to fix the protocol in the uncommon event that no such assignment can be made.)

To turn an additive sharing $[\![v]\!]^{\mathtt{A}}$ into a replicated sharing, the parties first rerandomize the additive sharing $[\![v]\!]^{\mathtt{A}}$ using a PRZS, by each party $\mathcal{P}_i$ computing $[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}+[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}$, and then set this to be the share $v^{\mathtt{R}}_{\mathcal{G}_i}$.

To make up the remaining $|\nabla|-n$ shares, the parties set any share not indexed by some set in $\{\mathcal{G}_i\}_{i\in[n]}$ to be the corresponding random share $r^{\mathtt{R}}_{\mathcal{G}}$ from a PRSS $[\![r]\!]^{\mathtt{R}}$. Thus, at this point, the replicated sharing is

$$\sum_{\mathcal{G}\in\nabla}v^{\mathtt{R}}_{\mathcal{G}}=\sum_{i\in[n]}v^{\mathtt{R}}_{\mathcal{G}_i}+\sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}}r^{\mathtt{R}}_{\mathcal{G}}=\sum_{i\in[n]}\left([\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i}+[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}\right)+\sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}}r^{\mathtt{R}}_{\mathcal{G}}=v+\sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}}r^{\mathtt{R}}_{\mathcal{G}}.$$

In other words, the current sharing is offset from v by the sum of the random shares taken from $[\![r]\!]^{\mathtt{R}}$. To resolve this error, shares from r are subtracted from the $v^{\mathtt{R}}_{\mathcal{G}_i}$ shares so that the sum of all shares is still v. To ensure each random share from $[\![r]\!]^{\mathtt{R}}$ is only subtracted once, a partition $\{\nabla^{(i)}\}_{i\in[n]}$ is computed so that $\mathcal{P}_i$ subtracts all shares $\{r^{\mathtt{R}}_{\mathcal{G}}\}_{\mathcal{G}\in\nabla^{(i)}}$ of the PRSS from $v^{\mathtt{R}}_{\mathcal{G}_i}$. All that remains to obtain a replicated sharing is for each $\mathcal{P}_i$ to send $v^{\mathtt{R}}_{\mathcal{G}_i}$ to every party who is supposed to hold it, since the parties already hold the share $v^{\mathtt{R}}_{\mathcal{G}} := r^{\mathtt{R}}_{\mathcal{G}}$ for all $\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}$. Using the PRZS combined with the PRSS means that the resulting sharing is uniform in the space of all share vectors $[\![v]\!]^{\mathtt{R}}$ that share the secret v.

In the running example one could take $\mathcal{G}_1 := \{1,3,4\}$, $\mathcal{G}_2 := \{1,2,4\}$, $\mathcal{G}_3 := \{1,2,3\}$, $\mathcal{G}_4 := \{2,3,4,5\}$, $\mathcal{G}_5 := \{1,2,5,6\}$ and $\mathcal{G}_6 := \{2,3,4,6\}$ and take the partition as follows:

$$\begin{aligned}
\nabla^{(1)} &= \{\{1,3,4\}\},\\
\nabla^{(2)} &= \{\{1,2,4\}\},\\
\nabla^{(3)} &= \{\{1,2,3\}\},\\
\nabla^{(4)} &= \{\{2,3,4,5\}\},\\
\nabla^{(5)} &= \{\{1,2,5,6\},\ \{1,3,5,6\},\ \{1,4,5,6\}\},\\
\nabla^{(6)} &= \{\{2,3,4,6\},\ \{2,3,5,6\},\ \{2,4,5,6\},\ \{3,4,5,6\}\}.
\end{aligned}$$

This results in shares being generated and sent as shown in Figure 6.8.

	$[\![v]\!]^{\mathtt{A}}$	$[\![t]\!]^{\mathtt{A}}$	$[\![v]\!]^{\mathtt{R}}$										
			123	124	134	1256	1356	1456	2345	2346	2356	2456	3456
$\mathcal{P}_1$	$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_1}$	$[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_1}$	◇	◇	□	◇	⊞	⊞					
$\mathcal{P}_2$	$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_2}$	$[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_2}$	◇	□		◇			◇	◇	⊞	⊞	
$\mathcal{P}_3$	$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_3}$	$[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_3}$	□		◇		⊞		◇	◇	⊞		⊞
$\mathcal{P}_4$	$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_4}$	$[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_4}$		◇	◇			⊞	□	◇		⊞	⊞
$\mathcal{P}_5$	$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_5}$	$[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_5}$				□	⊠	⊠	◇		⊞	⊞	⊞
$\mathcal{P}_6$	$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_6}$	$[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_6}$				◇	⊞	⊞		□	⊠	⊠	⊠

Key

◇	Receive share
□	Send share; defined by $\mathcal{G}_i$; computed as $[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i} + [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} - \sum$ ⊠'s in same row
⊞, ⊠	Evaluate PRF

Figure 6.8: Optimized Multiplication in Running Example.

This method directly generalizes the method used by [AFL+16], which concentrated on the case of the finite field $\mathbb{F}_2$ and a $(3,1)$-threshold, in the sense that this protocol is the same as theirs in this 3-party setting. However, prior to this new methodology it was not clear how to generate the remaining $|\nabla| - n$ shares and maintain correctness, and without incurring considerable additional communication overhead, since in the 3-party case $|\nabla| - n = 0$.

Choosing the Partition

After the sets $\{\mathcal{G}_i\}_{i\in[n]}$ are chosen, the remaining shares are assigned to the parties arbitrarily, where choices are made so that each party is assigned roughly the same overall total number of shares in order to balance the load.

Any partition $\{\nabla^{(i)}\}_{i\in[n]}$ of the set ∇ can be chosen with the constraint that for every $i \in [n]$ it holds that $\mathcal{G} \in \nabla^{(i)}$ implies $\mathcal{P}_i \in \mathcal{G}$. It is assumed that $\nabla^{(i)} \neq \varnothing$ for all $i \in [n]$, which may not always be possible, although there is an easy fix to the protocol in the unlikely event that this happens, described in Section 6.7.

The partition is chosen by considering all the maps $f : \nabla \to \mathcal{P}$ such that for every $i \in [n]$, $f(\mathcal{G}) = \mathcal{P}_i$ implies $\mathcal{P}_i \in \mathcal{G}$, and choosing a f such that $\mathrm{im}(f)$ is as large as possible. If f is not surjective then there is at least one set $f^{-1}(\{\mathcal{P}_i\})$ (for some i) which is empty. For small numbers of parties on a non-redundant $\mathscr{Q}_2$ access structure, such a map can always be found; the necessary adaptation to the protocol when this is not the case, and further relevant discussion, is given in Section 6.7.

The formal description of the subprotocol is given in Figure 6.9 and is followed by a justification of its correctness and security (in the passive security setting).

Subprotocol $\Pi_{\mathtt{AToROpt}}$

It is assumed that, as part of the larger protocol, parties have already called an instance of $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{R}}$ with input $(\mathtt{Initialize}, \Gamma, sid)$ and an instance of $\mathcal{F}_{\mathsf{RZS}}$ with input $(\mathtt{Initialize}, \mathcal{P}, sid)$. Parties hold an additive sharing $[\![v]\!]^{\mathtt{A}}$.

Additive to Replicated

1. The parties compute a new identifier id_t, call $\mathcal{F}_{\mathsf{RZS}}$ with input $(\mathtt{ZeroSharing}, id_t, \mathcal{P}, sid)$, and receive an additive sharing of zero $[\![t]\!]^{\mathtt{A}}$.
2. The parties compute a new identifier id_r, call $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{R}}$ with input $(\mathtt{SecretSharing}, id_r, sid)$, and receive $[\![r]\!]^{\mathtt{R}}$.[a]
3. Each $\mathcal{P}_i$ defines a sharing $[\![v]\!]^{\mathtt{R}}$ by doing the following:
   a) Set $v_{\mathcal{G}}^{\mathtt{R}} := r_{\mathcal{G}}^{\mathtt{R}}$ for all $\mathcal{G} \in \nabla \setminus \{\mathcal{G}_j\}_{j\in[n]}$ where $\mathcal{G} \ni \mathcal{P}_i$.
   b) Set $v_{\mathcal{G}_i}^{\mathtt{R}} := [\![v]\!]_{\mathcal{P}_i}^{\mathtt{A}} + [\![t]\!]_{\mathcal{P}_i}^{\mathtt{A}} - \sum_{\mathcal{G}\in f^{-1}(\{\mathcal{P}_i\})\setminus\{\mathcal{G}_i\}} r_{\mathcal{G}}^{\mathtt{R}}$.
4. For every $i \in [n]$, for every $\mathcal{P}_j \in \mathcal{G}_i$, $\mathcal{P}_i$ sends $v_{\mathcal{G}_i}^{\mathtt{R}}$ to $\mathcal{P}_j$ over a secure channel.
5. Each party $\mathcal{P}_i$ concatenates $\{v_{\mathcal{G}}^{\mathtt{R}} : \mathcal{G} \in \nabla \wedge \mathcal{G} \ni \mathcal{P}_i\}$ into a share vector $[\![v]\!]_{\mathcal{P}_i}^{\mathtt{R}}$.
6. Parties (locally) output $[\![v]\!]^{\mathtt{R}}$.

[a] Note that in the instantiation of $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{R}}$ via $\Pi_{\mathsf{RSS}}^{\mathtt{R}}$, the parties need not to call the random oracle on the keys $\{k_{\mathcal{G}_i}\}_{i=1}^{n}$ – only on the other keys – since these shares are discarded in the next step.

Figure 6.9: Optimized Protocol to Convert Additive Shares to Shares Under Replicated Secret-Sharing, $\Pi_{\mathtt{AToROpt}}$.

Correctness

In Step 3a, $\mathcal{P}_i$ computes all shares not assigned to a party by $\{\nabla^{(j)}\}_{j\in[n]}$; in Step 3b the share $v_{\nabla^{(i)}}^{\mathtt{R}}$ is computed; and in Step 4, $\mathcal{P}_i$ receives a share for every $\mathcal{G} \in \nabla$ where $G \ni i$, as is required by replicated secret-sharing.

To see that the replicated sharing shares the correct secret, observe that

$$
\begin{aligned}
\sum_{\mathcal{G}\in\nabla} v_{\mathcal{G}}^{\mathtt{R}} &= \sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}} v_{\mathcal{G}}^{\mathtt{R}} + \sum_{\mathcal{G}\in\{\mathcal{G}_i\}_{i\in[n]}} v_{\mathcal{G}}^{\mathtt{R}} \\
&= \sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}} r_{\mathcal{G}}^{\mathtt{R}} + \sum_{i\in[n]} \left([\![v]\!]_{\mathcal{P}_i}^{\mathtt{A}} + [\![t]\!]_{\mathcal{P}_i}^{\mathtt{A}} - \sum_{\mathcal{G}\in f^{-1}(\{\mathcal{P}_i\})\setminus\{\mathcal{G}_i\}} r_G^{\mathtt{R}} \right) \\
&= \sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}} r_{\mathcal{G}}^{\mathtt{R}} + \sum_{i\in[n]} [\![v]\!]_{\mathcal{P}_i}^{\mathtt{A}} + \sum_{i\in[n]} [\![t]\!]_{\mathcal{P}_i}^{\mathtt{A}} - \sum_{i\in[n]} \sum_{\mathcal{G}\in f^{-1}(\{\mathcal{P}_i\})\setminus\{\mathcal{G}_i\}} r_G^{\mathtt{R}} \\
&= \sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}} r_{\mathcal{G}}^{\mathtt{R}} + \sum_{i\in[n]} [\![v]\!]_{\mathcal{P}_i}^{\mathtt{A}} + 0 - \sum_{\mathcal{G}\in\nabla\setminus\{\mathcal{G}_i\}_{i\in[n]}} r_G^{\mathtt{R}} \\
&= v
\end{aligned}
$$

where the penultimate equality holds because $f^{-1}(\mathcal{P}) = \nabla$ and $f(\mathcal{G}_i) = \mathcal{P}_i$ for all $i \in [n]$ (so $\mathcal{G}_i$ is not contained in $f^{-1}(\{\mathcal{P}_j\})$ for any $j \neq i$).

Security

The key observation for security is that the PRZS masks the local computations: any unqualified set of parties $\mathcal{U}$ is missing at least one party, and hence one share of the PRZS, which means that the shares $\{v^{\mathbb{R}}_{\mathcal{G}_i}\}_{i\in[n]}$ are indistinguishable from uniformly random. All other computations are local and so security follows assuming a secure protocol for realizing $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ and $\mathcal{F}_{\mathsf{RZS}}$ is given.

6.5 Passively-Secure $\mathscr{Q}_2$ MPC Protocol

In this section the optimized (i.e. computationally-secure) variant of the additive-to-replicated conversion is bootstrapped to a full MPC protocol. Linear operations are straightforward; the passive multiplication procedure presented here starts as in Maurer's protocol – by forming an additive secret sharing of the product, which is always possible with replicated secret-sharing realizing a $\mathscr{Q}_2$ access structure – but now the optimized computational conversion procedure is used to convert the product to a replicated sharing. The passively-secure arithmetic black box $\mathcal{F}_{\mathsf{PABB}}$, which is the goal of passively-secure MPC, is given in Figure 6.10.

Functionality $\mathcal{F}_{\mathsf{PABB}}$

Initialize On input $(\texttt{Initialize}, \mathbb{F}, sid)$ from all parties, store $\mathbb{F}$ and initialize a new database DB with indexing set DB.Ids and store the field as $\texttt{DB.Field} := \mathbb{F}$.

Input On input $(\texttt{Input}, i, id, x, sid)$ from party $\mathcal{P}_i$ and $(\texttt{Input}, i, id, \perp, sid)$ from all other parties, where $i \in [n]$, id is a new identifier, and $x \in \texttt{DB.Field}$, set $\texttt{DB}[id] := x$ and insert id into DB.Ids.

Add On input $(\texttt{Add}, id_x, id_y, id_z, sid)$ from all parties, if $id_x, id_y \in \texttt{DB.Ids}$ and id_z is a new identifier, set $\texttt{DB}[id_z] := \texttt{DB}[id_x] + \texttt{DB}[id_y]$ and insert id_z into DB.Ids.

Multiply On command $(\texttt{Multiply}, id_x, id_y, id_z, sid)$ from all parties, if $id_x, id_y \in \texttt{DB.Ids}$ are present in memory and id_z is a new identifier, store $\texttt{DB}[id_z] := \texttt{DB}[id_x] \cdot \texttt{DB}[id_y]$ and insert id_z into DB.Ids.

Output To One On input $(\texttt{Output}, i, id, sid)$ from all parties where $id \in \texttt{DB.Ids}$ and $i \in [n]$, send $\texttt{DB}[id]$ to $\mathcal{P}_i$ and continue.

Output To All On input $(\texttt{Output}, 0, id, sid)$ from all parties, if $id \in \texttt{DB.Ids}$, send $\texttt{DB}[id]$ to all parties and continue.

Figure 6.10: Passive Arithmetic Black Box Functionality, $\mathcal{F}_{\mathsf{PABB}}$.

6.5.1 Multiplication and Input using Conversion

Multiplication

The real difficulty in creating an MPC protocol given a LSSS is in performing secure multiplication of secret-shared values, $[\![x]\!]^{\mathtt{R}}$ and $[\![y]\!]^{\mathtt{R}}$. With this goal, we begin by following [BW98] and define a surjective function $\mathsf{I} : \nabla^2 \to \mathcal{P}$ (**I**ntersection map) such that $\mathsf{I}(\mathcal{G}_1, \mathcal{G}_2) = \mathcal{P}_i$ implies that $\mathcal{P}_i \in \mathcal{G}_1 \cap \mathcal{G}_2$; the existence of such a function follows from the fact that the access structure is $\mathscr{Q}_2$, as discussed in Section 6.2. Note that there are possibly multiple choices for I. Note also that party $\mathsf{I}(\mathcal{G}_1, \mathcal{G}_2)$ holds a copy of share $x^{\mathtt{R}}_{\mathcal{G}_1}$ and $y^{\mathtt{R}}_{\mathcal{G}_2}$. The party indexed by $\mathsf{I}(\mathcal{G}_1, \mathcal{G}_2)$ will be put "in charge" of computing the cross term $x^{\mathtt{R}}_{\mathcal{G}_1} \cdot y^{\mathtt{R}}_{\mathcal{G}_2}$ in the multiplication protocol. Then each $\mathcal{P}_i$ (locally) computes

$$[\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i} := \sum_{(\mathcal{G}_1, \mathcal{G}_2) \in \mathsf{I}^{-1}(\{\mathcal{P}_i\})} x^{\mathtt{R}}_{\mathcal{G}_1} \cdot y^{\mathtt{R}}_{\mathcal{G}_2},$$

whence the parties can execute $\Pi_{\mathtt{AToROpt}}$.

Input

The "standard" way for a party $\mathcal{P}_i$ to provide input x is to generate a sharing $[\![x]\!]^{\mathtt{R}}$ and distribute the shares. In the protocol given here, it is also possible to make use of the conversion subprotocol $\Pi_{\mathtt{AToROpt}}$ for parties to provide inputs, at no extra setup cost: the parties take an additive sharing of zero $[\![t]\!]^{\mathtt{A}}$, and then $\mathcal{P}_i$ sets $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_i} := x + [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}$ and all other parties $\mathcal{P}_j$ ($j \neq i$) set $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_j} := [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_j}$, and then they run the $\Pi_{\mathtt{AToROpt}}$ subprotocol. In practice, this requires obtaining one PRZS and one PRSS which means the parties have to perform several PRF evaluations, so as the number of parties grows (and the number of required PRF evaluations grows exponentially) there may be a point at which it is beneficial to provide input using the standard approach.

It is crucial for the UC-security of this protocol that the simulator should be able to extract the inputs of the adversary in this optimized subprotocol. Fortunately, this is indeed the case as every share is held by at least one honest party, including the shares indexed by $\mathcal{G}_i$ for all $i \in A$, which are the shares actually "containing" the corrupt party's input in this optimized input subprotocol.

The protocol $\Pi^{\mathscr{Q}_2,\mathtt{R}}_{\mathsf{Online}}$ is given in Figure 6.11. This protocol is the analogue of [AFL+16] for arbitrary $\mathscr{Q}_2$ access structures and arbitrary finite fields.

Protocol $\Pi^{\mathscr{Q}_2,\mathtt{R}}_{\mathsf{Online}}$

This protocol is realized in the $\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$,$\mathcal{F}_{\mathsf{RZS}}$-hybrid model and makes use of the subprotocol $\Pi_{\mathtt{AToROpt}}$.

Initialize

1. The parties agree on a session identifer sid.
2. The parties call $\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$ with input $(\mathtt{Initialize}, \Gamma, sid)$.
3. The parties call $\mathcal{F}_{\mathsf{RZS}}$ with input $(\mathtt{Initialize}, \mathcal{P}, sid)$.

Input For $\mathcal{P}_i$ to provide input x, the parties compute a new identifier id_x and do the following:

1. Party $\mathcal{P}_i$ sets $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_i} := x$, and for each $j \neq i$, $\mathcal{P}_j$ sets $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_j} := 0$.
2. The parties run the subprotocol $\Pi_{\mathtt{AToROpt}}$ on $[\![x]\!]^{\mathtt{A}}$ to obtain a sharing $[\![x]\!]^{\mathtt{R}}$.

Add To add secrets $[\![x]\!]^{\mathtt{R}}$ and $[\![y]\!]^{\mathtt{R}}$, the parties and compute $[\![x]\!]^{\mathtt{R}} + [\![y]\!]^{\mathtt{R}}$ to obtain a sharing $[\![z]\!]^{\mathtt{R}}$.

Multiply To multiply secrets $[\![x]\!]^{\mathtt{R}}$ and $[\![y]\!]^{\mathtt{R}}$, the parties compute a new identifier id_z and do the following:

1. For each $i \in [n]$, $\mathcal{P}_i$ computes

$$[\![z]\!]^{\mathtt{A}}_{\mathcal{P}_i} := \sum_{(\mathcal{G}_1,\mathcal{G}_2)\in \mathsf{I}^{-1}(\{\mathcal{P}_i\})} x^{\mathtt{R}}_{\mathcal{G}_1} \cdot y^{\mathtt{R}}_{\mathcal{G}_2}.$$

2. The parties run the subprotocol $\Pi_{\mathtt{AToROpt}}$ on $[\![z]\!]^{\mathtt{A}}$ to obtain a sharing $[\![z]\!]^{\mathtt{R}}$.

Output To One To output a secret with identifier id_x to $\mathcal{P}_i$, the parties do the following:

1. For all $j \in [n]$, for all $\mathcal{G} \in \nabla^{(j)}$, for all $i \in [n]$, if $\mathcal{P}_i \notin \mathcal{G}$ then $\mathcal{P}_j$ sends $x^{\mathtt{R}}_{\mathcal{G}}$ to $\mathcal{P}_i$ over a secure channel.
2. Party $\mathcal{P}_i$ computes $x := \sum_{\mathcal{G}\in\nabla} x^{\mathtt{R}}_{\mathcal{G}}$.

Output To All To output a secret with identifier id_x to all parties, the parties do the following:

1. For all $i \in [n]$, for all $\mathcal{G} \in \nabla^{(i)}$, for all $j \in [n]$, if $\mathcal{P}_j \notin \mathcal{G}$ then $\mathcal{P}_i$ sends $x^{\mathtt{R}}_{\mathcal{G}}$ to $\mathcal{P}_j$ over an authenticated channel.
2. All parties compute $x := \sum_{\mathcal{G}\in\nabla} x^{\mathtt{R}}_{\mathcal{G}}$.

Figure 6.11: Online Protocol for a $\mathscr{Q}_2$ Access Structure Using Replicated Secret-Sharing, $\Pi^{\mathscr{Q}_2,\mathtt{R}}_{\mathsf{Online}}$.

6.5.2 Correctness

Correctness of **Input** and **Multiply** follows from the outlines given in Section 6.5.1. Correctness of **Add** follows by the linearity of the LSSS and of **Output** by the correctness of reconstruction.

6.5.3 Security

Theorem 6.3. *Let Γ be a non-redundant $\mathscr{Q}_2$ access structure and let $\{\nabla^{(i)}\}_{i\in[n]}$ be a partition of the set ∇ as defined above. Then the protocol $\Pi^{\mathscr{Q}_2,\mathtt{R}}_{\mathsf{Online}}$ UC-securely realizes the functionality $\mathcal{F}_{\mathsf{PABB}}$ against a static, passive adversary, in the $\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$,$\mathcal{F}_{\mathsf{RZS}}$-hybrid model.*

The alterations to the protocol for when there is no surjective partition are discussed in Section 6.7.

Proof. The simulator is given in Figure 6.12 and the transcript in Figure 6.13.

Simulator $\mathcal{S}_{\mathsf{PABB}}$

Since the subprotocol $\Pi_{\mathtt{AToROpt}}$ is used in **Input** and **Multiply**, a macro SAToR() is given for the simulation (at the end).

Initialize

1. Agree on some sid with $\mathcal{A}$.
2. Await the call to $\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$ with input $(\mathtt{Initialize},\Gamma,sid)$ and initialize a local instance.
3. Await the call to $\mathcal{F}_{\mathsf{RZS}}$ with input $(\mathtt{Initialize},\mathcal{P},sid)$ and initialize a local instance.

Input When party $\mathcal{P}_i$ is to provide input, the simulator computes a new identifier id_x and does the following:

1. Set $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_i} := 0$ for all $i \in [n]\setminus A$ on behalf of (emulated) honest parties.
2. Execute $\mathtt{SAToR}(id_x)$ to obtain x.
   If $i \in A$, then call $\mathcal{F}_{\mathsf{PABB}}$ with input $(\mathtt{Input},i,id,x,sid)$ on behalf of $\mathcal{P}_i$ and $(\mathtt{Input},i,id,\perp,sid)$ on behalf of all $\mathcal{P}_j \in \mathcal{A}\setminus\{\mathcal{P}_i\}$.
   If $i \in [n]\setminus A$, then call $(\mathtt{Input},i,id,\perp,sid)$ on behalf of all $\mathcal{P}_j \in \mathcal{A}$.

Add To add $[\![x]\!]^{\mathtt{R}}$ and $[\![y]\!]^{\mathtt{R}}$, compute a new identifier id_z, set $[\![z]\!]^{\mathtt{R}} := [\![x]\!]^{\mathtt{R}} + [\![y]\!]^{\mathtt{R}}$, and call $\mathcal{F}_{\mathsf{PABB}}$ with input $(\mathtt{Add},id_x,id_y,id_z,sid)$.

Multiply To multiply $x^{\mathtt{R}}$ with $y^{\mathtt{R}}$, the simulator computes a new identifier id_z and then does the following:

1. On behalf of each (emulated) honest party, $\mathcal{P}_i \in \mathcal{P}\setminus\mathcal{A}$, compute
$$[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i} := \sum_{(\mathcal{G}_1,\mathcal{G}_2)\in \mathsf{I}^{-1}(\{\mathcal{P}_i\})} x^{\mathtt{R}}_{\mathcal{G}_1}\cdot y^{\mathtt{R}}_{\mathcal{G}_2}.$$
2. Execute $\mathtt{SAToR}([\![v]\!]^{\mathtt{A}})$ and then call $\mathcal{F}_{\mathsf{PABB}}$ with input $(\mathtt{Multiply},id_x,id_y,id_z,sid)$.

Output To One To output identifier id_x, the simulator does the following:

1. Send the command $(\mathtt{Output},i,id_x,sid)$ to $\mathcal{F}_{\mathsf{PABB}}$ and then do the following:
   If $i \in [n]\setminus A$, then await shares $\{x^{\mathtt{R}}_{\mathcal{G}} : \mathcal{G} \in \bigcup_{i\in A}\nabla^{(i)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}$ from $\mathcal{A}$.
   If $i \in A$, then do the following:
   - Retrieve from memory the corresponding shares $[\![x]\!]^{\mathtt{R}}$.

Simulator $\mathcal{S}_{\mathsf{PABB}}$ (continued)

- Await the output x from $\mathcal{F}_{\mathsf{PABB}}$.
- Sample $x^{\mathsf{R}}_{\mathcal{G}'} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{\mathcal{G}\in\nabla} x^{\mathsf{R}}_{\mathcal{G}} = x$ where $\mathcal{G}' \in \nabla$ is any set satisfying $\mathcal{G}' \cap A = \varnothing$.
- Send $\{x^{\mathsf{R}}_{G} : \mathcal{G} \in \bigcup_{i\in[n]\setminus A} \nabla^{(i)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}$ to $\mathcal{A}$.

2. (No simulation is required for this step.)

Output To All To output a share with identifier id_x, the simulator does the following:

1. Send the command $(\texttt{Output}, 0, id_x, sid)$ to $\mathcal{F}_{\mathsf{PABB}}$ and then do the following:
   - Await the output x from $\mathcal{F}_{\mathsf{PABB}}$.
   - Retrieve from memory the corresponding shares $[\![x]\!]^{\mathsf{R}}$.
   - Sample $[\![x]\!]^{\mathsf{R}}_{G'} \leftarrow \mathcal{U}(\mathbb{F})$ subject to $\sum_{G\in\nabla} x^{\mathsf{R}}_{\mathcal{G}} = x$ where $\mathcal{G}' \in \nabla$ is any set satisfying $\mathcal{G}' \cap A = \varnothing$.
   - For each $i \in A$, send $\{x^{\mathsf{R}}_{\mathcal{G}} : \mathcal{G} \in \bigcup_{j\in[n]\setminus A} \nabla^{(j)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}$ to $\mathcal{A}$.
   - For each $i \in [n] \setminus A$, await shares $\{x^{\mathsf{R}}_{\mathcal{G}} : \mathcal{G} \in \bigcup_{j\in A} \nabla^{(j)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}$ from $\mathcal{A}$.
2. (No simulation is required for this step.)

SAToR The macro $\texttt{SAToR}(id_v)$ for converting an additive sharing $[\![v]\!]^{\mathsf{A}}$ to a replicated sharing $[\![v]\!]^{\mathsf{R}}$ is defined as follows:

1. Compute a new identifier id_z, await the call to $\mathcal{F}_{\mathsf{RZS}}$ with input $(\texttt{ZeroSharing}, id_z, \mathcal{P}, sid)$ from $\mathcal{A}$, then await the shares $\{[\![t]\!]^{\mathsf{A}}_{\mathcal{P}_i}\}_{i\in A}$ from $\mathcal{A}$, compute shares for (emulated) honest parties internally, and store all shares.
2. Compute a new identifier id_s, await the call to $\mathcal{F}^{\mathsf{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_s, sid)$ from $\mathcal{A}$, and execute the local instance honestly, returning appropriate outputs to $\mathcal{A}$ and storing the output shares, $[\![r]\!]^{\mathsf{R}}$.
3. For all $i \in [n] \setminus A$,
   a) Set $v^{\mathsf{R}}_{\mathcal{G}} := r^{\mathsf{R}}_{\mathcal{G}}$ for all $\mathcal{G} \in \nabla \setminus \{\nabla^{(j)}\}_{j\in[n]}$ such that $\mathcal{G} \ni \mathcal{P}_i$.
   b) Set
   $$v^{\mathsf{R}}_{\mathcal{G}_i} := [\![v]\!]^{\mathsf{A}}_{\mathcal{P}_i} + [\![t]\!]^{\mathsf{A}}_{\mathcal{P}_i} - \sum_{G\in f^{-1}(\{i\})\setminus\{\mathcal{G}_i\}} r^{\mathsf{R}}_{\mathcal{G}}.$$
4. For every $i \in [n] \setminus \mathcal{A}$, for every $j \in A$, if $\mathcal{P}_j \in \mathcal{G}_i$ then send the share $v^{\mathsf{R}}_{\mathcal{G}_i}$ to $\mathcal{A}$.
   For every $i \in A$, for every $j \in [n] \setminus A$, if $\mathcal{P}_j \in \mathcal{G}_i$, then await the share $v^{\mathsf{R}}_{\mathcal{G}_i}$ from $\mathcal{A}$.
5. On behalf of each (emulated) honest party $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{A}$, concatenate $\{v^{\mathsf{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \wedge \mathcal{G} \ni \mathcal{P}_i\}$ into a share vector $[\![v]\!]^{\mathsf{R}}_{\mathcal{P}_i}$.
6. (Locally) return $v := \sum_{\mathcal{G}\in\nabla} v^{\mathsf{R}}_{\mathcal{G}}$.

Figure 6.12: Simulator $\mathcal{S}_{\mathsf{PABB}}$ for $\mathcal{F}_{\mathsf{PABB}}$.

Note that the simulator's local copies of shares are consistent throughout in the sense that any linear operations are emulated locally by honest parties. This means that, for example, if $\mathcal{Z}$ provides as input x and y and then requests the outputs x, y and $a \cdot x + b \cdot y$ then the *shares* are consistent as well as the secrets – i.e. for all $\mathcal{G} \in \nabla$,

$$a \cdot x^{\mathsf{R}}_{\mathcal{G}} + b \cdot y^{\mathsf{R}}_{\mathcal{G}} = (a \cdot x + b \cdot y)^{\mathsf{R}}_{\mathcal{G}}.$$

Procedure	From	To	Message
Initialize	$\mathcal{A}$	$\mathcal{F}_{\mathsf{RSS}}^{\mathbb{R}}$	$(\texttt{Initialize}, \Gamma, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{RZS}}$	$(\texttt{Initialize}, \mathcal{P}, sid)$
Input	[See $\Pi_{\mathtt{AToROpt}}$]		
Add	n/a	n/a	n/a
Multiply	[See $\Pi_{\mathtt{AToROpt}}$]		
Output To One	$\mathcal{S}$	$\mathcal{A}$	$\{x_{\mathcal{G}}^{\mathbb{R}} : \mathcal{G} \in \bigcup_{i \in [n] \setminus A} \nabla^{(i)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}$ (if $\mathcal{P}_i \in \mathcal{A}$)
	$\mathcal{A}$	$\mathcal{S}$	$\{x_{\mathcal{G}}^{\mathbb{R}} : \mathcal{G} \in \bigcup_{i \in A} \nabla^{(i)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}$ (if $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{A}$)
Output To All	$\mathcal{S}$	$\mathcal{A}$	$\{x_{\mathcal{G}}^{\mathbb{R}} : \mathcal{G} \in \bigcup_{j \in [n] \setminus A} \nabla^{(j)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}_{i \in A}$
	$\mathcal{A}$	$\mathcal{S}$	$\{x_{\mathcal{G}}^{\mathbb{R}} : \mathcal{G} \in \bigcup_{j \in A} \nabla^{(j)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\}_{i \in [n] \setminus A}$
$\Pi_{\mathtt{AToROpt}}$	$\mathcal{A}$	$\mathcal{F}_{\mathsf{RZS}}$	$(\texttt{ZeroSharing}, id_z, \mathcal{P}, sid)$
	$\mathcal{F}_{\mathsf{RZS}}$	$\mathcal{A}$	$[\![t]\!]_{\mathcal{A}}^{\mathbb{A}}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{RSS}}^{\mathbb{R}}$	$(\texttt{SecretSharing}, id_s, sid)$
	$\mathcal{F}_{\mathsf{RSS}}^{\mathbb{R}}$	$\mathcal{A}$	$[\![r]\!]_{\mathcal{A}}^{\mathbb{R}}$
	$\mathcal{S}$	$\mathcal{A}$	$\{v_{\mathcal{G}_i}^{\mathbb{R}} : i \in [n] \setminus A \wedge \mathcal{G}_i \cap A \neq \varnothing\}$
	$\mathcal{A}$	$\mathcal{S}$	$\{v_{\mathcal{G}_i}^{\mathbb{R}} : i \in A \wedge \mathcal{G}_i \setminus A \neq \varnothing\}$

Figure 6.13: Transcript for $\Pi_{\mathsf{Online}}^{\mathscr{Q}_2,\mathbb{R}}$.

It is clear from the transcript that the key points for distinguishing between worlds are in the output stage and in the execution of $\Pi_{\mathtt{AToROpt}}$. Notice that a corrupt party's input can always be extracted since $\mathcal{G}_i$ always contains at least one honest party (since every set in ∇ contains at least one honest party, by the definition of $\mathscr{Q}_2$), so $\mathcal{S}$ can always extract inputs using knowledge of the PRZS mask and the PRSS shares. This means that the output in the ideal world is correct according to the inputs of (real) honest parties and the adversary. Now, in the output stage, since there is at least one set $\mathcal{G} \in \nabla$ such that $\mathcal{A} \cap \mathcal{G} = \varnothing$ (again, because the access structure is $\mathscr{Q}_2$), $\mathcal{S}$ can fix the output to be whatever it chooses – and in particular, to what it receives from $\mathcal{F}_{\mathsf{PABB}}$.

It only remains to show that the execution $\Pi_{\mathtt{AToROpt}}$ does not reveal the fact that $\mathcal{S}$ set the emulated honest parties' inputs to 0 during the simulation. This is achieved by a standard argument involving a sequence of hybrid worlds in the following way. Let $\mathcal{A} \in \Delta$ be the set of parties corrupted by the adversary, let $h := |\mathcal{P} \setminus \mathcal{A}|$, and for simplicity assume that the indexing set for $\mathcal{A}$, A, indexes the last $n-h$ parties so that $[h]$ indexes the honest parties. Define the i^{th} hybrid world as follows:

Hybrid i The $\mathcal{F}_{\mathsf{RSS}}^{\mathbb{R}}$, $\mathcal{F}_{\mathsf{RZS}}$-hybrid world in which the simulator is handed the inputs of all honest parties $\mathcal{P}_j \in \mathcal{P} \setminus \mathcal{A}$ where $j \leq i$.

The simulator for the i^{th} world is defined by replacing the instruction $[\![x]\!]^{\mathtt{A}}_{\mathcal{P}_j} \coloneqq 0$ in Step 1 of **Input** in $\mathcal{S}_{\mathsf{PABB}}$ with the instruction $[\![x_j]\!]^{\mathtt{A}}_{\mathcal{P}_j} \coloneqq x_j$, where x_j is the input of honest party $\mathcal{P}_j$, for each $j \leq i$. Thus **Hybrid 0** is exactly the ideal world, in which the simulator knows none of the honest parties' inputs, and **Hybrid h** is the $\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}, \mathcal{F}_{\mathsf{RZS}}$-hybrid world, where the simulator acts as honest parties would in a protocol execution.

Claim 6.1. The world **Hybrid i** is indistinguishable from **Hybrid i+1** for all i such that $0 \leq i < h$.

Proof. Since inputs can always be extracted and outputs can always be doctored by $\mathcal{S}$ to be the same secrets in the real (hybrid) world as in the ideal world, the only difference to the simulation between consecutive worlds is when an honest party provides their input, (which is done independently for each i), which means that distinguishing between consecutive worlds is the same task for all values of $i < h$. In detail, to distinguish between **Hybrid i-1** and **Hybrid i**, $\mathcal{Z}$ must distinguish between the distributions of shares it observes in the transcript for the input of $\mathcal{P}_i$; that is, between

$$\left\{r^{\mathtt{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \setminus \{\mathcal{G}_j\}_{j\in[n]} \wedge \mathcal{G} \cap \mathcal{A} \neq \varnothing\right\} \cup \left\{v^{\mathtt{R}}_{\mathcal{G}_j} : \mathcal{G}_j \cap \mathcal{A} \neq \varnothing \wedge j \neq i\right\} \cup \left\{[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} - \sum_{\mathcal{G}\in f^{-1}(\{\mathcal{P}_i\})\setminus\{\mathcal{G}_i\}} r^{\mathtt{R}}_{\mathcal{G}}\right\}$$

and

$$\left\{r^{\mathtt{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \setminus \{\mathcal{G}_j\}_{j\in[n]} \wedge \mathcal{G} \cap \mathcal{A} \neq \varnothing\right\} \cup \left\{v^{\mathtt{R}}_{\mathcal{G}_j} : \mathcal{G}_j \cap \mathcal{A} \neq \varnothing \wedge j \neq i\right\} \cup \left\{x + [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} - \sum_{\mathcal{G}\in f^{-1}(\{\mathcal{P}_i\})\setminus\{\mathcal{G}_i\}} r^{\mathtt{R}}_{\mathcal{G}}\right\}$$

where the first distribution is produced by a simulator that sets the input to be 0, as in **Hybrid i-1**, and the second is produced by a simulator that knows the input of the honest party $\mathcal{P}_i$, as in **Hybrid i**.

Note that these two sets of shares are missing at least one share, indexed by some set $\mathcal{G}$ such that $\mathcal{G} \cap \mathcal{A} = \varnothing$. Now either $\mathcal{G} \in \{\mathcal{G}_i\}_{i\in[n]}$, or $\mathcal{G} \in \nabla \setminus \{\mathcal{G}_i\}_{i\in[n]}$.

If $\mathcal{G} \in \{\mathcal{G}_i\}_{i\in[n]}$ then $\mathcal{G} = \mathcal{G}_j$ for some $j \in [n]$. Thus $\mathcal{Z}$ has no information on the share $[\![t]\!]^{\mathtt{A}}_{\mathcal{P}_j}$ of the PRZS which masks for $v^{\mathtt{R}}_{\mathcal{G}_j}$. This means that both distributions above are indistinguishable from uniform, since there is no way for $\mathcal{Z}$ to compute a sum and cancel out this share.

If $\mathcal{G} \in \nabla \setminus \{\mathcal{G}_i\}_{i\in[n]}$, then $\mathcal{Z}$ has no information on some share $r^{\mathtt{R}}_{\mathcal{G}}$ which masks one element of $\{v^{\mathtt{R}}_{\mathcal{G}_j} : j \in [n] \setminus A\}$. This means that again both distributions are indistinguishable from uniform. ■

Since there are only polynomially-many hybrid worlds, **Hybrid 0** is indistinguishable from **Hybrid h** by transitivity, which is exactly saying that the ideal and $\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}, \mathcal{F}_{\mathsf{RZS}}$-hybrid worlds are indistinguishable. □

6.5.4 Communication Complexity

Unlike in the IT protocol $\Pi_{\texttt{AToAny}}$, by using $\Pi_{\texttt{AToROpt}}$, it is no longer necessary for the parties to be connected in a complete network during the passive multiplication: instead, parties are connected as defined by the set

$$E_{\texttt{AToR}} = \left\{(i,j) \in [n]^2 : \mathcal{P}_j \in \mathcal{G}_i \setminus \{\mathcal{P}_i\}\right\}$$

where $(i,j) \in E_{\texttt{AToR}}$ implies that $\mathcal{P}_i$ is connected to $\mathcal{P}_j$ by a unidirectional channel. These channels must be secure, and so they are denoted by $\mathsf{SC}(E_{\texttt{AToR}})$.

For a single party $\mathcal{P}_i$ to receive output, it must receive every share it does not hold from a party that does hold the share, over a secure channel. The required set of connections is defined by

$$E^i_{\texttt{Open}} := \left\{(j,i) \in [n] \times \{i\} : \mathcal{G} \in \nabla^{(j)} \wedge \mathcal{G} \not\ni \mathcal{P}_i\right\}.$$

For all parties to receive an output, it is the same as one party receiving output but over authenticated channels. The set of connections is defined as

$$E_{\texttt{Open}} := \bigcup_{i \in [n]} E^i_{\texttt{Open}}.$$

The costs are summarized in Table 6.2, where the cost for initialization is the cost of instantiating $\mathcal{F}^{\mathsf{R}}_{\mathsf{RSS}}$ and $\mathcal{F}_{\mathsf{RZS}}$ as given in Table 6.1 in Section 6.3.

Procedure	Number of bits	Channels
Initialize	[See Table 6.1]	
Input	$\sum_{i\in[n]}(\lvert\mathcal{G}_i\rvert - 1)\cdot\ell\cdot M$	$\mathsf{SC}(E_{\texttt{AToR}})$
Add	n/a	n/a
Multiply	$\sum_{i\in[n]}(\lvert\mathcal{G}_i\rvert - 1)\cdot\ell\cdot T$	$\mathsf{SC}(E_{\texttt{AToR}})$
Output To One	$\left\lvert\{\mathcal{G}\in\nabla : \mathcal{G}\not\ni\mathcal{P}_i\}\right\rvert\cdot\ell$	$\mathsf{SC}(E^i_{\texttt{Open}})$
Output To All	$\sum_{i\in[n]}\left\lvert\{\mathcal{G}\in\nabla : \mathcal{G}\not\ni\mathcal{P}_i\}\right\rvert\cdot\ell$	$\mathsf{AC}(E_{\texttt{Open}})$

Table 6.2: Total communication cost to realize $\mathcal{F}_{\mathsf{PABB}}$ with M inputs and T total multiplications.

For circuits with high multiplicative depth, the communication cost is dominated by the cost of $\Pi_{\texttt{AToROpt}}$, in which for each conversion, party $\mathcal{P}_i$ sends $|\mathcal{G}_i| - 1$ finite-field elements, so the total is $\sum_{i\in[n]}(|\mathcal{G}_i| - 1)$. For a threshold scheme, this total cost is $n\cdot(n - t - 1) = O(n^2)$ field elements – a *linear* cost per party per multiplication. This sharply contrasts the IT case in which the cost is $O(n^{1.5}\cdot 2^n)$ – an *exponential* cost per party.

For Example 6.1, the set of channels required for a passive multiplication is given by

$$E_{\mathtt{AToR}} \coloneqq \Big\{(1,3),(1,4),(2,1),(2,4),(3,1),(3,2),(4,2),(4,3),(4,5),\\ (5,1),(5,2),(5,6),(6,2),(6,3),(6,4)\Big\}.$$

During the procedure, one field element is sent over each of these channels, so the parties send 15 finite-field elements over 15 uni-directional secure channels for a passive multiplication, which can also be seen by inspecting Figure 6.8. This equates to a bandwidth saving, compared to the initial protocol of Maurer, of 93% in the number of transmitted finite-field elements, and a saving of 50% in the number of secure channels.

6.6 Actively-Secure $\mathscr{Q}_2$ MPC Protocol

In this section, an actively-secure MPC protocol in the preprocessing model is given, making use of the passively-secure protocol of the previous section and the cheap authentication from Chapter 3. The overall protocol is then a relatively conventional approach to obtaining active security:

1. The preprocessing phase, realizing $\mathcal{F}_{\mathsf{Prep}}$ (Figure 4.7):
   - Generate and multiply random secrets to obtain Beaver triples with passive security. This requires the passive protocol $\Pi^{\mathscr{Q}_2,\mathtt{R}}_{\mathsf{Online}}$ in Figure 6.11.
   - Check triples by *sacrificing*. This requires communication over a set of reduced set of *authenticated* channels and makes use of the protocol Π_{Open} from Chapter 3.
2. The online phase, realizing $\mathcal{F}_{\mathsf{ABB}}$ (Figure 2.14):
   - To evaluate a circuit, parties execute the protocol Π_{Online} from Chapter 4. Additions require no communication and multiplications require only the same subnetwork of authenticated channels as for sacrificing instead of a complete network.

Authentication for opened secrets was discussed in detail in Chapter 3 and is dealt with by the functionality $\mathcal{F}_{\mathsf{Open}}$ in Figure 4.1. An important aspect of the authentication procedure for a $\mathscr{Q}_2$ access structure is that secrets can be checked in batches;

thus, comparably to [FLNW17] and analogously to the subprotocol Π_{MACCheck} (given in Figure 4.14) used in full-threshold protocols such as SPDZ [DPSZ12], it is possible to batch-check the correctness of Beaver triples, which means that, asymptotically, the cost of generating a Beaver triple with active security is the same as the cost of performing two passive multiplications and two openings of secrets. This is discussed in further detail in Section 6.6.3.

Note that this standard bootstrapping to active security is unlike the method in [FLNW17] where the online multiplication protocol involves executing the passively-secure multiplication protocol and checking correctness using a Beaver triple. The traditional method allows the online protocol to be executed over authenticated, as opposed to secure, channels. The preprocessing protocol $\Pi^{\mathcal{Q}_2,\mathrm{R}}_{\mathsf{Prep}}$ is given in Figure 6.14.

Protocol $\Pi^{\mathcal{Q}_2,\mathrm{R}}_{\mathsf{Prep}}$

This protocol is realized in the $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{Open}}$, $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$, $\mathcal{F}_{\mathsf{RZS}}$-hybrid model.

Initialize The parties do the following:

1. Agree on a session identifer sid.
2. Set $R := \lceil \sigma / \log|\mathbb{F}| \rceil$.
3. Call an instance of $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{\mathrm{R}}, sid)$.
4. Call an instance of $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{Initialize}, \Gamma, sid)$.
5. Call an instance of $\mathcal{F}_{\mathsf{RZS}}$ with input $(\texttt{Initialize}, \mathcal{P}, sid)$.
6. Set `Abort` to false.
7. Agree on a sharing of 1, $[\![1]\!]^{\mathrm{R}}$.[a]

Mask In order for $\mathcal{P}_i$ to obtain a mask, the parties do the following:

1. Compute a new identifier id_r and then call $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_r, sid)$ to obtain a sharing $[\![r]\!]^{\mathrm{R}}$.
2. Call $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Open}, i, id_r, sid)$ to open r to $\mathcal{P}_i$. If the $\mathcal{F}_{\mathsf{Open}}$ returns the message Abort then (locally) output $\perp$ and halt; otherwise party $\mathcal{P}_i$ (locally) outputs $(r, [\![r]\!]^{\mathrm{R}}_{\mathcal{P}_i})$ and for all $j \in [n] \setminus \{i\}$ party $\mathcal{P}_j$ (locally) outputs $[\![r]\!]^{\mathrm{R}}_{\mathcal{P}_j}$.

Triples To generate T triples, the parties do the following:

1. **Generate** For $k = 1, \ldots, (R+1) \cdot T$, do the following:
   a) Compute new identifiers id_{a_k} and id_{b_k} and then call $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_{a_k}, sid)$ and $(\texttt{SecretSharing}, id_{b_k}, sid)$ to obtain $[\![a_k]\!]^{\mathrm{R}}$ and $[\![b_k]\!]^{\mathrm{R}}$.
   b) Execute **Multiply** of $\Pi^{\mathcal{Q}_2,\mathrm{R}}_{\mathsf{Online}}$ to obtain $[\![c_k]\!]^{\mathrm{R}}$.
2. **Sacrifice**
   a) Call a new instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \mathbb{F}^{R \cdot T}, sid)$.

Protocol $\Pi^{\mathscr{Q}_2,\mathrm{R}}_{\mathsf{Prep}}$ (continued)

b) Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\mathtt{RElt}, sid)$ to obtain $\{\{\rho_{k+\ell\cdot T}\}_{k=1}^{T}\}_{\ell=1}^{R}$.

c) For $k = 1,\ldots,T$, do the following:[b]

i. For $\ell = 1,\ldots,R$, do the following:

A. Set $[\![r_{k+\ell\cdot T}]\!]^{\mathrm{R}} := [\![a_k]\!]^{\mathrm{R}} - [\![a_{k+\ell\cdot T}]\!]^{\mathrm{R}}$ and $[\![s_{k+\ell\cdot T}]\!]^{\mathrm{R}} := [\![b_k]\!]^{\mathrm{R}} - \rho_{k+\ell\cdot T}\cdot[\![b_{k+\ell\cdot T}]\!]^{\mathrm{R}}$.

B. Call $\mathcal{F}_{\mathsf{Open}}$ with inputs $(\mathtt{Open}, 0, id_{r_{k+\ell\cdot T}}, sid)$ and $(\mathtt{Open}, 0, id_{s_{k+\ell\cdot T}}, sid)$.

C. If the flag `Abort` has not been set to true, the parties locally compute value

$$[\![t_{k+\ell\cdot T}]\!]^{\mathrm{R}} := r_{k+\ell\cdot T}\cdot s_{k+\ell\cdot T}\cdot[\![1]\!]^{\mathrm{R}} + s_{k+\ell\cdot T}\cdot[\![a_{k+\ell\cdot T}]\!]^{\mathrm{R}} + \rho_{k+\ell\cdot T}\cdot r_{k+\ell\cdot T}\cdot[\![b_{k+\ell\cdot T}]\!]^{\mathrm{R}} + \rho_{k+\ell\cdot T}\cdot[\![c_{k+\ell\cdot T}]\!]^{\mathrm{R}} - [\![c_k]\!]^{\mathrm{R}}.$$

d) Call $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Verify}, sid)$, and if the parties have not aborted then call $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Open}, 0, id_{t_{k+\ell\cdot T}}, sid)$ for every $k \in [T]$ and for every $\ell \in [R]$. If any returned value is not zero, the parties set the flag `Abort` to true.

3. **Check** If the flag `Abort` is not set to true then the parties (locally) output $([\![a_k]\!]^{\mathrm{R}}, [\![b_k]\!]^{\mathrm{R}}, [\![c_k]\!]^{\mathrm{R}})_{k=1}^{T}$ and discard all other secret-sharings, and otherwise they call $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Broadcast}, \mathsf{Abort}, sid)$, (locally) output $\perp$, and halt.

[a] For example, let i be the first non-zero component of $\mathbf{t}$; then set the randomness vector $\mathbf{x}$ to be the vector $\mathbf{0}$ except with the i^{th} component equal to $1/\mathbf{t}_i$.

[b] All openings on different sets of triples are independent and can be batched.

Figure 6.14: Preprocessing Protocol for a $\mathscr{Q}_2$ Access Structure using Replicated Secret-Sharing, $\Pi^{\mathscr{Q}_2,\mathrm{R}}_{\mathsf{Prep}}$.

Remark 6.1. Often in actively-secure MPC protocols, input and output masks are generated in the preprocessing phase and are then used in the online phase for parties to provide inputs and obtain private outputs. (See, for example, [CDI05, DPSZ12].) The input procedure was summarized in Section 2.5. For private outputs, the parties use a secret-shared mask r known to $\mathcal{P}_i$ as follows: to reveal secret-shared x to $\mathcal{P}_i$, the parties compute and $[\![x]\!] - [\![r]\!]$ and open this secret using authenticated channels; then $\mathcal{P}_i$ can compute $x = (x - r) + r$.

However, if the access structure is $\mathscr{Q}_2$, then a party to whom a secret is revealed can detect whether or not the secret is correct, as described in Section 3.2. Rather than use error-detection, the actively-secure protocols in this chapter follow the method involving masks as it allows the online phase to be executed entirely over authenticated channels, rather than requiring secure channels.

6.6.1 Correctness

Correctness of $\Pi_{\mathsf{Prep}}^{\mathscr{D}_2,\mathrm{R}}$ follows from the correctness of the passively-secure multiplication protocol: the other parts of the protocol only involve opening secrets to all parties or to one party.

6.6.2 Security

It is necessary to show that if the adversary cheats then the honest parties detect it, except with negligible probability in the statistical security parameter.

Lemma 6.1. *For a fixed $k \in [T]$, if $t_{k+\ell\cdot T} = 0$ for every $\ell \in [R]$, then it holds that $a_k \cdot b_k = c_k$ except with probability at most $2^{-\sigma}$.*

Proof. Suppose $t_{k+\ell\cdot T} = 0$ for all $\ell \in [R]$ but $c_k = a_k \cdot b_k + \varepsilon_k$ for some $\varepsilon_k \neq 0$. The adversary must introduce errors $\{\varepsilon_{k+\ell\cdot T}\}_{\ell=1}^{R}$ for triples $\{(a_{k+\ell\cdot T}, b_{k+\ell\cdot T}, c_{k+\ell\cdot T})\}_{\ell=1}^{R}$ when executing the passive multiplication subprotocol so that $c_{k+\ell\cdot T} = a_{k+\ell\cdot T} \cdot b_{k+\ell\cdot T} + \varepsilon_{k+\ell\cdot T}$ but that the sacrifice equation still holds.

Since $\rho_{k+\ell\cdot T}$ is an output of $\mathcal{F}_{\mathsf{CoinFlip}}$ and not known when the triples are generated, the only way that $t_{k+\ell\cdot T} = 0$ but $a_k \cdot b_k = c_k + \varepsilon_k$ where $\varepsilon_k \neq 0$ is for $\mathcal{A}$ to guess each $\rho_{k+\ell\cdot T}$ so that

$$\rho_{k+\ell\cdot T} \cdot \varepsilon_{k+\ell\cdot T} - \varepsilon_k = 0$$

for all $\ell \in [R]$. This is equivalent to guessing $\{\rho_{k+\ell\cdot T}\}_{\ell=1}^{R}$, which can be done correctly with probability at most $|\mathbb{F}|^{-R} \leq 2^{-\sigma}$. □

Theorem 6.4. *The protocol Π_{Prep} UC-securely realizes $\mathcal{F}_{\mathsf{Prep}}$ against a static, active adversary in the $\mathcal{F}_{\mathsf{Open}}$, $\mathcal{F}_{\mathsf{CoinFlip}}$, $\mathcal{F}_{\mathsf{RSS}}^{\mathrm{R}}$, $\mathcal{F}_{\mathsf{RZS}}$-hybrid model with statistical security σ.*

Proof. The simulator is given in Figure 6.15 and the transcript in Figure 6.16.

Simulator $\mathcal{S}_{\mathsf{Prep}}$

Initialize The simulator does the following:

1. Agree on a session identifier sid with $\mathcal{A}$.
2. Set $R := \lceil \sigma / \log|\mathbb{F}| \rceil$.
3. Await the call to $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{\mathrm{R}}, sid)$ from $\mathcal{A}$ and then call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{\mathrm{R}}, sid)$ on behalf of each corrupt party.
4. Await the call to $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{Initialize}, \Gamma, sid)$ from $\mathcal{A}$ and initialize a local instance.
5. Await the call to $\mathcal{F}_{\mathsf{RZS}}$ with input $(\texttt{Initialize}, \mathcal{P}, sid)$ from $\mathcal{A}$ and initialize a local instance.
6. Set `Abort` to false.
7. Agree on a sharing of $[\![1]\!]^{\mathrm{R}}$ with $\mathcal{A}$.

Mask To create a mask for $\mathcal{P}_i$, the simulator does the following:

1. Compute a new identifier id_r, await the call to $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_r, sid)$ from $\mathcal{A}$, call the internal copy of $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_r, sid)$ to obtain $[\![r]\!]^{\mathrm{R}}$, and send the shares $[\![r]\!]^{\mathrm{R}}_{\mathcal{A}}$ to $\mathcal{A}$.
2. Await the call to $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Open}, id_r, i, sid)$ from $\mathcal{A}$, and then do the following:
   - Call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Mask}, i, id_r, sid)$ and if $i \in A$ also send r.
   - When $\mathcal{F}_{\mathsf{Prep}}$ executes $\texttt{Sample}(id_r)$, send the shares $[\![r]\!]^{\mathrm{R}}_{\mathcal{A}}$ to $\mathcal{F}_{\mathsf{Prep}}$. Then,
     If $i \in A$,
     - Send $[\![r]\!]^{\mathrm{R}}_{\mathcal{P}\setminus\mathcal{A}}$ to $\mathcal{A}$.
     - If $\mathcal{A}$ calls $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid)$ then send the message Abort to $\mathcal{F}_{\mathsf{Prep}}$ and halt; otherwise send the message OK.

     If $i \in [n] \setminus A$,
     - Await $[\![r]\!]^{\mathrm{R}}_{\mathcal{A}}$ from $\mathcal{A}$.
     - If $N \cdot [\![r]\!]^{\mathrm{R}} \neq \mathbf{0}$ then send the message Abort to $\mathcal{F}_{\mathsf{Prep}}$ and call the local instance of $\mathcal{F}_{\mathsf{Open}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid)$ and execute it honestly with $\mathcal{A}$; otherwise, send the message OK.

Triples To generate T triples, the simulator does the following:

1. **Generate** For $k = 1, \ldots, (R+1) \cdot T$, compute identifiers id_{a_k}, id_{b_k} and id_{c_k}, call $\mathcal{F}_{\mathsf{Prep}}$ with input $(\texttt{Triple}, (id_{a_k}, id_{b_k}, id_{c_k})_{k=1}^{T}, sid)$, and then do the following:
   a) Call the internal copy of $\mathcal{F}^{\mathrm{R}}_{\mathsf{RSS}}$ with input $(\texttt{SecretSharing}, id_{a_k}, sid)$ and $(\texttt{SecretSharing}, id_{b_k}, sid)$ to obtain $[\![a_k]\!]^{\mathrm{R}}$ and $[\![b_k]\!]^{\mathrm{R}}$, and when $\mathcal{F}_{\mathsf{Prep}}$ executes $\texttt{Sample}(id_{a_k})$ and $\texttt{Sample}(id_{b_k})$, send $[\![a_k]\!]^{\mathrm{R}}_{\mathcal{A}}$ and $[\![b_k]\!]^{\mathrm{R}}_{\mathcal{A}}$ to both $\mathcal{A}$ and $\mathcal{F}_{\mathsf{Prep}}$.
   b) Execute **Multiply** of $\Pi^{\mathscr{Q}_2,\mathrm{R}}_{\mathsf{Online}}$ honestly with $\mathcal{A}$, running the simulation of $\Pi_{\mathtt{AToROpt}}$ by executing $\texttt{SAToR}(id_v)$ from $\mathcal{S}_{\mathsf{PABB}}$, to obtain a share vector $[\![\widetilde{c}_k]\!]^{\mathrm{R}}$, and when $\mathcal{F}_{\mathsf{Prep}}$ executes $\texttt{Sample}(id_{c_k})$, send $[\![\widetilde{c}_k]\!]^{\mathrm{R}}_{\mathcal{A}}$.
2. **Sacrifice**
   a) Await the call to $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \mathbb{F}^{R \cdot T}, sid)$ and initialize a (new) local instance.

Simulator $\mathcal{S}_{\mathsf{Prep}}$ (continued)

b) Await the call to $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\mathtt{RElt}, \mathbb{F}^{R\cdot T}, sid)$ from $\mathcal{A}$, execute it honestly to obtain $\{\rho_{k+\ell\cdot T}\}_{k=1}^{R\cdot T}$, and send this set to $\mathcal{A}$.

c) For $k = 1,\ldots,T$, do the following:

i. For $\ell = 1,\ldots,R$, do the following:

A. Set $[\![r_{k+\ell\cdot T}]\!]^{\mathrm{R}} := [\![a_k]\!]^{\mathrm{R}} - [\![a_{k+\ell\cdot T}]\!]^{\mathrm{R}}$ and $[\![s_{k+\ell\cdot T}]\!]^{\mathrm{R}} := [\![b_k]\!]^{\mathrm{R}} - \rho_{k+\ell\cdot T}\cdot[\![b_{k+\ell\cdot T}]\!]^{\mathrm{R}}$.

B. Await the call to $\mathcal{F}_{\mathsf{Open}}$ with inputs $(\mathtt{Open}, id_{r_{k+\ell\cdot T}}, sid)$ and $(\mathtt{Open}, id_{s_{k+\ell\cdot T}}, sid)$ from $\mathcal{A}$ and execute honestly with $\mathcal{A}$ using the local instance of $\mathcal{F}_{\mathsf{Open}}$.

C. Compute the honest parties' shares of

$$[\![t_{k+\ell\cdot T}]\!]^{\mathrm{R}} := r_{k+\ell\cdot T}\cdot s_{k+\ell\cdot T}\cdot[\![1]\!]^{\mathrm{R}} + s_{k+\ell\cdot T}\cdot[\![a_{k+T}]\!]^{\mathrm{R}} + \rho_{k+\ell\cdot T}\cdot r_{k+\ell\cdot T}\cdot[\![b_{k+\ell\cdot T}]\!]^{\mathrm{R}} + \rho_{k+\ell\cdot T}\cdot[\![\widetilde{c}_{k+\ell\cdot T}]\!]^{\mathrm{R}} - [\![\widetilde{c}_k]\!]^{\mathrm{R}}.$$

d) Await the call to $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Verify}, sid)$ from $\mathcal{A}$ and execute the procedure honestly with $\mathcal{A}$. If the (emulated) honest parties did not abort, then await the calls to $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Open}, id_{t_{k+\ell\cdot T}}, sid)$ for every $k \in [T]$ and for every $\ell \in [R]$, and execute the procedures honestly. If any (emulated) honest party would have set their flag `Abort` to true then $\mathcal{S}$ sets its own `Abort` to true.

3. **Check** If $\mathcal{A}$ calls $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Broadcast}, \mathsf{Abort}, sid)$, or if the flag `Abort` is set to true, then call the internal copy of $\mathcal{F}_{\mathsf{Open}}$ with input $(\mathtt{Broadcast}, \mathsf{Abort}, sid)$, execute it honestly with $\mathcal{A}$, and then send the message Abort to $\mathcal{F}_{\mathsf{Prep}}$; otherwise, send the message OK to $\mathcal{F}_{\mathsf{Prep}}$.

Figure 6.15: Simulator $\mathcal{S}_{\mathsf{Prep}}$ for $\mathcal{F}_{\mathsf{Prep}}$.

It is clear from the transcript that the parties do not have inputs and the only outputs are triples and masks. This means that there are only two ways for an environment, $\mathcal{Z}$, to attempt to distinguish between hybrid-world and ideal-world executions: 1) Examining shares generated by corrupt parties to see if they are "carried through" by the simulator into the ideal world; or 2) Making the adversary generate invalid share vectors, masks, or triples, so that the hybrid world would abort but the simulator does not detect this. It is important that the probability with which honest parties can detect cheating behaviour be overwhelming in σ in order for the protocol to be (statistically) correct.

For the first, recall that since replicated secret-sharing is used, $\mathcal{S}$ receives all shares held by $\mathcal{A}$ during the execution of **Multiply**, which means that $\mathcal{S}$ can pass these on to $\mathcal{F}_{\mathsf{Prep}}$. This means that the shares passed on to (real) honest parties by $\mathcal{F}_{\mathsf{Prep}}$ are consistent with the adversarially-generated shares during the execution of **Multiply**. It is also necessary to show that nothing is revealed to $\mathcal{Z}$ by the fact that $\mathcal{S}$ does not

Procedure	From	To	Message
Initialize	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{\mathtt{R}}, sid)$
	$\mathcal{A}$	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$(\texttt{Initialize}, \Gamma, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{RZS}}$	$(\texttt{Initialize}, \mathcal{P}, sid)$
Mask	$\mathcal{A}$	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$(\texttt{SecretSharing}, id_r, sid)$
	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$\mathcal{A}$	$[\![r]\!]^{\mathtt{R}}_{\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$(\texttt{Open}, id_r, i, sid)$
	$\mathcal{F}_{\mathsf{Open}}$	$\mathcal{A}$	$[\![r]\!]^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}}$
Triples			
Generate	$\mathcal{A}$	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$(\texttt{SecretSharing}, id_{a_k}, sid)$
	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$\mathcal{A}$	$[\![a_k]\!]^{\mathtt{R}}_{\mathcal{A}}$
	$\mathcal{A}$	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$(\texttt{SecretSharing}, id_{b_k}, sid)$
	$\mathcal{F}^{\mathtt{R}}_{\mathsf{RSS}}$	$\mathcal{A}$	$[\![b_k]\!]^{\mathtt{R}}_{\mathcal{A}}$
	[See $\Pi_{\mathtt{AToROpt}}$, Figure 6.13]		
Sacrifice	$\mathcal{A}$	$\mathcal{F}_{\mathsf{CoinFlip}}$	$(\texttt{Initialize}, \mathbb{F}^{R\cdot T}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{CoinFlip}}$	$(\texttt{RElt}, \mathbb{F}^{R\cdot T}, sid)$
	$\mathcal{F}_{\mathsf{CoinFlip}}$	$\mathcal{A}$	$\{\rho_{k+\ell\cdot T}\}_{k=1}^{R\cdot T}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$((\texttt{Open}, id_{r_{k+\ell\cdot T}}, sid)_{k=1}^{T})_{\ell=1}^{R}$
	$\mathcal{F}_{\mathsf{Open}}$	$\mathcal{A}$	$(([\![r_{k+\ell\cdot T}]\!]^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}})_{k=1}^{T})_{\ell=1}^{R}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$((\texttt{Open}, 0, id_{s_{k+\ell\cdot T}}, sid)_{k=1}^{T})_{\ell=1}^{R}$
	$\mathcal{F}_{\mathsf{Open}}$	$\mathcal{A}$	$(([\![s_{k+\ell\cdot T}]\!]^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}})_{k=1}^{T})_{\ell=1}^{R}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{CoinFlip}}$	$(\texttt{RElt}, \mathbb{F}^{(R-1)\cdot T}, sid)$
	$\mathcal{F}_{\mathsf{CoinFlip}}$	$\mathcal{A}$	$(((\sigma_{k+\ell\cdot T,\ell'})_{k=1}^{T})_{\ell=1}^{R})_{\ell'=1}^{R}$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$(\texttt{Verify}, sid)$
	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$(\texttt{Open}, 0, id_{u_{\ell'}}, sid)_{\ell=1}^{R}$
	$\mathcal{F}_{\mathsf{Open}}$	$\mathcal{A}$	$([\![u_{\ell'}]\!]^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}})_{\ell'=1}^{R}$
Check	$\mathcal{A}$	$\mathcal{F}_{\mathsf{Open}}$	$(\texttt{Verify}, sid)$

Figure 6.16: Transcript for $\Pi^{\mathscr{Q}_2,\mathtt{R}}_{\mathsf{Prep}}$.

know the values of a_k and b_k and hence cannot compute "correct" sharings $[\![c_k]\!]^{\mathtt{R}}_{\mathcal{G}_i}$ for $i \in [n] \setminus A$. However, it was shown in Claim 6.1 that the environment never has enough information to learn the value of the secret in the additive sharing being converted.

For the second, it is necessary to show that if the adversary introduces errors, then $\mathcal{S}$ signals $\mathcal{F}_{\mathsf{Prep}}$ to abort in the ideal world with overwhelming probability in σ. There are only two places to cheat: the corrupt parties can create an invalid share vector during the execution of $\Pi_{\mathtt{AToROpt}}$, or the share vectors can be valid but it does not hold that $a_k \cdot b_k = c_k$ for one or more $k \in [T]$. In the first case, the simulator can always detect the error and tell $\mathcal{F}_{\mathsf{Prep}}$ to abort since it emulates $\mathcal{F}_{\mathsf{Open}}$ honestly as a local instance. In the second case, Lemma 6.1 shows that honest parties will abort, except with negligible probability in σ if $\mathcal{A}$ cheated during the execution of **Multiply**. □

6.6.3 Communication Complexity

For this section it will be assumed that a partition $\{\nabla^{(i)}\}_{i\in[n]}$ where $\nabla^{(i)} \neq \varnothing$ for all $i \in [n]$ can be found. For redundant access structures, the redundant parties being removed from the computation phase only interact with the remaining parties in the input and output phases. Further discussion is given in Section 6.7.

The total communication cost is summarized in Table 6.3, and an explanation is given below, which makes use of the definitions of $E_{\texttt{AToR}}$, $E^i_{\texttt{Open}}$, and $E_{\texttt{Open}}$ given in Section 6.5.4. The field size is denoted by $\ell := \lceil \log |\mathbb{F}| \rceil$.

Preprocessing phase The costs associated with **Initialize** are given in Table 6.1. For **Mask**, the cost is given for $\mathcal{P}_i$: to receive a mask, every party must send all of their shares to $\mathcal{P}_i$ as outlined in Section 3.2, and so the set of channels required is

$$E^i_{\texttt{Mask}} := ([n] \setminus \{i\}) \times \{i\}.$$

For **Triples**, the passive multiplication protocol is used for generation requiring the connections $E_{\texttt{AToR}}$, and sacrificing triples requires $E_{\texttt{Open}}$ from Section 6.5.4. For checking triples, the complete graph of channels is needed, which is defined as

$$E_{\texttt{BC}} := \left\{(i,j) \in [n]^2 : i \neq j\right\}.$$

Remark 6.2. While the channels $E_{\texttt{Open}}$ suffice for opening secrets to all parties when using replicated secret-sharing, for other LSSSs the channels required depends on the map q as described in Section 3.3, which in general is a larger number of channels since the authenticated opening procedure is more complex than simply all parties receiving all shares they do not hold.

Online phase For **Input**, specifically for party $\mathcal{P}_i$ to provide input, it broadcasts a field element, requiring $\mathsf{AC}(E^i_{\texttt{BC}})$ where

$$E^i_{\texttt{BC}} := \{i\} \times ([n] \setminus \{i\}).$$

The procedure **Multiply**, using Beaver triples, involves opening secrets to all parties, so the same network as for triple sacrifice is required, $\mathsf{AC}(E_{\texttt{Open}})$. In **Output To One**, providing output to party $\mathcal{P}_i$ requires the channels $\mathsf{AC}(E_{\texttt{Open}})$ since the parties reveal $x - r$ and party $\mathcal{P}_i$ computes $(x-r)+r$ where r is a mask, and **Output To All** can also be performed over $\mathsf{AC}(E_{\texttt{Open}})$ since the verification step will authenticate that this was

Phase	Procedure	Number of bits	Channels
Preprocessing	**Initialize**	[See Table 6.1]	
	Mask	$\left(\left(\sum_{\mathcal{G}\in\nabla}\left\lvert\mathcal{G}\right\rvert\right)-\left\lvert\{\mathcal{G}\in\nabla:\mathcal{G}\not\ni\mathcal{P}_i\}\right\rvert\right)\cdot\ell$	$\mathsf{SC}(E^i_{\mathtt{Mask}})$
	Triples		
	Generation	$(1+\lceil\sigma/\ell\rceil)\cdot\sum_{i\in[n]}(\left\lvert\mathcal{G}_i\right\rvert-1)\cdot\ell\cdot T$	$\mathsf{SC}(E_{\mathtt{AToR}})$
	Sacrifice (Open)	$3\cdot\lceil\sigma/\ell\rceil\cdot\sum_{\mathcal{G}\in\nabla}\left\lvert\mathcal{P}\setminus\mathcal{G}\right\rvert\cdot\ell\cdot T$	$\mathsf{AC}(E_{\mathtt{Open}})$
	Sacrifice (Check)	$2\cdot n\cdot(n-1)\cdot\kappa$	$\mathsf{AC}(E_{\mathtt{BC}})$
	Check	n/a	n/a
Online	**Input**	$(n-1)\cdot\ell\cdot M$	$\mathsf{AC}(E^i_{\mathtt{BC}})$
	Add	n/a	n/a
	Multiply	$2\cdot\sum_{\mathcal{G}\in\nabla}\left\lvert\mathcal{P}\setminus\mathcal{G}\right\rvert\cdot\ell\cdot T$	$\mathsf{AC}(E_{\mathtt{Open}})$
	Output To All	$\sum_{\mathcal{G}\in\nabla}\left\lvert\mathcal{P}\setminus\mathcal{G}\right\rvert\cdot\ell$	$\mathsf{AC}(E_{\mathtt{Open}})$
	Output To One	$\sum_{\mathcal{G}\in\nabla}\left\lvert\mathcal{P}\setminus\mathcal{G}\right\rvert\cdot\ell$	$\mathsf{AC}(E_{\mathtt{Open}})$
	Verify	$2\cdot n\cdot(n-1)\cdot\kappa$	$\mathsf{AC}(E_{\mathtt{BC}})$

Table 6.3: Total communication cost to realize $\mathcal{F}_{\mathsf{ABB}}$ with M inputs and T total multiplications.

done correctly (instead of requiring all parties to broadcast all of their shares). Finally, **Verify** requires all parties to broadcast, so the network is $E_{\mathtt{BC}}$.

The costs for the running example are now given. The main communication costs in the protocol are those associated with multiplication. This involves passive multiplication in the preprocessing phase to generate the triples, for which costs were analysed in Section 6.5.4, and then opening secrets over authenticated channels in the online phase. For the latter, the following connections are required:

$$E_{\mathtt{Open}}=\Big\{(1,2),(1,5),(1,6),(2,3),(2,5),(2,6),(3,4),(3,5),(3,6),(4,1),\\(4,6),(5,2),(5,3),(5,4),(6,1),(6,2),(6,3),(6,4),(6,5)\Big\}.$$

The number of field elements sent over these channels for one opening is

$$\sum_{i\in[n]}\sum_{\mathcal{G}\in\nabla^{(i)}}|\mathcal{P}\setminus\mathcal{G}|=n\cdot|\nabla|-\sum_{i\in[n]}\sum_{\mathcal{G}\in\nabla^{(i)}}|\mathcal{G}|=n\cdot|\nabla|-\sum_{\mathcal{G}\in\nabla}|\mathcal{G}|.$$

Note that this is more than the number of authenticated channels because, for example, $\mathcal{P}_6$ sends both shares $x^{\mathtt{R}}_{\{1,3,4,6\}}$ and $x^{\mathtt{R}}_{\{2,3,5,6\}}$ to $\mathcal{P}_1$; this contrasts the computation for passive multiplication which requires the same number of field elements as channels.

Thus each opening requires the transmission of $6\cdot11-(3\cdot3+8\cdot4)=25$ finite-field elements, and so a multiplication requires 50 finite-field elements over 19 authenticated channels. Opening final output values still requires a complete network of authenti-

cated channels for the verification, but is performed far less frequently than the basic multiplication operation.

6.7 No Partition

To conclude this chapter, modifications to the protocol are given when a partition $\{\nabla^{(i)}\}_{i\in[n]}$ satisfying $\nabla^{(i)} \neq \varnothing$ for all $i \in [n]$ cannot be found. This occurs for a non-redundant access structure if the number of maximally unqualified sets is smaller than the number of parties.

6.7.1 Existence of Non-Redundant Access Structures with No Partition

First it is necessary to show that this is indeed possible. An example of a 6-party access structure is given in Example 6.2.

Example 6.2. Consider the following access structure:

$$\Delta^+ := \{\{1,2,4\},\{1,3,5\},\{2,3\},\{4,5\},\{6\}\}$$

for which

$$\Gamma^- := \{\{1,6\},\{2,5\},\{3,4\},\{1,2,3\},\{1,4,5\},\{2,6\},\{3,6\},\{4,6\},\{5,6\}\}.$$

One can check that every set in $2^{[6]}$ is a subset or superset of at least one maximally unqualified or minimally qualified set, which determines whether or not the set is qualified, hence these sets indeed form a complete monotone access structure. It is easily verified that this access structure is $\mathscr{Q}_2$ and contains no redundant parties. However, since there are only five sets in Δ^+, there is no surjective map f from the five sets in ∇ to the six parties in $\mathcal{P}$.

6.7.2 Modified Protocol

Recall from Section 6.4.2 that a map $f : \nabla \to \mathcal{P}$ is chosen such that $\mathrm{im}(f)$ is as large as possible. For any $\mathcal{P}_i \in \mathcal{P} \setminus \mathrm{im}(f)$, fix $\nabla^{(i)} := \varnothing$. The modification to the protocol is to apply the protocol as given for all $\mathcal{P}_i \in \mathrm{im}(f)$, and use the standard IT sharing protocol for all $\mathcal{P}_i$ with $\mathcal{P}_i \in \mathcal{P} \setminus \mathrm{im}(f)$. The multiplication protocol then becomes the protocol $\Pi_{\mathtt{AToRNP}}$ given in Figure 6.17.

Subprotocol $\Pi_{\mathtt{AToRNP}}$

It is assumed that, as part of the larger protocol, parties have already called an instance of $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{R}}$ with input $(\mathtt{Initialize}, \Gamma, sid)$ and an instance of $\mathcal{F}_{\mathsf{RZS}}$ with input $(\mathtt{Initialize}, \mathcal{P}, sid)$. Parties hold an additive sharing $[\![v]\!]^{\mathtt{A}}$.

1. The parties agree on a new identifier id_t and call $\mathcal{F}_{\mathsf{RZS}}$ with input $(\mathtt{ZeroSharing}, id_t, \mathcal{P}, sid)$ and receive an additive sharing of zero $[\![t]\!]^{\mathtt{A}}$.
2. The parties agree on a new identifier id_r and call $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{R}}$ with input $(\mathtt{SecretSharing}, id_r, sid)$ to obtain $[\![r]\!]^{\mathtt{R}}$.
3. Each $\mathcal{P}_i \in \mathrm{im}(f)$ defines a sharing $v^{\mathtt{R}}$ by setting
   a) Set $v^{\mathtt{R}}_{\mathcal{G}} \coloneqq r^{\mathtt{R}}_{\mathcal{G}}$ for all $\mathcal{G} \in \nabla \setminus \{\mathcal{G}_j\}_{j\in[n]}$ where $\mathcal{G} \ni \mathcal{P}_i$.
   b) Set $v^{\mathtt{R}}_{\mathcal{G}_i} \coloneqq [\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i} + [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i} - \sum_{\mathcal{G}\in f^{-1}(\{\mathcal{P}_i\})\setminus\{\mathcal{G}_i\}} r^{\mathtt{R}}_{\mathcal{G}}$.
4. For each $i \in [n]$, for all $\mathcal{P}_j \in \mathcal{G}_i$, $\mathcal{P}_i$ sends $v^{\mathtt{R}}_{\mathcal{G}_i}$ to $\mathcal{P}_j$ over a secure channel.
5. Each party $\mathcal{P}_i$ concatenates $\{v^{\mathtt{R}}_{\mathcal{G}} : \mathcal{G} \in \nabla \wedge \mathcal{G} \ni \mathcal{P}_i\}$ into a share vector $[\![v]\!]^{\mathtt{R}}_{\mathcal{P}_i}$.
6. For each $\mathcal{P}_i \notin \mathrm{im}(f)$, party $\mathcal{P}_i$ samples a share vector $[\![u_i]\!]^{\mathtt{R}} \coloneqq [\![[\![v]\!]^{\mathtt{A}}_{\mathcal{P}_i} + [\![t]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]^{\mathtt{R}}$ and distributes the shares.
7. The parties set the final output to be $[\![v]\!]^{\mathtt{R}} \coloneqq [\![v]\!]^{\mathtt{R}} + \sum_{\{i:\mathcal{P}_i\in\mathcal{P}\setminus\mathrm{im}(f)\}} [\![u_i]\!]^{\mathtt{R}}$.

Figure 6.17: Protocol to Convert Additive Shares to Shares Under Any LSSS With No Partition, $\Pi_{\mathtt{AToRNP}}$.

The correctness of the modified protocols follows from the correctnes of $\Pi_{\mathtt{AToROpt}}$ and the fact that the secret-sharing scheme is linear. Security comes from the fact that $\Pi_{\mathtt{AToROpt}}$ is secure and the only difference is that there are some additional shares that are shared with IT security.

Communication Complexity

The only outstanding issue is to adapt the formulae for when f is not surjective. The only difference is in how $\Pi_{\mathtt{AToROpt}}$ is executed and how secrets are opened.

Conversion The graph is

$$E_{\mathtt{AToRNP}} \coloneqq \{(i,j) : i \in \mathrm{im}(f) \wedge \mathcal{G} \in \nabla^{(i)} \wedge \mathcal{P}_j \in \mathcal{G} \setminus \{\mathcal{P}_i\}\} \cup \{(i,j) : \mathcal{P}_i \in \mathcal{P} \setminus \mathrm{im}(f) \wedge \mathcal{G} \in \nabla \wedge \mathcal{P}_j \in \mathcal{G} \setminus \{\mathcal{P}_i\}\}$$

and the set of channels for converting is then $\mathsf{SC}(E_{\mathtt{AToRNP}})$. Recall that secure channels are used to perform the multiplication. The number of field elements over the network

is

$$\sum_{\mathcal{P}_i \in \mathrm{im}(f)} \sum_{\mathcal{G} \in \nabla^{(i)}} (|\mathcal{G}| - 1) + \sum_{\mathcal{P}_i \in \mathrm{im}(f) \setminus \mathcal{P}} \left(\sum_{\mathcal{G} \in \nabla : \mathcal{G} \ni \mathcal{P}_i} (|\mathcal{G}| - 1) + \sum_{\mathcal{G} \in \nabla, \mathcal{G} \not\ni \mathcal{P}_i} |\mathcal{G}| \right)$$
$$= \sum_{\mathcal{G} \in \nabla} (|\mathcal{G}| - 1) + \sum_{\mathcal{P}_i \in \mathrm{im}(f) \setminus \mathcal{P}} \left(\sum_{\mathcal{G} \in \nabla : \mathcal{G} \ni \mathcal{P}_i} (|\mathcal{G}| - 1) + \sum_{\mathcal{G} \in \nabla, \mathcal{G} \not\ni \mathcal{P}_i} |\mathcal{G}| \right).$$

Opening Authenticated channels are used for opening secrets. In order for secrets to be opened with authentication in replicated secret-sharing, it suffices for each party to receive every share it does not hold from some other party. In the case where f is not surjective, the "partial" partition can be used in the same way the "full" partition was used before: each party $\mathcal{P}_i$ is put in charge of sending the shares indexed by sets in $\nabla^{(i)}$ to all parties that do not hold it. Thus the set of channels is:

$$E_{\texttt{OpenNP}} \coloneqq \{(i, j) : \mathcal{P}_i \in \mathrm{im}(f) \ \wedge \ \mathcal{G} \in \nabla^{(i)} \ \wedge \ \mathcal{P}_j \in \mathcal{P} \setminus \mathcal{G}\}$$

and the total number of finite-field elements sent over the network is

$$\sum_{\mathcal{P}_i \in \mathrm{im}(f)} \sum_{\mathcal{G} \in \nabla^{(i)} : \mathcal{G} \ni \mathcal{P}_j} (|\mathcal{P} \setminus \mathcal{G}| - 1) = \sum_{\mathcal{G} \in \nabla} (|\mathcal{P} \setminus \mathcal{G}| - 1).$$

Chapter 7

$\mathscr{Q}_2$ MPC for Large Numbers of Parties

This chapter is based on work published at CT-RSA 2019 under the title Error Detection in Monotone Span Programs with Application to Communication-Efficient Multi-party Computation *[SW19] and was joint work with Nigel Smart. The main contribution of that work referring to error detection for $\mathscr{Q}_2$ access structures was discussed separately in Chapter 3. This chapter focuses on how to use the results in multi-party computation (MPC).*

A new section, Section 7.3, has been introduced. It describes the generation of secrets with information-theoretic security for arbitrary $\mathscr{Q}_2$ access structures, giving a constant factor saving in communication cost.

This chapter In this chaper, the results on error-detection for linear secret-sharing schemes (LSSSs) realizing $\mathscr{Q}_2$ access structures are used to describe an MPC protocol for situations in which the access structure and the number of parties makes the computationally-secure protocol of Chapter 6 too costly.

7.1 Overview

One of the questions left open by Furukawa et al. [FLNW17] was generalizing to an arbitrary number of parties while avoiding replicated secret-sharing. While replicated secret-sharing offers flexibility in being able to realize *any* access structure, unfortunately it can require an exponentially-large number of shares to be held by each party for each shared secret, depending on the access structure: moreover, the num-

ber of pseudorandom function (PRF) keys to generate pseudorandom secret-sharings (PRSSs) depends linearly on the number of maximally-unqualified sets. For example, for a $(256, 128)$-threshold access structure using the former method, the parties would need to agree on roughly as many keys as there are atoms in the observable universe. The exponential factor has significant repercussions for both communication and computation complexity: on the communication side, parties must agree on exponentially-many keys, and to open each secret, exponentially-many field elements must be sent over the network; on the computation side, generating a single shared secret requires each party to evaluate an exponential subset of these keys. For the protocols in this chapter, the costs are essentially quadratic in the number of parties.

A key observation for obtaining better asymptotic efficiency is that in Π_{Online}, given in Figure 4.8, the LSSS need not be multiplicative. There are then three obvious ways to reduce the costs of the overall protocol, which are discussed in this chapter:

- Perform the replicated protocol as described for small numbers of parties (incurring exponential costs), and then convert this by local operations for use in the online phase. In this case, the preprocessing phase is costly but the online phase is very cheap.
- Outsource the preprocessing: then since the parties performing preprocessing can reshare into whatever LSSS is desired, the online phase is cheap.
- Generate random secrets without starting at replicated secret-sharing, and then use (an adapted version of) the information-theoretic (IT) protocol of Maurer from Section 6.4.1 to generate triples in the preprocessing phase.

The first two methods are straightforward and follow from the results in previous chapters. The focus in this chapter is primarily on the third method.

7.2 Preliminaries

The only preliminary information required for this chapter that has not yet been discussed involves how to convert sharings for different LSSSs by *local* operations.

7.2.1 Locally Converting Replicated Shares

Cramer et al. [CDI05] showed how to convert replicated sharings into sharings of any other LSSS by local computations. This procedure is given in Figure 7.1. For correctness, notice that for all $i \in [n]$,

$$\sum_{\mathcal{G}\in\nabla:\mathcal{P}_i\in\mathcal{G}} r^{\mathsf{R}}_{\mathcal{G}} \cdot [\![1_{\mathcal{G}}]\!]_{\mathcal{P}_i} = \sum_{\mathcal{G}\in\nabla} r^{\mathsf{R}}_{\mathcal{G}} \cdot [\![1_{\mathcal{G}}]\!]_{\mathcal{P}_i}$$

since $[\![1_{\mathcal{G}}]\!]_{\mathcal{P}_i} = \mathbf{0}$ if $\mathcal{P}_i \notin \mathcal{G}$ by definition of $[\![1_{\mathcal{G}}]\!]$. Then correctness holds by linearity of the LSSS.

Subprotocol $\Pi_{\mathtt{RToAny}}$

This protocol was given by Cramer et al. [CDI05]. At this point in the protocol, the parties have a replicated sharing $[\![x]\!]^{\mathsf{R}}$, where $\mathcal{P}_i$ holds $[\![x]\!]^{\mathsf{R}}_{\mathcal{P}_i}$, and will convert it to a sharing of a different LSSS, $[\![\cdot]\!]$.

Initialize The parties agree on a set of sharings of 1, $\{[\![1_{\mathcal{G}}]\!]\}_{\mathcal{G}\in\nabla}$ where for each $\mathcal{G} \in \nabla$, $\mathrm{supp}([\![1_{\mathcal{G}}]\!]) \subseteq \{j \in [m] : \rho(j) \in \mathcal{G}\}$.

Convert Each party $\mathcal{P}_i$ computes

$$\sum_{\mathcal{G}\in\nabla:\mathcal{P}_i\in\mathcal{G}} r^{\mathsf{R}}_{\mathcal{G}} \cdot [\![1_{\mathcal{G}}]\!]_{\mathcal{P}_i}.$$

Figure 7.1: Protocol to Convert Replicated Secret-Sharing to Any LSSS, $\Pi_{\mathtt{RToAny}}$.

7.3 Generating Information-Theoretic Uniformly-Random Secrets

A functionality for generating uniformly-random secrets according to *any* LSSS is given in Figure 7.2. Notice that this functionality produces t random sharings at a time, for reasons that will be explained later. For a small number of parties, the key-setup required in realizing $\mathcal{F}^{\mathtt{Any}}_{\mathsf{RSS}}$, via $\Pi^{\mathsf{R}}_{\mathsf{RSS}}$ from Chapter 6 is modest. However, the cost is asymptotically $O(2^n/\sqrt{n})$ for threshold access structures, so for large numbers of parties the generation of random shares is unlikely to yield good results.

The naïve method of obtaining a random sharing is for every party to sample a random secret, distribute shares, and for the parties to take the sum; then, since at least one party is honest, the resulting secret is uniform. Chapter 3 used redundancy in an LSSS for a $\mathscr{Q}_2$ access structure to reduce the cost of opening secrets with active

security. The goal of this section is to provide a methodology for generating uniformly-random secrets more efficiently than by the naïve method by exploiting redundancy in the $\mathscr{Q}_2$ access structure once more.

Functionality $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{Any}}$

Initialize On input $(\mathtt{Initialize}, \Gamma, sid)$ from all honest parties and $\mathcal{S}$, where sid is a new session identifier, $\mathcal{P}$ is the set of parties, and Γ is an access structure, await further messages.

Sharing of Secret On input $(\mathtt{SecretSharing}, \{id_k\}_{k=1}^t, sid)$ from all honest parties and $\mathcal{S}$, the functionality does the following:

1. Await a set of shares $\{[\![x^k]\!]_{\mathcal{A}}\}_{k=1}^t$ from $\mathcal{S}$ and error vectors $\{\boldsymbol{\varepsilon}^k\}_{k=1}^t$ where $\boldsymbol{\varepsilon}^k \in \mathbb{F}^m$ for all $k \in [t]$.
2. For each $k \in [t]$, sample $\mathbf{x}^k \leftarrow \mathcal{U}\left(\{\mathbf{x} \in \mathbb{F}^d : M_{\mathcal{A}}\mathbf{x}^k = [\![x^k]\!]\}\right)$ and set $[\![x^k]\!] := M \cdot \mathbf{x}^k + \boldsymbol{\varepsilon}^k$.
3. For each $i \in [n] \setminus A$, send $\{[\![x^k]\!]_{\mathcal{P}_i}\}_{k\in[d]}$ to honest $\mathcal{P}_i$.

Figure 7.2: Functionality for Secret-Sharings of Random Secrets for Any LSSS, $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{Any}}$.

Using a Qualified Set of Parties

Perhaps the most obvious way of obtaining uniformly-random shares is for every party in any set of qualified parties to generate a random secret and distribute the shares, and to take the sum of these secrets. Since the set is qualified, it contains at least one honest party and so the resulting secret is uniformly-random. For an (n,t)-threshold access structure, this gives 1 sharing from $t+1$. However, when the number of parties is large, this leaves a lot of work to a small set of parties.

Generalization of Damgård-Nielsen

Damgård and Nielsen [DN07] showed how to do this in a much more symmetrical manner in the threshold setting. More specifically, they showed how to obtain $n-t$ random secrets from n secrets, where t is a constant fraction of n satisfying $n-t > t$. This method exploits the fact that any set of $n-t$ parties contains an honest party, so if every random sharing contains a contribution from at least one honest party then the secret is uniform. In this section this method is generalized, which is crucial for obtaining (general) asymptotic improvement to protocols making use of Beaver's circuit randomization.

Approach for Threshold $\mathscr{Q}_2$ Access Structures The technique of [DN07] is as follows. Each party $\mathcal{P}_i$ samples some $[\![r_i]\!]$ and distributes the shares. Then the parties

compute

$$\begin{pmatrix} [\![s_1]\!] \\ \vdots \\ [\![s_{n-t}]\!] \end{pmatrix} := V^\top \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_n]\!] \end{pmatrix}$$

where V is the Vandermonde matrix

$$\begin{pmatrix} 1^0 & \cdots & 1^{n-t-1} \\ \vdots & \ddots & \vdots \\ n^0 & \cdots & n^{n-t-1} \end{pmatrix}.$$

The key observation is that if the adversary corrupts some set of parties $\mathcal{A}$ (of size t), then the remaining set of $n-t$ parties, $\mathcal{P} \setminus \mathcal{A}$, can still force the resulting set of shares $([\![s_1]\!], \ldots, [\![s_{n-t}]\!])^\top$ to be a set of uniformly-random secrets since every submatrix of V consisting of $n-t$ rows is linearly independent. Explicitly,

$$\begin{pmatrix} [\![s_1]\!] \\ \vdots \\ [\![s_{n-t}]\!] \end{pmatrix} := V^\top \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_n]\!] \end{pmatrix} = {V_{\mathcal{P}\setminus\mathcal{A}}}^\top \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_n]\!] \end{pmatrix}_{\mathcal{P}\setminus\mathcal{A}} + {V_{\mathcal{A}}}^\top \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_n]\!] \end{pmatrix}_{\mathcal{A}}$$

where the first summand is always uniformly-random since ${V_{\mathcal{P}\setminus\mathcal{A}}}^\top$ has full rank and $([\![r_1]\!], \ldots, [\![r_n]\!])_{\mathcal{P}\setminus\mathcal{A}}{}^\top$ is a vector of shares that are generated by honest parties.

Approach for General $\mathscr{Q}_2$ Access Structures In the general case, the argument is that for any set of parties $\mathcal{A}$ corrupted by the adversary, the rows owned by the remaining (honest) parties $\mathcal{P} \setminus \mathcal{A}$ have rank d, where d is the rank of the monotone span program (MSP) matrix. The reason this is sufficient is that it means that whatever set of parties is corrupted, the shares that are output are always shares of uniformly-random secrets. In fact, the necessary property required for using the matrix for randomness extraction is exactly that it should be a share-reconstructable MSP for the $\mathscr{Q}_2$ access structure. The notion of share-reconstructability was introduced in Section 3.6. Note that this implies it is necessary for the access structure to be $\mathscr{Q}_2$, since for any unqualified set $\mathcal{U} \in \Delta$, the target vector $\mathbf{t}$ necessarily lies in the span of the rows owned by $\mathcal{P} \setminus \mathcal{U}$.

These observations suggest that the parties can perform randomness extraction using any share-reconstructable MSP that realizes the same access structure. However, it is possible to do better than this: given an access structure Γ, let

$$\overline{\Delta} := \left\{ \mathcal{U} \in 2^{\mathcal{P}} : \mathcal{P} \setminus \mathcal{U} \in \Gamma \right\}$$

and consider the complete monotone access structure it defines. (Note that it is indeed a complete monotone access structure since $\overline{\Delta}$ is trivially closed under the subset operation as Γ is closed under the superset operation.) One could call this access structure the "$\mathscr{Q}_2$ closure" of Γ since if Δ is $\mathscr{Q}_2$ then $\overline{\Delta}$ is the largest superset of Δ that is still $\mathscr{Q}_2$. Now consider the complement of $\overline{\Delta}$ in $2^{\mathcal{P}}$: since the access structure Γ is complete, a set is in Γ if and only if it is not in Δ; thus the set of complements can be expressed as $\overline{\Gamma} = \{\mathcal{Q} \in 2^{\mathcal{P}} : \mathcal{P} \setminus \mathcal{Q} \notin \Gamma\}$. Hence the "$\mathscr{Q}_2$ closure" is exactly the dual access structure. (See Section 2.4.1 for the definition of dual access structure.)

The result of Cramer et al. generalizes in the following way: one can use the matrix from any share-reconstructable MSP realizing the dual access structure Γ^* for randomness extraction (or indeed realizing any access structure Γ' satisfying $\Gamma^* \subseteq \Gamma' \subseteq \Gamma$). In the general case, this means that the parties must together provide m sharings, and will obtain d. (Recall that the MSP matrix is in $\mathbb{F}^{m \times d}$.) The MSP matrix used to perform randomness extraction need not be the same as the matrix used to share the secrets. This generalization coincides with the original construction that used a $(n, n-t)$ Vandermonde matrix for an (n,t)-threshold access structure, since this is exactly the MSP matrix for Shamir's secret-sharing (which is share reconstructable) of the dual access structure, $(n, n-t-1)$-threshold.

Note that MSPs with these properties always exist because replicated secret-sharing is always share-reconstructable (see Theorem 3.1), but will not always be efficient. However, note that there is often a saving over the naïve method of obtaining 1 sharing from n sharings: for example, in an (n,t)-threshold scheme, the parties generate $m^{\mathbb{R}} = n \cdot \binom{n-1}{n-(n-t-1)-1}$ sharings and obtain $d^{\mathbb{R}} = \binom{n}{n-(n-t-1)}$, which is an average of obtaining 1 sharing from $t+1$ sharings. This matches the intuition that the method described above is "as efficient" as using a qualified set of parties to obtain 1 sharing from $t+1$ sharings, but the load is balanced amongst all the parties.[1] Unfortunately, the technique still incurs a linear cost, but this method works regardless of the access structure, and not just for threshold schemes. Nevertheless, this is a constant saving in communication cost. Note that in practice, using the matrix $M^{\mathbb{R}}$ is relatively cheap since the matrix is sparse so the cost of computing this matrix-vector multiplication is modest. Since for large numbers of parties the exponential blow-up of replicated secret-sharing is exactly what this chapter seeks to avoid, in general a more efficient share-reconstructable MSP is likely to be preferable, if one exists.

[1] Clearly in this case the Vandermonde matrix should be used: the example is only intended to give intuition for the asymptotics.

These observations give rise to the protocol given in Figure 7.3. To differentiate between the MSP in which the secrets are shared and the MSP used for randomness extraction, the variables relating to the latter have the superscript $\mathtt{R}$, which has been used previously to indicate replicated secret-sharing, which may be used here but is not required.

Protocol $\Pi^{\mathtt{Any}}_{\mathsf{RSS}}$

Initialize Given an MSP, $[\![\cdot]\!]$, realizing a $\mathscr{Q}_2$ access structure Γ, agree on any share-reconstructable MSP realizing Γ^*, and let $M^{\mathtt{R}} \in \mathbb{F}^{m^{\mathtt{R}} \times d^{\mathtt{R}}}$ be the corresponding MSP matrix.

Sharing of Secret

1. Each party $\mathcal{P}_i$ samples a set $\{r_k\}_{k=1}^{|\rho^{\mathtt{R}}(i)|} \leftarrow \mathcal{U}(\mathbb{F})$.
2. Each party $\mathcal{P}_i$ generates sharings $\{[\![r_k]\!]\}_{k=1}^{|\rho^{\mathtt{R}}(i)|}$ and distributes the shares.
3. The parties compute
$$\begin{pmatrix} [\![s_1]\!] \\ \vdots \\ [\![s_{d^{\mathtt{R}}}]\!] \end{pmatrix} := {M^{\mathtt{R}}}^{\top} \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_{m^{\mathtt{R}}}]\!] \end{pmatrix}$$
and output $\{[\![s_k]\!]\}_{k=1}^{d^{\mathtt{R}}}$.

Figure 7.3: Protocol for Secret-Sharings of Multiple Random Secrets for Any LSSS, $\Pi^{\mathtt{Any}}_{\mathsf{RSS}}$.

Lemma 7.1. *The secret-sharings $\{[\![s_k]\!]\}_{k=1}^{d^{\mathtt{R}}}$ generated in $\Pi^{\mathtt{Any}}_{\mathsf{RSS}}$ are shares of independent and uniformly-random secrets with respect to the access structure Γ.*

Proof. The shares have the same access structure as the original secrets as they are computed as a linear combination of secret-shared data (which is performed locally). It remains to show that the resulting secrets are uniformly-random in the secret-sharing space.

Let $\mathcal{A}$ denote the set of parties corrupted by $\mathcal{A}$. Then

$$\begin{pmatrix} [\![s_1]\!] \\ \vdots \\ [\![s_{d^{\mathtt{R}}}]\!] \end{pmatrix} := {M^{\mathtt{R}}}^{\top} \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_{m^{\mathtt{R}}}]\!] \end{pmatrix} = {M^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}}}^{\top} \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_{m^{\mathtt{R}}}]\!] \end{pmatrix}_{\mathcal{P}\setminus\mathcal{A}} + {M^{\mathtt{R}}_{\mathcal{A}}}^{\top} \cdot \begin{pmatrix} [\![r_1]\!] \\ \vdots \\ [\![r_{m^{\mathtt{R}}}]\!] \end{pmatrix}_{\mathcal{A}}.$$

Since $\mathcal{A}$ is a corrupt set of parties and $M^{\mathtt{R}}$ realizes the dual access structure, the set $\mathcal{P}\setminus\mathcal{A}$ of honest parties is qualified. Moreover, because the MSP is share-reconstructable, the matrix $M^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}}$ has rank $d^{\mathtt{R}}$. Thus the vector $([\![r_1]\!], \cdots, [\![r_{m^{\mathtt{R}}}]\!])^{\top}_{\mathcal{P}\setminus\mathcal{A}} \cdot M^{\mathtt{R}}_{\mathcal{P}\setminus\mathcal{A}}$ is a vector of

independent, uniformly-distributed secrets in $\mathbb{F}^{d^{\mathrm{R}}}$, which means $(\llbracket s_1 \rrbracket, \ldots, \llbracket s_{d^{\mathrm{R}}} \rrbracket)^\top$ is a vector of independent, uniformly-random secrets. □

The threshold protocol described by Cramer et al. [CDI05] was not proved secure in the universal composability (UC) framework. Indeed, the main difficulty in doing so is defining the right functionality and showing it is possible to simulate.

Theorem 7.1. *The protocol $\Pi^{\mathtt{Any}}_{\mathsf{RSS}}$ UC-securely realizes the functionality $\mathcal{F}^{\mathtt{Any}}_{\mathsf{RSS}}$ in the plain model against a static, active adversary.*

Proof. The simulator is given in Figure 7.4.

Simulator $\mathcal{S}^{\mathtt{Any}}_{\mathsf{RSS}}$

Initialize Call $\mathcal{F}^{\mathtt{Any}}_{\mathsf{RSS}}$ with input $(\mathtt{Initialize}, \Gamma, sid)$.

Sharing of Secret

1. On behalf of each (emulated) honest party $\mathcal{P}_i$, sample a set of vectors $\{\mathbf{x}^k\}_{k \in \rho^{\mathrm{R}^{-1}}(i)} \leftarrow \mathcal{U}(\mathbb{F}^d)$ and compute sharings $\llbracket r_k \rrbracket := M \cdot \mathbf{x}^k$.
2. Distribute and receive shares as follows:
   - For each $k \in [m^{\mathrm{R}}] \setminus \rho^{\mathrm{R}^{-1}}(A)$, send $\llbracket r_k \rrbracket_{\mathcal{A}}$ to $\mathcal{A}$.
   - For each $k \in [m^{\mathrm{R}}] \cap \rho^{\mathrm{R}^{-1}}(A)$, await a share vector $\llbracket r_k \rrbracket_{\mathcal{P} \setminus \mathcal{A}}$.
   - Pad $\llbracket r_k \rrbracket_{\mathcal{P} \setminus \mathcal{A}}$ with 0's to become a full share vector $\llbracket r_k \rrbracket$ and set $\boldsymbol{\varepsilon}^k := N \cdot \llbracket r_k \rrbracket$.
   - Sample $\mathbf{x}^k \leftarrow \mathcal{U}\left(\left\{\mathbf{x} \in \mathbb{F}^d : M_{\mathcal{P} \setminus \mathcal{A}} \cdot \mathbf{x} = \llbracket r_k \rrbracket_{\mathcal{P} \setminus \mathcal{A}} - \boldsymbol{\varepsilon}^k_{\mathcal{P} \setminus \mathcal{A}}\right\}\right)$ and set $\llbracket r_k \rrbracket := M \cdot \mathbf{x}^k + \boldsymbol{\varepsilon}^k$.
   - Compute $(\llbracket s_1 \rrbracket, \ldots, \llbracket s_{d^{\mathrm{R}}} \rrbracket)^\top := (\llbracket r_1 \rrbracket, \ldots, \llbracket r_{m^{\mathrm{R}}} \rrbracket)^\top \cdot M^{\mathrm{R}}$.
   - Compute $\left(\tilde{\boldsymbol{\varepsilon}}^1, \ldots, \tilde{\boldsymbol{\varepsilon}}^{d^{\mathrm{R}}}\right)^\top := \left(\boldsymbol{\varepsilon}^1, \ldots, \boldsymbol{\varepsilon}^{m^{\mathrm{R}}}\right)^\top \cdot M^{\mathrm{R}}$.
   - Send $\{\llbracket s_k \rrbracket_{\mathcal{A}}\}_{k \in [d^{\mathrm{R}}]}$ and $\{\tilde{\boldsymbol{\varepsilon}}^k\}_{k \in [d^{\mathrm{R}}]}$ to $\mathcal{F}^{\mathtt{Any}}_{\mathsf{RSS}}$.
3. (No simulation is required for this step.)

Figure 7.4: Simulator $\mathcal{S}^{\mathtt{Any}}_{\mathsf{RSS}}$ for $\mathcal{F}^{\mathtt{Any}}_{\mathsf{RSS}}$.

The tricky part about this simulation is to show correctness: that is, to ensure that the final output shares of (real) honest parties form a valid share vector with the shares held by corrupt parties if they act honestly. Since Lemma 7.1 implies the final share vectors are uniform regardless of the set $\mathcal{A}$ of parties the adversary corrupts, and the linear combination is public, it suffices to show that this holds for a single random vector generated between $\mathcal{S}$ and $\mathcal{A}$, i.e. where $\mathcal{S}$ sends shares $\llbracket r \rrbracket_{\mathcal{A}}$ for the sharing $\llbracket r \rrbracket$ of a random secret r, and $\mathcal{A}$ responds with shares $\llbracket r' \rrbracket_{\mathcal{P} \setminus \mathcal{A}}$, so that the final sharing held by corrupt and honest parties should be $\llbracket r + r' \rrbracket$. (Note that $\mathcal{S}$ must send the vector first as the adversary may be rushing.)

The argument is similar to that given in the proof of Theorem 5.1. Suppose $\mathcal{S}$ generates $[\![r]\!]$ honestly and sends $[\![r]\!]_{\mathcal{A}}$ to $\mathcal{A}$. Then $\mathcal{A}$ responds with a set $[\![r']\!]_{\mathcal{A}}$. Let $[\![x]\!]_{\mathcal{P}\setminus\mathcal{A}} \coloneqq [\![r]\!]_{\mathcal{P}\setminus\mathcal{A}} + [\![r']\!]_{\mathcal{P}\setminus\mathcal{A}}$. Then $\mathcal{S}$ samples $[\![x']\!]_{\mathcal{A}}$ such that $([\![x]\!]_{\mathcal{P}\setminus\mathcal{A}}, [\![x']\!]_{\mathcal{A}})$ is a valid share vector. Then it sends $[\![x']\!]_{\mathcal{A}}$ to $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{Any}}$, which samples $[\![x'']\!]_{\mathcal{P}\setminus\mathcal{A}}$ so that $([\![x'']\!]_{\mathcal{P}\setminus\mathcal{A}}, [\![x']\!]_{\mathcal{A}})$ is a valid share vector and sends the shares of $[\![x'']\!]_{\mathcal{P}\setminus\mathcal{A}}$ to (real) honest parties, which output these shares to the environment.

Let $\mathbf{r}'$ be defined as the "actual" vector chosen by $\mathcal{A}$ so that $[\![r']\!] = M \cdot \mathbf{r}'$ and let $\mathbf{r}$ be the vector chosen by $\mathcal{S}$ to generate $[\![r]\!]$ as $M \cdot \mathbf{r}$. Let $\mathbf{x} \coloneqq \mathbf{r} + \mathbf{r}'$. The process of sampling, restricting, sending and resampling means the final share vector lies in the space

$$\begin{aligned}
&M \cdot \left(M_{\mathcal{A}}^{-1} \cdot \left(M \cdot \left(M_{\mathcal{P}\setminus\mathcal{A}}^{-1} \cdot [\![x]\!]_{\mathcal{P}\setminus\mathcal{A}}\right)_{\mathcal{A}}\right)\right) \\
=&M \cdot \left(M_{\mathcal{A}}^{-1} \cdot (M_{\mathcal{A}} \cdot (\mathbf{x} + \ker(M_{\mathcal{P}\setminus\mathcal{A}})))\right) \\
=&M \cdot (\mathbf{x} + \ker(M_{\mathcal{P}\setminus\mathcal{A}}) + \ker(M_{\mathcal{A}})) \\
=&M \cdot \mathbf{x} + M \cdot \ker(M_{\mathcal{P}\setminus\mathcal{A}}) + M \cdot \ker(M_{\mathcal{A}})
\end{aligned}$$

where for a matrix A, here A^{-1} denotes taking a preimage. Now the first summand is equal to the "actual" randomness vector the parties used to generate the secrets that were sent, and is therefore indeed a particular solution to the equation above. The second summand is any sharing supported by $\mathcal{P}\setminus\mathcal{A}$, and since there is always a sharing of 1 with support $\mathcal{P}\setminus\mathcal{A}$, the final share vector is a sharing of any secret in the field. The third summand corresponds to a sharing supported by $\mathcal{A}$, which by Lemma 3.1 is a sharing of 0 since the access structure is $\mathscr{Q}_2$. Consequently, the final share vector obtained by the environment from the real honest parties' shares and the shares generated by $\mathcal{A}$ is a valid share vector.

Finally, observe that if the adversary does not send shares to honest parties that correspond to a valid share vector, then the simulator is able to compute the errors using the error-detection matrix N (since the access structure is $\mathscr{Q}_2$). This is analogous to computing the syndrome of a codeword in Coding Theory. The functionality introduces the same errors in the ideal world, and so the environment is not able to distinguish between the real and ideal worlds. □

7.4 Information-Theoretic Preprocessing

Only linear operations are performed on shares in the protocol Π_{Online}, since the non-linearity of multiplication is handled by the Beaver triples. The consequence of this is

that the LSSS used in the online phase need not be multiplicative. Typically, multiplicative LSSSs realizing a given access structure require at least the same number of shares in total as non-multiplicative schemes.

The idea in this section is to execute the preprocessing phase using a multiplicative LSSS to generate Beaver triples, but then to convert to a LSSS with a smaller number of shares by local computations so that the online phase may proceed with this "more efficient" LSSS.

Almost the entire preprocessing protocol is the same as in Chapter 6, so the full protocol is not given in this section. Instead, only the information-theoretic multiplication subprotocol will be given. This passively-secure subprotocol can be slotted directly into the preprocessing protocol of Chapter 6, and the IT functionality $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{Any}}$ can be used to obtain triple multiplicands (i.e. a's and b's) instead of the functionality $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{R}}$.

7.4.1 LSSS to Multiplicative LSSS

In order to use the IT multiplication protocol of Maurer [Mau06] in which the additive sharing of the product is converted back into the original LSSS, as outlined in Section 6.4.1, the LSSS must be multiplicative.

The first step of IT preprocessing is to recall the observation of Cramer et al. [CDM00] that any LSSS realizing a $\mathscr{Q}_2$ access structure can be made multiplicative by at most doubling the total number of shares. The observation is that if a secret is shared according to an access structure Γ *and* its dual Γ^*, written as $(\llbracket\cdot\rrbracket, \llbracket\cdot\rrbracket^*)$ then by local computations the parties can obtain an additive sharing of the secret. The details of the construction are not given here as they are not important for the protocol. Thus the first task is to find an optimal multiplicative LSSS for the access structure (in terms of total number of shares); if one does not exist then the task is to find an optimal LSSS and then to perform the computation to convert it to a multiplicative scheme.

The original LSSS is denoted by $\llbracket\cdot\rrbracket$ and the LSSS derived from for computing products locally by $\llbracket\cdot\rrbracket^{\Pi}$. Using this notation, it always holds that either $\llbracket\cdot\rrbracket^{\Pi} = \llbracket\cdot\rrbracket$ or $\llbracket\cdot\rrbracket^{\Pi} = (\llbracket\cdot\rrbracket, \llbracket\cdot\rrbracket^*)$. This notation should not be confused with $\llbracket\cdot\rrbracket^{[2]}$, which is computed with respect to an initial LSSS $\llbracket\cdot\rrbracket$ and denotes any LSSS realizing the access structure of the product of two secrets under $\llbracket\cdot\rrbracket$, which is typically simply additive secret-sharing, $\llbracket\cdot\rrbracket^{\mathtt{A}}$.

7.4.2 Multiplicative LSSS to Preprocessing

Once the parties have generated a large number of secrets according to the multiplicative LSSS using $\mathcal{F}_{\mathsf{RSS}}^{\mathtt{Any}}$, they can obtain an additive sharing of the product of two secrets by local operations (by the definition of multiplicativity) and then execute the IT protocol $\Pi_{\mathtt{AToAny}}$ from Section 6.4 that turns an additive sharing into a sharing under any other LSSS. In the case that $[\![\cdot]\!]^{\Pi} = ([\![\cdot]\!], [\![\cdot]\!]^*)$, i.e. where it is necessary to extend the LSSS in order to make it multiplicative, a couple of points must be noted.

Firstly, since the online phase does not require multiplicativity, it suffices for the parties to reshare their summands under the *original* LSSS, not in the product LSSS. This reduces the amount of communication required when establishing the sharing of the product of secrets.

Secondly, if a and b are secret-shared in the product LSSS, i.e. as $[\![a]\!]^{\Pi}$ and $[\![b]\!]^{\Pi}$, and only the original LSSS is going to be used in the online phase, then these sharings need to be converted to the original (non-multiplicative) sharings so that all three parts of the triple are shared according to the same scheme. Fortunately, this can be done very efficiently. It was observed by Cramer et al. [CDI05] that some LSSSs can be converted to other LSSSs that realize the same access structure by *local* computations. Moreover, for a product LSSS generated from an LSSS by the method from [CDM00] (outlined in the previous section), this is indeed the case for converting back into the original scheme. In this case, the "local computation" involves each party $\mathcal{P}_i$ taking the shares $([\![x]\!]_{\mathcal{P}_i}, [\![x]\!]^*_{\mathcal{P}_i})$ of the secret x and simply discarding $[\![x]\!]^*_{\mathcal{P}_i}$.

In summary, the protocol involves parties generating shares $[\![a]\!]^{\Pi}$ and $[\![b]\!]^{\Pi}$, performing the local computation to obtain $[\![c]\!]^{\mathtt{A}}$, and then resharing this as $[\![c]\!]$ defined as $\sum_{i\in[n]} [\![[\![c]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]$. The subprotocol $\Pi^{\mathtt{IT}}_{\mathsf{Mult}}$ for generating triples after obtaining $[\![a]\!]^{\Pi}$ and $[\![b]\!]^{\Pi}$ is given in Figure 7.5.

Subprotocol $\Pi^{\mathtt{IT}}_{\mathsf{Mult}}$

At this point, the parties are assumed to hold sharings $[\![a]\!]^{\Pi}$ and $[\![b]\!]^{\Pi}$, where $[\![\cdot]\!]^{\Pi}$ is the product LSSS derived from $[\![\cdot]\!]$.

Multiply

1. Each party $\mathcal{P}_i$ computes $[\![c]\!]^{\mathtt{A}}_{\mathcal{P}_i} := \langle \boldsymbol{\mu}^{(i)}, [\![a]\!]^{\Pi}_{\mathcal{P}_i} \otimes [\![b]\!]^{\Pi}_{\mathcal{P}_i} \rangle$ where $\boldsymbol{\mu}^{(i)}$ is the vector from Section 2.4.4.
2. Each party $\mathcal{P}_i$ samples a share vector $[\![[\![c]\!]^{\mathtt{A}}_{\mathcal{P}_i}]\!]$ for the summand $[\![c]\!]^{\mathtt{A}}_{\mathcal{P}_i}$, and distributes the shares over secure channels.
3. Each party $\mathcal{P}_i$ awaits shares from all parties and computes $[\![c]\!]_{\mathcal{P}_i} := \sum_{j\in[n]} [\![[\![c]\!]^{\mathtt{A}}_{\mathcal{P}_j}]\!]_{\mathcal{P}_i}$.
4. The parties perform local computations to convert $[\![a]\!]^{\Pi}$ to $[\![a]\!]$ and $[\![b]\!]^{\Pi}$ and $[\![b]\!]$.
5. The parties output $([\![a]\!], [\![b]\!], [\![c]\!])$.

Figure 7.5: Subprotocol for Information-Theoretic Multiplication, $\Pi^{\mathtt{IT}}_{\mathsf{Mult}}$.

The arithmetic circuit can now be evaluated in the usual way (i.e. using Π_{Online}).

7.5 Communication Complexity

In this section, analysis of the communication complexity will be expressed in terms of the costs for Shamir's secret-sharing so that the results may be compared with other protocols in the honest-majority setting, but it is important to note that many of the techniques from this chapter – and indeed throughout this thesis – apply to general $\mathscr{Q}_2$ access structures, which many other protocols do not.

For a threshold access structure, the topologies from Section 6.5.4 and Section 6.6.3 can be expressed as follows:

$$
\begin{aligned}
E^i_{\mathtt{BC}} &= \{i\} \times ([n] \setminus \{i\}) \\
E_{\mathtt{BC}} &= \{(i,j) \in [n]^2 : i \neq j\} \\
E^i_{\mathtt{Output}} &= ([n] \setminus \{i\}) \times \{i\} \\
E_{\mathtt{Open}} &= \{(i,j) \in [n]^2 : j = i + k \bmod n \text{ for } k \in [t]\} \\
E_{\mathtt{AToR}} &= \{(i,j) \in [n]^2 : j = i + k \bmod n \text{ for } k \in [n-t-1]\}
\end{aligned}
$$

7.5.1 Preprocessing

In this section, the cost of generating preprocesing for a threshold access structure in two different ways is given:

- Parties execute the protocol from Chapter 6 and then convert using $\Pi_{\texttt{RToAny}}$.
- Parties execute the IT protocol from Section 7.4.

Preprocessing with Computational Security

If the parties use replicated secret-sharing and convert, then the cost is shown in Table 7.1, which is the same as Table 6.1 but for a threshold access structure. Notice that $\Pi_{\texttt{RToAny}}$ can be applied after the triple has been generated but *before* it is sacrificed, which would save communication costs further.

Procedure	Number of bits	Channels
Initialize		
PRZS key commitments	$2 \cdot \kappa \cdot n \cdot (n-1)$	$\mathsf{AC}(E_{\texttt{BC}})$
Opening commitments	$3 \cdot \kappa \cdot n \cdot (n-1)$	$\mathsf{SC}(E_{\texttt{BC}})$
PRSS key commitments	$2 \cdot \kappa \cdot n \cdot \binom{n-1}{n-t-1} \cdot (n-t-1)$	$\mathsf{AC}(E_{\texttt{BC}})$
Opening commitments	$3 \cdot \kappa \cdot n \cdot \binom{n-1}{n-t-1} \cdot (n-t-1)$	$\mathsf{SC}(E_{\texttt{BC}})$
Mask		
Generation	n/a	n/a
Open	$(n-1) \cdot \ell$	$\mathsf{SC}(E^i_{\texttt{Output}})$
Triples		
Generation (a and b)	n/a	n/a
Generation (c)	$(1 + \lceil \sigma/\ell \rceil) \cdot n \cdot (n-t-1) \cdot \ell \cdot T$	$\mathsf{SC}(E_{\texttt{AToR}})$
Sacrifice (Open)	$3 \cdot \lceil \sigma/\ell \rceil \cdot n \cdot t \cdot \ell \cdot T$	$\mathsf{AC}(E_{\texttt{Open}})$
Sacrifice (Check)	n/a	n/a

Table 7.1: Total preprocessing communication cost to realize $\mathcal{F}_{\mathsf{Prep}}$ performing T multiplications using $\mathcal{F}^{\mathbb{R}}_{\mathsf{RSS}}$ and $\mathcal{F}_{\mathsf{RZS}}$.

Preprocessing with Statistical Security

Whereas PRSSs can be generated non-interactively after the key-setup phase, Table 7.2 shows the amount of communication required to generate T triples.

The key point to notice is that for a small number of parties, the binomial term is small enough that the key-setup phase is tractible; for a larger number of parties, the IT protocol essentially replaces the binomial term with a quadratic term that also depends on the number of multiplication gates, T, in the circuit. Note that in the generation of the triple, obtaining c is only a constant factor more expensive (in terms of communica-

Procedure	Number of bits	Channels
Initialize	n/a	n/a
Mask		
Generation	$(1+\lceil\sigma/\ell\rceil)\cdot\lceil\frac{T}{n-t}\rceil\cdot n\cdot(n-1)\cdot\ell$	$\mathsf{SC}(E_{\mathtt{BC}})$
Open	$(n-1)\cdot\ell$	$\mathsf{SC}(E^i_{\mathtt{Output}})$
Triples		
Generation (a and b)	$(1+\lceil\sigma/\ell\rceil)\cdot\lceil\frac{T}{n-t}\rceil\cdot n\cdot(n-1)\cdot\ell$	$\mathsf{SC}(E_{\mathtt{BC}})$
Generation (c)	$(1+\lceil\sigma/\ell\rceil)\cdot n\cdot(n-1)\cdot\ell\cdot T$	$\mathsf{SC}(E_{\mathtt{BC}})$
Sacrifice (Open)	$3\cdot\lceil\sigma/\ell\rceil\cdot n\cdot t\cdot\ell\cdot T$	$\mathsf{AC}(E_{\mathtt{Open}})$
Sacrifice (Check)	$n\cdot(n-1)\cdot\kappa$	$\mathsf{AC}(E_{\mathtt{BC}})$

Table 7.2: Total preprocessing communication cost to realize $\mathcal{F}_{\mathsf{Prep}}$ performing T multiplications using $\mathcal{F}^{\mathtt{Any}}_{\mathsf{RSS}}$.

tion) than the method using replicated secret-sharing since the term $n-t-1=O(n)$ is replaced by $n-1$.

7.5.2 Online Phase

The total cost of the actively-secure protocol is given in Table 7.3. There is a clear advantage to converting to Shamir's secret-sharing for executing the online phase as the binomial factors that would appear in Table 6.3 by considering a threshold access structure are replaced with factors linear in n.

Procedure	Number of bits	Channels
Input	$n\cdot(n-1)\cdot\ell$	$\mathsf{AC}(E^i_{\mathtt{BC}})$
Add	n/a	n/a
Multiply	$2\cdot n\cdot t\cdot\ell\cdot T$	$\mathsf{AC}(E_{\mathtt{Open}})$
Output To One	$(n-1)\cdot\ell$	$\mathsf{SC}(E^i_{\mathtt{Output}})$
Output To All	$n\cdot t\cdot\ell\cdot T$	$\mathsf{AC}(E_{\mathtt{Open}})$
Verify	$2\cdot n\cdot(n-1)\cdot\kappa$	$\mathsf{AC}(E_{\mathtt{BC}})$

Table 7.3: Total online communication cost to realize $\mathcal{F}_{\mathsf{ABB}}$ performing T multiplications.

7.5.3 Comparison with Other Protocols

Comparison with Maurer

The protocol of Maurer [Mau06] is not realized in the preprocessing model, and it only offers passive security. Nonetheless, it is interesting to compare.

In Maurer's protocol, assuming a circuit with a large number of multiplications (i.e. so that $n\cdot(n-1)\cdot\kappa \ll T$), the total number of bits sent over the network per multiplication is $n\cdot(n-1)\cdot\ell$.

In the protocol here, assuming a large field so that $\lceil\sigma/\ell\rceil = 1$, the cost is

$$n\cdot(n-t-1)\cdot\ell + 3\cdot n\cdot t\cdot\ell + 2\cdot n\cdot t\cdot\ell = n\cdot(n+4\cdot t-1)\cdot\ell$$

(where the authentication check is effectively 0). Since $t < \lceil\frac{n-1}{2}\rceil$, this cost is at most $3\cdot n\cdot(n-1)\cdot\ell$. Thus the cost of *actively*-secure multiplication with computational security is asymptotically just 3 times the cost of Maurer's passive multiplication with IT security.

Comparison with Chida et al. [CGH+18]

The protocol of Chida et al. [CGH+18] takes quite a different approach to obtaining actively-secure multiplication, bootstrapping the 2-round passive protocol for multiplication due to Damgård and Nielsen [DN07], discussed briefly in Section 2.5.3, to active security. The main point of their passive protocol was to create a scalable protocol: the overall communication cost is linear in the number of parties rather than quadratic as in protocols using Beaver's circuit randomization.

The preprocessing phase requires generating sharings of random secrets under both the original LSSS and its associated product LSSS, i.e. as $(\llbracket r\rrbracket, \llbracket r\rrbracket^{[2]})$ where r is uniformly random. To multiply $\llbracket x\rrbracket$ and $\llbracket y\rrbracket$ in the online phase, the parties locally compute the sharing $\llbracket x\cdot y\rrbracket^{[2]}$, open the secret $\llbracket x\cdot y\rrbracket^{[2]} - \llbracket r\rrbracket^{[2]}$ and compute $\llbracket x\cdot y\rrbracket := \llbracket r\rrbracket + (x\cdot y - r)$. For active security, message authentication codes (MACs) are maintained on every secret; in the online phase, every circuit operation is performed on both the secrets and the corresponding MACs. The preprocessing for this protocol also facilitates the use of randomness extraction described in Section 7.3. For non-threshold access structures, the techniques for randomness extraction in Section 7.3 can be used to improve on their efficiency.

While [CGH+18] scales better with the number of parties, there are several scenarios in which the protocols in this chapter might be preferable. Firstly, [CGH+18] requires the online phase to use a multiplicative LSSS, which for general $\mathcal{Q}_2$ access structures

may necessitate either doubling the number of shares (compared to the online phase of the protocol in this chapter) or using replicated secret-sharing. Secondly, the online phase of the protocol in this chapter involves a constant factor less communication over small fields since in [CGH+18], $\lceil \sigma / \log |\mathbb{F}| \rceil$ MACs are required for each secret and hence $1 + \lceil \sigma / \log |\mathbb{F}| \rceil$ passive multiplications are needed for one active multiplication; thus for situations in which parties outsource their preprocessing, the protocol from this chapter is better. It should be noted that the *overall* asymptotic costs are comparable with respect to σ since triple generation depends linearly on $\lceil \sigma / | \log \mathbb{F} | \rceil$. Thirdly, in high-latency networks, for example over wide-area networks (WANs), one would expect the online phase of [CGH+18] to be slower as each multiplication requires two rounds instead of one.

Chapter 8

Actively-Secure Mixed Protocol

This chapter is based on work in submission under the title MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security *[RW19] and was joint work with Dragoș Rotaru.*

This chapter Many efficient protocols exist for evaluating general Boolean [BDOZ11, NNOB12,HSS17,WRK17b,HOSS18a] and arithmetic [DPSZ12,DKL+13,KOS16,KPR18] circuits in the multi-party setting. Garbled circuits (GCs) are often used in situations where the round complexity is important, for example over a wide-area network (WAN), since they can be used to evaluate circuits in a constant number of rounds, but using them to evaluate arithmetic circuits is generally expensive. On the other hand, while linear secret-sharing scheme (LSSS)-based multi-party computation (MPC) for arithmetic circuits is efficient for "purely" arithmetic operations such as additions and multiplications, for more complex non-linear operations, the communication costs are significantly higher. In this chapter, a protocol is given that allows GCs and LSSS to be stitched together with active security, allowing one to choose how to evaluate different parts of a large "mixed" circuit.

8.1 Overview

The motivation for this chapter is that many real-world use cases of computing on private data involve a mixture of computations better-suited to Boolean circuits – such as the comparison of two integers – and computations better for arithmetic computations – such as evaluating statistical formulae.

One way of combining these types of computation involves using LSSS-based MPC over a finite field or ring to emulate arithmetic over the integers, and performing operations such as comparisons between secrets (i.e. $<, >, =$) using (what can be thought of as) a costly emulation of a Boolean computation. One of the shortcomings of LSSS-based MPC is that these natural but more involved procedures require special preprocessing and several rounds of communication.

Another way of combining the computation is to use (Boolean) GCs instead of secret-sharing for circuits involving many bit-wise operations. The disadvantage of using GCs for mixed computations is that performing general arithmetic computations in Boolean circuits can be expensive since addition and multiplication must be computed bit-wise. Garbled circuits will be explained in detail later, but for now the following intuition suffices: a GC is a randomized version of a circuit produced by a so-called *garbler* that hard-wires its own inputs; the circuit is handed to a so-called *evaluator*, that evaluates it on its inputs; finally, the garbler and evaluator engage in some procedure for decoding the final circuit output. The security guarantee is roughly the same as for LSSS-based MPC: that the garbler and evaluator should only learn the output and what can be inferred from the output and their own inputs. Garbling of arithmetic circuits is challenging and the best-known techniques [BMR16, Ben18] require $O(p)$ ciphertexts to be sent for every multiplication, where p is the field characteristic. Conversely, techniques for Boolean circuits have been known since the 1980s [Yao86, Oral presentation] and have seen significant improvements since that time, as outlined in Section 2.6. The circuit garbling described above is a two-party protocol. In multi-party garbling, every party acts as both the garbler and evaluator, a protocol for which was first given by Beaver et al. [BMR90]. In the recent past, Lindell et al. [LPSY15] showed how to garble with active security (efficiently) using MPC, after which followed much work with a similar approach [WRK17b, HSS17, KY18] so that multi-party garbling of Boolean circuits is considered efficient enough to be practical.

So-called *mixed protocols* are those in which parties switch between secret-sharing and garbled circuits mid-way through a computation, thus enjoying the efficiency of the basic addition and multiplication operations in any field using the former and the low-round complexity of GCs for non-linear subroutines using the latter. One can think of mixed protocols as allowing parties to choose the most efficient field in which to evaluate different parts of a circuit. For mixed protocols to be efficient, clearly the cost of switching between secret-sharing and garbling, performing the operation, and switching back must be more efficient than the method that does not require switching, per-

haps achieved by relegating some computation to the offline phase.

There is a rich literature of mixed protocols in the two-party setting with passive security – for example, [BPSW07, HKS+10, KSS14, BDK+18, IMZ19]. One of the more recent works is that of Demmler et al. [DSZ15], known as Arithmetic-Boolean-Yao (ABY), in which parties convert between arithmetic, Boolean, and "Yao" sharings. For small subcircuits, converting arithmetic shares of a secret to Boolean shares of the bit-decomposition of the same secret – without any garbling – suffices for efficiency gains over evaluating the same circuit without switching. However, for large subcircuits, using garbling additionally allows reducing online costs.

Mohassel and Rindal [MR18] constructed a three-party protocol known as ABY3 for mixing these three types of sharing in the active-security setting assuming at most one corruption. However, actively-secure mixed protocols in a general multi-party setting have hitherto not received much attention. One of the reasons for this is perhaps that there is a difficult technical challenge to overcome in ensuring that authentication of secrets is maintained through the conversion – that is, to ensure that the adversary cannot introduce errors during the switching procedure.

The goal of this chapter is to realize a **c**ircuit and **a**rithmetic **b**lack **b**ox (CABB), given by the functionality $\mathcal{F}_{\mathsf{CABB}}$ (which extends $\mathcal{F}_{\mathsf{ABB}}$) in Figure 8.1, which is an abstraction of a mixed protocol in the active-security setting. The solution uses MPC in a black-box way, and while the method used for garbling the circuit has some loose requirements that must be satisfied, it is compatible with many state-of-the-art multi-party Boolean circuit-garbling techniques; the only requirement is that parties should be able to authenticate their own choices of inputs, and that XOR can be computed between authenticated bits. (This is discussed in detail later.) As the garbling uses MPC as a black box, being executed in the $\mathcal{F}_{\mathsf{RPrep}}$-hybrid model, no restriction need be made on the access structure, making it compatible with the recent compilers for general access structures [ABF+18, ACK+19].

Functionality $\mathcal{F}_{\mathsf{CABB}}$

This functionality extends $\mathcal{F}_{\mathsf{ABB}}$ in Figure 2.14. The function $\texttt{BitDec} : \mathbb{F} \to \{0,1\}^{\lfloor \log |\mathbb{F}| \rfloor}$ takes an element of $\mathbb{F}$ and computes its bit-decomposition as a sequence of bits.

Evaluate Circuit On input $(\texttt{EvaluateCircuit}, C, (id_k)_{k=1}^{t}, id, sid)$ where $t \cdot \log|\texttt{DB.Field}|$ is the arity of Boolean circuit C, if $id_i \in \texttt{DB.Ids}$ for all $i \in [t]$, then await a message OK or Abort from the adversary. If the message is OK, then set $\texttt{DB}[id] := C(\texttt{BitDec}(\texttt{DB}[id_1]), \ldots, \texttt{BitDec}(\texttt{DB}[id_t]))$ and continue; otherwise, send the message Abort to all parties, and then halt.

Figure 8.1: Circuit and Arithmetic Black Box Functionality, $\mathcal{F}_{\mathsf{CABB}}$.

8.1.1 Switching Mechanism

The exposition in this chapter is focused mainly on the full-threshold setting, and hence authentication is achieved using information-theoretic (IT) message authentication codes (MACs) as described in Section 2.5.3, but since $\mathcal{F}_{\mathsf{RPrep}}$ is used as a black box, any access structure can be used as long as $\mathcal{F}_{\mathsf{RPrep}}$ can be realized.

The key challenge for mixed protocols in the active-security setting is that secrets must remain authenticated through the conversion from LSSS to GC and back again. In the full-threshold setting, the naïve way of maintaining authentication from LSSS to GC is for parties to bit-decompose the shares of their secrets and the MACs locally and use these as input bits to the circuit, and checking the MAC inside the GC. This necessitates a considerable amount of online communication since each party needs to broadcast a pseudorandom function (PRF) key for each bit in the bit-decomposition of each share of each secret and its MAC. This method also requires garbling several additions and multiplications inside the circuit to check the MAC, and some methodology for obtaining secret-shared output (in the arithmetic field) and corresponding MACs (perhaps by requiring random masks to be provided as auxiliary input to the circuit). The advantage of this solution, despite these challenges, is that it requires no additional preprocessing, nor adaptations to the garbling procedure.

Contrasting this approach, the idea in this chapter is to make use of special preprocessing to speed up the conversion in the online phase of the protocol. This preprocessing takes the form of "**d**oubly-shared" **a**uthenticated **bits**, dubbed *daBits*, following the nomenclature set out in [NNOB12]. These doubly-shared secrets are values in $\{0,1\}$ shared and authenticated both in $\mathbb{F}_p$ and $\mathbb{F}_{2^l}$, where 0 and 1 here denote the additive and multiplicative identities, respectively, in each field. The idea is to use these daBits to convert authenticated, secret-shared data in $\mathbb{F}_p$, where p is a large prime, to authen-

ticated secret-shared data in $\mathbb{F}_{2^l}$.

Remark 8.1. The parameter l is not directly related to $\ell := \lfloor \log p \rfloor$ since the bits of an element in $\mathbb{F}_p$ will be translated into a *set* of secret-shared bits in $\mathbb{F}_{2^l}$, not a single $\mathbb{F}_{2^l}$ field element. The parameters are chosen so that $\ell = O(\sigma)$ and $l = O(\kappa)$ in order that the MPC protocol is secure with statistical security σ and the circuit garbling is secure with computational security κ.

The switching procedure of a secret-shared element x in $\mathbb{F}_p$ to a set of input bits to a garbled circuit involves constructing a random secret r in $\mathbb{F}_p$ using daBits, opening $x - r$, bit-decomposing this public value (requiring no communication) and using these as input bits for a GC. Then the parties can evaluate the garbled circuit locally, where the circuit has a prepended subcircuit that adds r and computes the result modulo p. This general idea for conversion has been used in many of the prior works in the two-party semi-honest security setting. Garbling of the subcircuit is achieved using the bit decomposition of r with the parts of the daBits in $\mathbb{F}_{2^l}$. It will be shown in Section 8.4 that retrieving outputs from the circuit in secret-shared form can be done by local operations, whence they can proceed with further LSSS-based computations on secrets in $\mathbb{F}_p$. This method keeps the authentication check mostly outside of the circuit, instead requiring that the MAC on $x - r$ be correct. In fact, this approach is oblivious to the method used for authenticating, which means that daBits can be generated using any MPC protocol that offers authentication, such as the protocols in Chapters 6 and 7. This process of mixing arithmetic and Boolean circuits by facilitating switching between secret-sharing and garbled circuits is dubbed "circuit MArBling", from **M**ixing **Ar**ithmetic and **B**oolean circuits.

The rough structure is shown in Figure 8.2, where the Boolean circuit C is evaluated on inputs x and y that are initially secret-shared in $\mathbb{F}_p$, and the output is a set of secret-shared bits in $\mathbb{F}_p$ representing the bit-decomposition of an element of $\mathbb{F}_p$. The procedure called "Mask conversion" (explained in the following sections) is a local operation in the online phase. The notation $(a - b)_j$ is used to denote the j^{th} bit in the binary expansion of the integer $a - b$.

The only use of doubly-shared masks is at the two boundaries between a garbled circuit and secret-shared data (i.e. circuit input and output): all secrets used in evaluating arithmetic circuits (i.e. using standard LSSS-based MPC) are authenticated shares in $\mathbb{F}_p$ only. All other secrets "inside" the circuit (that is, for all wires that are *not* circuit input or output wires) are authenticated shares of bits in $\mathbb{F}_{2^l}$ only. The *online* communication cost is that of each party broadcasting a single element of $\mathbb{F}_p$ and then broad-

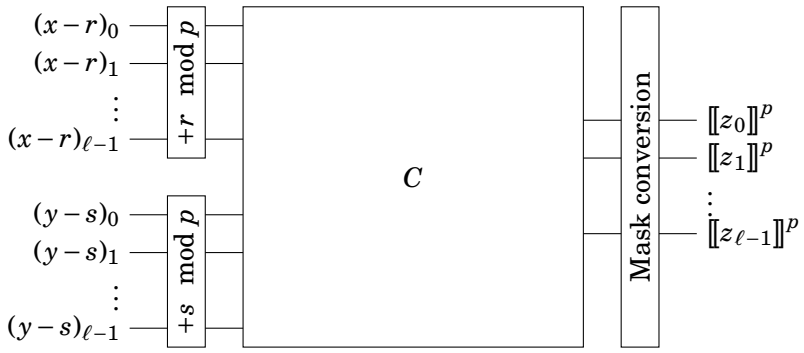


Figure 8.2: Conversion Overview.

casting $\log p$ PRF keys (of length κ) per input, for a circuit of any depth. Thus the online cost is $O(\kappa \cdot \log p)$ bits per party, per $\mathbb{F}_p$ input to the Boolean circuit.

Remark 8.2. While essentially all of the basic actively-secure MPC protocols enable the evaluation of additions and multiplications, for more complicated non-linear functions the only solutions that exist are those that require additional assumptions on the input data. For example, comparison requires bit decomposition, which itself requires that secrets be bounded by some constant. Since the bits of each input are directly inserted into the circuit, this additional assumption can be avoided.

8.1.2 Structure

The realization of the functionality $\mathcal{F}_{\mathsf{CABB}}$ is achieved in the following way:

- In the preprecessing phase:
  1. The MPC functionality $\mathcal{F}_{\mathsf{RPrep}}$ is extended to the functionality $\mathcal{F}_{\mathsf{RPrep+}}$ given in Figure 8.3 to allow the same bits to be generated in two independent $\mathcal{F}_{\mathsf{RPrep}}$ sessions in two different fields, $\mathbb{F}_p$ and $\mathbb{F}_{2^l}$.
  2. The $\mathbb{F}_{2^l}$ instance of $\mathcal{F}_{\mathsf{RPrep}}$ inside $\mathcal{F}_{\mathsf{RPrep+}}$ is used to perform a form of garbling known as SPDZ-BMR-style garbling [LPSY15, KY18].
- In the online phase:
  - The protocol Π_{Online} is used with the instance of $\mathcal{F}_{\mathsf{RPrep}}$ over $\mathbb{F}_p$ to perform LSSS-based MPC over a prime field, realizing the $\mathcal{F}_{\mathsf{ABB}}$ part of $\mathcal{F}_{\mathsf{CABB}}$.
  - The protocol $\Pi_{\mathsf{BMREvaluate}}$ to used to evaluate circuits garbled as preprocessing, realizing the circuit-evaluation part of $\mathcal{F}_{\mathsf{CABB}}$.

– A switching procedure is used to convert between secret-sharing and garbled circuits and back again.

Functionality $\mathcal{F}_{\mathsf{RPrep+}}$

This functionality extends the reactive functionality $\mathcal{F}_{\mathsf{RPrep}}$ with commands to generate the same bits in two independent sessions.

Instances of $\mathcal{F}_{\mathsf{RPrep}}$

Two independent copies of $\mathcal{F}_{\mathsf{RPrep}}$ are identified via session identifiers sid_p and sid_{2^l}.

Additional command

daBits On receiving $(\texttt{daBits}, (id_k)_{k=1}^t, sid_p, sid_{2^l})$, from all parties where $\{id_k\}_{k=1}^t$ are fresh identifiers,

1. Sample a set $\{b_k\}_{k=1}^t \leftarrow \mathcal{U}(\{0,1\})$.
2. Execute the macro $\texttt{Sample}(id_{b_k})$ from each instance of $\mathcal{F}_{\mathsf{RPrep}}$ with $\mathcal{S}$ for each $k \in [t]$.
3. Await a message Abort or OK from $\mathcal{S}$. If the message is OK and `Abort` is false, then for all $i \in [n] \setminus A$, send $\{[\![b_k]\!]^p_{\mathcal{P}_i}, [\![b_k]\!]^{2^l}_{\mathcal{P}_i}\}_{k=1}^t$ to $\mathcal{P}_i$; otherwise, send the message Abort to all honest parties, and then halt.

Figure 8.3: Functionality for Two MPC Engines, with daBits in Both, $\mathcal{F}_{\mathsf{RPrep+}}$.

A summary of the dependencies is given in Figure 8.4.

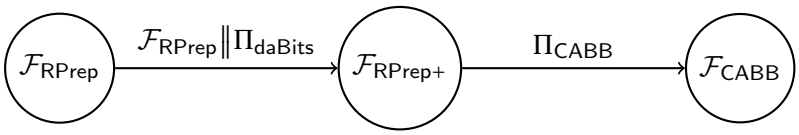


Figure 8.4: Conversion Protocol Dependencies.

8.2 Preliminaries

This section provides an overview of the MPC techniques important for this chapter, then gives a brief recapitulation of the protocol known as SPDZ-BMR developed by Lindell et al. [LPSY15] for computing a multi-party garbled circuit with active security, followed by an overview of the conversion technique.

Throughout this chapter, it will be assumed that the parties are connected in a complete synchronous network of secure channels, i.e. $\mathsf{SC}(E_{\mathsf{BC}})$, and have access to a broadcast channel, which can be instantiated in the random oracle model over this set of

secure channels as described in Section 2.3.4 since the MPC and GC protocols here only offer security with abort.

8.2.1 MPC

The protocol will make use of MPC as a black box; it involves reactive computation so the functionality $\mathcal{F}_{\mathsf{RPrep}}$ from Section 4.6 is used (rather than $\mathcal{F}_{\mathsf{ABB}}$).

For the protocols that realize $\mathcal{F}_{\mathsf{CABB}}$, it is not important how authentication is achieved – i.e. whether by linear MACs so that $[\![\cdot]\!] = ([\![\cdot]\!]^{\mathtt{A}}, [\![\gamma(\cdot)]\!]^{\mathtt{A}})$ as in SPDZ, or by error-detection properties as for $\mathscr{Q}_2$ access structures – the key point is that secrets can be opened with authentication. Shares are denoted in the following three ways:

$$\begin{aligned}\text{Sharing in } \mathbb{F}_p\text{: } & [\![a]\!]^p \\ \text{Sharing in } \mathbb{F}_{2^l}\text{: } & [\![c]\!]^{2^l} \\ \text{Sharing in both: } & [\![b]\!]^{p,2^l} = ([\![b]\!]^p, [\![b]\!]^{2^l}) \text{ where } b \in \{0,1\}.\end{aligned}$$

The sharing $[\![b]\!]^{p,2^l}$, called a daBit, is considered correct if the bit is the same in both fields – that is, either both the additive identity or both the multiplicative identity, in their fields. Creating daBits efficiently is one of the main tasks of this chapter.

Conditions on the secret-sharing field

Let $\ell := \lfloor \log p \rfloor$. Throughout, MPC is executed in $\mathbb{F}_p$ where p is some large prime, but the conversion protocol is only secure if it is possible to generate uniformly random field elements by sampling bits uniformly at random $\{[\![r_j]\!]^p\}_{j=0}^{\ell-1}$ and summing them to get $[\![r]\!]^p := \sum_{j=0}^{\ell-1} 2^j \cdot [\![r_j]\!]^p$. This requires that $1 - \frac{2^\ell}{p} = O(2^{-\sigma})$, which roughly speaking says that p is only slightly larger than a power of 2. (By symmetry of this argument one can require that p just be close (above or below) to a power of 2.) Recall that sampling a uniform element of $\{0,1\}^\ell$ produces the same distribution as sampling ℓ bits independently by standard Measure Theory. It follows from Lemma 8.1 that under this assumption on p, the statistical distance between the uniform distribution over $\mathbb{F}_p$ and the same over $\{0,1\}^\ell$ is negligible.

Lemma 8.1. *Let $\ell = \lfloor \log p \rfloor$, let D be the probability mass function for the uniform distribution $\mathcal{D}$ over $[0,p) \cap \mathbb{Z}$ and let E be the probability mass function for the uniform distribution $\mathcal{E}$ over $[0,2^\ell) \cap \mathbb{Z}$. Then the statistical distance between distributions is negligible in the security parameter if $1 - \frac{2^\ell}{p} = O(2^{-\sigma})$.*

Proof. By definition of statistical distance (see Definition 2.2),

$$\Delta(\mathcal{D},\mathcal{E}) = \frac{1}{2}\cdot\sum_{x=0}^{p-1}|D(x)-E(x)| = \frac{1}{2}\cdot\sum_{x=0}^{2^\ell-1}\left|\frac{1}{p}-\frac{1}{2^\ell}\right| + \frac{1}{2}\cdot\sum_{x=2^\ell}^{p-1}\left|\frac{1}{p}-0\right|$$
$$= \frac{1}{2}\cdot 2^\ell\cdot\frac{p-2^\ell}{p\cdot 2^\ell} + \frac{1}{2}\cdot(p-2^\ell)\cdot\frac{1}{p} = 1-\tfrac{2^\ell}{p} = O(2^{-\sigma}).$$

□

Note on XOR

The protocol makes heavy use of the (generalized) XOR operation. This can be defined in any field as the function

$$f:\mathbb{F}_p\times\mathbb{F}_p\to\mathbb{F}_p$$
$$(x,y)\mapsto x+y-2\cdot x\cdot y, \tag{8.1}$$

which coincides with the usual XOR function for fields of characteristic 2. In LSSS-based MPC, addition requires no communication, so computing XOR in $\mathbb{F}_{2^l}$ is for free; the cost in $\mathbb{F}_p$ (char$(p)>2$) is one multiplication, which requires a Beaver triple and some communication. This operation is the main cost associated with the preprocessing phase, since generating daBits with active security requires generating lots of them and then computing several XORs in both fields.

Agreement of Random Strings

The protocols require algorithms that provide the parties with specific random strings, shown in Figure 8.5. Such random strings can be agreed in the $\mathcal{F}_{\mathsf{CoinFlip}}$-hybrid model using $\mathcal{F}_{\mathsf{CoinFlip}}$ to generate seeds and then using deterministic algorithms locally to compute the necessary shared data, as demonstrated by Π_{Rand} in Figure 8.6. Unfortunately, a clean description of $\mathcal{F}_{\mathsf{Rand}}$ that simply samples uniformly from the required sets is not possible for a technical reason, as discussed in Section 6.3.2. It is, however, easy to get around this by defining the functionality to sample a seed uniformly and to use this to generate the random string deterministically itself. There is no communication in the protocol beyond the call to $\mathcal{F}_{\mathsf{CoinFlip}}$ for a seed, so the simulation in the $\mathcal{F}_{\mathsf{CoinFlip}}$-hybrid world is trivial: the $\mathcal{S}$ replaces the seed provided as output by its local instance of $\mathcal{F}_{\mathsf{CoinFlip}}$ with the seed sampled and output by $\mathcal{F}_{\mathsf{Rand}}$.

Functionality $\mathcal{F}_{\mathsf{Rand}}$

Initialize On input $(\texttt{Initialize}, sid)$ from all honest parties and $\mathcal{S}$, await further messages.

Random subset On input $(\texttt{RSubset}, X, t, sid)$ where X is a set and t satisfies $t \leq |X|$,

1. Sample a seed $seed \leftarrow \mathcal{U}(\{0,1\}^\kappa)$.
2. Set $\pi := \texttt{Shuffle}(seed, |X|)$.
3. Let $X = \{x_i\}_{i=1}^{|X|}$ and set $S := \{x_{\pi(i)} : 1 \leq i \leq t\}$.
4. Send S to all honest parties and $\mathcal{S}$, and additionally send $seed$ to $\mathcal{S}$.

Random buckets On input $(\texttt{RBuckets}, X, t, sid)$ where X is a set and $t \in \mathbb{N}$ such that $|X|/t \in \mathbb{N}$, do the following:

1. Sample a seed $seed \leftarrow \mathcal{U}(\{0,1\}^\kappa)$.
2. Set $\pi := \texttt{Shuffle}(seed, |X|)$.
3. Let $X = \{x_i\}_{i=1}^{|X|}$ and for each $i = 1$ to $|X|/t$, set $X_i := \{x_{\pi(j)} : (i-1) \cdot t < j \leq i \cdot t\}$.
4. Send $(X_i)_{i=1}^{|X|/t}$ to all honest parties and $\mathcal{S}$, and additionally send $seed$ to $\mathcal{S}$.

Figure 8.5: Random Sampling Functionality, $\mathcal{F}_{\mathsf{Rand}}$.

Protocol Π_{Rand}

This protocol is realized in the $\mathcal{F}_{\mathsf{CoinFlip}}$-hybrid model. Let $\texttt{Shuffle}()$ be the Knuth Shuffle in Figure 2.1.

Initialize Parties agree on a session identifier sid and computational security parameter κ, and call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{Initialize}, \{0,1\}^\kappa, sid)$.

Random Subset To compute a random subset of size t of a set X, parties do the following:

1. Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid)$ to obtain a seed $seed$.
2. Set $\pi := \texttt{Shuffle}(seed, |X|)$.
3. Let $X = \{x_i\}_{i=1}^{|X|}$ and set $S := \{x_{\pi(i)} : 1 \leq i \leq t\}$.

Random Buckets To put a set of items indexed by a set X into buckets of size $t \in \mathbb{N}$ where $|X|/t \in \mathbb{N}$, the parties do the following:

1. Call $\mathcal{F}_{\mathsf{CoinFlip}}$ with input $(\texttt{RElt}, sid)$ to obtain a seed $seed$.
2. Set $\pi := \texttt{Shuffle}(seed, |X|)$.
3. Let $X = \{x_i\}_{i=1}^{|X|}$ and for each $i = 1$ to $|X|/t$, set $X_i := \{x_{\pi(j)} : (i-1) \cdot t < j \leq i \cdot t\}$.

Figure 8.6: Random Sampling Protocol, Π_{Rand}.

Cut-and-choose

A process known as *cut-and-choose* is a way of one party proving a statement to another party by generating a large number of randomized instances of a problem and revealing

the secrets used to generate a large random subset so that with overwhelming probability in the statistical security parameter, the remaining secret instances that were not revealed are correct. Its general definition is not particularly important for this chapter as the specific probabilities are calculated on an *ad hoc* basis in the following sections, but it should be noted that it is a standard technique in cryptography.

8.2.2 Garbled Circuits

In this section, protocols for multi-party garbling are explained. The focus of this chapter is on showing how to use special preprocessing to make mixed protocols efficient, and consequently, the garbling methods described here are intended only to show the use of daBits *in situ* – there is no claim of novelty in the garbling methods presented here. Indeed, there are many optimizations to the outline given that are not discussed since the precise garbling techniques are not important and they complicate exposition.

The garbling protocol presented is due to Keller and Yanai [KY18], which can be thought of as a variant of SPDZ-BMR, as it is the easiest to explain from the point of using MPC as a black box.

Garbling as an MPC Protocol

An outline of circuit garbling in the two-party setting was given in Section 8.1. The concrete process, which only offers passive security, is now given. It is assumed that the garbler and evaluator have already agreed on a Boolean circuit they wish to evaluate on the union of their secret inputs.

To garble the circuit, first the garbler samples a *global difference* Δ, and then for each wire u in the circuit samples a "zero key" $k_{u,0}$ and sets the corresponding "one key" as $k_{u,1} \coloneqq k_{u,0} \oplus \Delta$. The keys can then be written as $k_{u,b} = k_{u,0} \oplus b \cdot \Delta$ for $b \in \{0,1\}$. (The reason for this setup is explained below, and is part of the *FreeXOR* technique.) A wire connection exiting one gate and entering another is considered one wire, as are all circuit input and output wires. A *masking* (or *permutation*) bit λ_u is also sampled for each wire, the reason for which will be explained shortly. Then the garbler converts each Boolean fan-in-two gate g, where $g : \{0,1\}^2 \to \{0,1\}$, with input wires u and v and output wire w to a set of 4 ciphertexts as shown in Table 8.1, where Enc is an encryption algorithm that takes two symmetric keys and a message as input. The details of the encryption scheme are explained later.

u	v	Ciphertexts
0	0	$\mathsf{Enc}_{k_{u,0},k_{v,0}}\left(k_{w,0} \oplus \Delta \cdot (g(0 \oplus \lambda_u, 0 \oplus \lambda_v) \oplus \lambda_w) \middle\| (g(0 \oplus \lambda_u, 0 \oplus \lambda_v) \oplus \lambda_w)\right)$
0	1	$\mathsf{Enc}_{k_{u,0},k_{v,1}}\left(k_{w,0} \oplus \Delta \cdot (g(0 \oplus \lambda_u, 1 \oplus \lambda_v) \oplus \lambda_w) \middle\| (g(0 \oplus \lambda_u, 1 \oplus \lambda_v) \oplus \lambda_w)\right)$
1	0	$\mathsf{Enc}_{k_{u,1},k_{v,0}}\left(k_{w,0} \oplus \Delta \cdot (g(1 \oplus \lambda_u, 0 \oplus \lambda_v) \oplus \lambda_w) \middle\| (g(1 \oplus \lambda_u, 0 \oplus \lambda_v) \oplus \lambda_w)\right)$
1	1	$\mathsf{Enc}_{k_{u,1},k_{v,1}}\left(k_{w,0} \oplus \Delta \cdot (g(1 \oplus \lambda_u, 1 \oplus \lambda_v) \oplus \lambda_w) \middle\| (g(1 \oplus \lambda_u, 1 \oplus \lambda_v) \oplus \lambda_w)\right)$

Table 8.1: Yao's garbled gate with FreeXOR and Point-And-Permute.

From the table, observe that the gate is transformed into a set of four ciphertexts, each encrypting one of two output keys concatenated with a bit called a *signal bit*, computed as a function of masking bits and gate input bits.

Once every gate has been garbled, the garbler sends all ciphertexts to the evaluator, and for every wire u corresponding to one of its own inputs it sends a key and a signal bit $\Lambda_u \coloneqq u \oplus \lambda_u$ to the evaluator:

$$(k_{u,u\oplus\lambda_u} \| u \oplus \lambda_u)$$

where $u \in \{0,1\}$ is the garbler's input on wire u. (Note that in practice the garbler can just hardwire its inputs instead of sending ciphertexts and keys, but the exposition is clearer if this is not assumed.)

In the protocol known as oblivious transfer (OT), a sender sends two messages and a receiver chooses to receive one; the security guarantee of the protocol is that the sender does not learn which message was selected and the receiver does not learn anything about the message it did not select. This process is used to transfer keys for circuit input wires that correspond to the evaluator's inputs from the garbler to the evaluator: specifically, the garbler provides as input to the OT protocol the messages

$$(k_{v,0\oplus\lambda_v} \| 0 \oplus \lambda_v)$$
$$(k_{v,1\oplus\lambda_v} \| 1 \oplus \lambda_v)$$

where λ_v is known only to the garbler (but can be immediately deduced by the evaluator using its own input). The evaluator selects the first if its input is $v = 0$, or the second if its input is $v = 1$. Thus the evaluator obtains

$$(k_{v,v\oplus\lambda_v}, v \oplus \lambda_v)$$

which corresponds to the key k_{v,Λ_v} with signal bit $\Lambda_v = v \oplus \lambda_v$, and learns nothing about the other key by the security of the OT protocol.

Evaluation proceeds in the following way. Given a gate, signal bits Λ_u and Λ_v and corresponding keys k_{u,Λ_u} and k_{v,Λ_v}, the evaluator decrypts the ciphertext corresponding to the row (Λ_u, Λ_v), i.e. the ciphertext

$$\mathsf{Enc}_{k_{u,\Lambda_u},k_{v,\Lambda_v}}(k_{w,0} \oplus \Delta \cdot (g(\Lambda_u \oplus \lambda_u, \Lambda_v \oplus \lambda_v) \oplus \lambda_w) \| (g(\Lambda_u \oplus \lambda_u, \Lambda_v \oplus \lambda_v) \oplus \lambda_w))$$

which returns the key $k_{w,0} \oplus \Delta \cdot (g(u,v) \oplus \lambda_w)$ since $\Lambda_u \oplus \lambda_u = u$ and $\Lambda_v \oplus \lambda_v = v$, along with the bit $(g(u,v) \oplus \lambda_w)$. Now since $w = g(u,v)$, it holds that $g(u,v) \oplus \lambda_w = w \oplus \lambda_w = \Lambda_w$ and so the key learnt is k_{w,Λ_w} and the bit is Λ_w. The evaluator now uses this key (and the output key of another gate) to decrypt the next gate. The evaluator proceeds iteratively through the circuit and obtains a key (or multiple keys) as final output. The evaluator and garbler then interact to obtain the output to which the circuit output keys correspond.

Point-and-Permute The technique of using masking bits as described above is called *point-and-permute* and was introduced by Beaver et al. [BMR90]. The purpose of the masking bits is so that the evaluator does not learn partial evaluations of the circuit, which may otherwise leak information about the garbler's input. Since the masks are unknown and uniform, the signal bits leak nothing about the partial evaluation. Note that it is only necessary to hide *internal* circuit wire values from the evaluator, so masking bits on circuit input wires can be set to 0.

The original technique for determining the output key while keeping the internal circuit wire secret was to use some sort of authenticated encryption and for the evaluator to permute the ciphertexts arbitrarily, so that the garbler had to decrypt all four ciphertexts and output whichever was a valid plaintext; using point-and-permute, the garbler only needs to decrypt one ciphertext per gate.

FreeXOR The reason for choosing keys to differ by the global difference Δ is that it leads to an efficient way to garble XOR gates: instead of choosing an independent output wire key $k_{w,0}$ and mask λ_w for XOR gates, the key can be set to $k_{w,0} \coloneqq k_{u,0} \oplus k_{v,0}$ and $\lambda_w \coloneqq \lambda_u \oplus \lambda_v$. Then because the (XOR) difference between every zero/one key pair in the circuit is the same (the global difference, Δ), there is no need to send ciphertexts for

XOR gates: the evaluator computes $\Lambda_w := \Lambda_u \oplus \Lambda_v$ and

$$\begin{aligned} k_{u,\Lambda_u} \oplus k_{v,\Lambda_v} &= k_{u,0} \oplus \Delta \cdot (v \oplus \lambda_v) \oplus k_{v,0} \oplus \Delta \cdot (v \oplus \lambda_v) \\ &= k_{u,0} \oplus k_{v,0} \oplus \Delta \cdot (u \oplus v \oplus \lambda_u \oplus \lambda_v) \\ &= k_{w,0} \oplus \Delta \cdot (w \oplus \lambda_w) \\ &= k_{w,\Lambda_w}. \end{aligned}$$

This technique was introduced by Schneider and Kolesnikov [KS08] and is known as FreeXOR. It requires additional assumptions on the encryption scheme, discussed below when the encryption scheme is defined.

Multi-Party Garbling Overview

In multi-party garbling, every party acts as both garbler and evaluator. Each party generates a circuit for which it knows all the wire keys, but where each ciphertext is encrypted under *all* parties' corresponding wire keys. This means that every party must evaluate all n circuits in parallel to decrypt each subsequent gate and so learn all n succeeding keys.

Lindell and Pinkas [LP07] showed how to use cut-and-choose to obtain active security. The orthogonal approach of SPDZ-BMR garbling is to force the parties to act honestly when garbling by using actively-secure MPC to compute the ciphertexts. Lindell et al. [LPSY15] gave a generic multi-party method, known as SPDZ-BMR, for garbling in a constant number of rounds with active security where the preprocessed material is obtained from SPDZ [DPSZ12]. Their method is (roughly) to execute the classic [BMR90] multi-party garbling protocol using SPDZ to generate all the necessary secrets (such as random bits and keys) and to compute the ciphertexts. While the SPDZ-BMR protocol garbles Boolean circuits, the wire masks are arithmetic shares in $\mathbb{F}_p$ of binary values, and the wire keys are random elements of $\mathbb{F}_p$ secret-shared amongst the parties. Importantly, it was shown that it is not necessary for parties to provide zero-knowledge proofs that the evaluations of the PRF used for encryption, which are computed locally by each party, are done honestly, as the evaluators will abort with overwhelming probability if parties cheat in this way.

The FreeXOR garbling technique crucially relies on the fact that the keys are elements of a field of characteristic 2. Towards the goal of a multi-party garbling protocol with FreeXOR, one might hope to perform SPDZ-BMR over $\mathbb{F}_{2^l}$. One of the reasons this was not considered for SPDZ-BMR was (presumably) that the SPDZ offline phase

was much faster for large prime fields than extension fields. Indeed, the most efficient variant of SPDZ used BGV [BGV12] as the somewhat-homomorphic encryption (SHE) scheme, which means that while the preprocessing phase can be parallelized through ciphertext packing, for large extension fields – and in particular for finite extensions of $\mathbb{F}_2$ – the amount of packing is severely limited. However, shortly after this Keller et al. [KOS16] showed how to use OT to perform the offline phase even more efficiently. This solution was shown to be more efficent than using SHE for extension fields. Despite recent work [KPR18] showing that SHE solutions outperform OT solutions for large prime fields, [KOS16] remains faster over extension fields. Subsequently, Keller and Yanai [KY18] showed how to apply FreeXOR in the multi-party setting using SPDZ-BMR-style garbling where the SPDZ shares are in $\mathbb{F}_{2^l}$ instead of $\mathbb{F}_p$.

Meanwhile, Hazay et al. [HSS17] also showed how to obtain FreeXOR in the multi-party setting, again over $\mathbb{F}_{2^l}$, but take a different approach from SPDZ-BMR: they do not make use of a a full-blown MPC functionality and instead produce an *unauthenticated* garbled circuit – it is merely additively shared, whereas in SPDZ-BMR and [KY18], the garbled circuit is authenticated with MACs. Active security comes from the fact that an incorrectly-garbled circuit will only cause the parties to abort when evaluating it. This approach requires only a single (authenticated) $\mathbb{F}_2$ multiplication per AND gate.

The implementation of the protocols in this chapter[1] uses the multi-party Boolean circuit garbling protocol due to Keller and Yanai [KY18], which is less efficient than [HSS17] and [WRK17b], but the implementation [Kel19] was easier to integrate with the SPDZ compiler to be able to switch between different online phases of an MPC program [ABF+18]. Despite this there is good reason to believe that the generation of the specialized preprocessing required in the solution here dovetails with most if not all of these alternative these garbling schemes as the only requirements are the following:

- Parties should be able to authenticate their own secret inputs (in fact, secret bits suffices), for whatever authentication method is used in the protocol.
- Parties should be able to compute the XOR of authenticated bits.

Unfortunately, the authentication is usually abstracted away in garbling functionalities so one cannot make straightforward claims about using garbling in a black-box way.

[1] The implementation was done by the coauthor of this work.

Encryption

So far, discussion of encryption has been abstract; in this section the definition as used in the SPDZ-BMR protocol is given. Messages are encrypted by computing the XOR of the message with a pseudorandom one-time-pad generated by a PRF under keys held by multiple parties as described below. In detail, the encryption of a message $m \in \{0,1\}^\kappa$ under keys $k_u := k_u^1 \| \cdots \| k_u^n$ and $k_v := k_v^1 \| \cdots \| k_v^n$ and nonce r, where $\mathcal{P}_i$ holds they keys $k_u^i, k_v^i \in \{0,1\}^\kappa$, is defined as

$$\mathsf{Enc}_{k_u,k_v}(m;r) := \left(\bigoplus_{j=1}^{n} F_{k_u^j,k_v^j}(r) \right) \oplus m$$

where

$$F_{k_a,k_b}(\cdot) := F_{k_a}(\cdot) \oplus F_{k_b}(\cdot).$$

For clarity, the formula will be explicitly in the circuit garbling rather than abstract to the encryption notation. Use of the PRF for encryption as above, and the integration of the FreeXOR technique to circuit garbling, require specific assumptions on the PRF, as is discussed in [CKKZ12] and [HSS17]. Analysis of the security of this encryption scheme is omitted as the details of the garbling are not the focus of this chapter.

Multi-Party Garbled Gate Using this encryption scheme, a garbled gate is defined by the parties computing the following $4 \cdot n$ ciphertexts, indexed by $(\alpha,\beta) \in \{0,1\}^2$ and $j \in [n]$:

$$[\![\tilde{g}^j_{\alpha,\beta}]\!]^{2^l} := \left(\bigoplus_{i=1}^{n} [\![F_{k^i_{u,\alpha},k^i_{v,\beta}}(id_g,j)]\!]^{2^l} \right) \oplus [\![k^j_{w,0}]\!]^{2^l} \oplus [\![\Delta^j]\!]^{2^l} \cdot \left((\alpha \oplus [\![\lambda_u]\!]^{2^l}) \cdot (\beta \oplus [\![\lambda_v]\!]^{2^l}) \oplus [\![\lambda_w]\!]^{2^l} \right)$$

where id_g denotes the gate index and Δ^j is the global difference of $\mathcal{P}_j$. Each PRF evaluation $F_{k^i_{u,\alpha},k^i_{v,\beta}}(id_g,j)$ is evaluated by party $\mathcal{P}_i$ and then provided as input to the MPC engine by calling $\mathcal{F}_{\mathsf{RPrep}}$ with input

$$(\texttt{Input}, i, id_{F^i_{\alpha,\beta}}, F_{k^i_{u,\alpha},k^i_{v,\beta}}(id_g,j), sid).$$

The masking bits are generated as random bits using $\mathcal{F}_{\mathsf{RPrep}}$ so no party knows them. Thus each gate requires $4 \cdot n$ MPC multiplications:

$$[\![\Delta^j]\!]^{2^l} \cdot [\![\lambda_u]\!]^{2^l}, \quad [\![\Delta^j]\!]^{2^l} \cdot [\![\lambda_v]\!]^{2^l}, \quad [\![\Delta^j]\!]^{2^l} \cdot [\![\lambda_w]\!]^{2^l}, \quad ([\![\Delta^j]\!]^{2^l} \cdot [\![\lambda_u]\!]^{2^l}) \cdot [\![\lambda_v]\!]^{2^l}.$$

and n^2 inputs for the PRF evaluations.

Once these ciphertexts have been computed, they are revealed to all parties, and so each party $\mathcal{P}_i$ holds $\{\widetilde{g}^j_{\alpha,\beta} : (\alpha,\beta) \in \{0,1\}^2,\ j \in [n]\}$. The full protocol is given in Figures 8.9, 8.10 and 8.11 but it is modified to allow for the use of the special preprocessing and for parties to obtain secret-shared output, so it is best read in the context of later sections.

Note that since the keys for the PRF are in the field $\mathbb{F}_{2^l}$ in the garbling protocol, the instance of $\mathcal{F}_{\mathsf{RPrep}}$ must be over a field with $l = \Omega(\kappa)$ for computational security.

Output

In the usual BMR protocol, immediately after garbling, the parties open the masks for output wires. This enables all parties to view the output bits. Instead, in the conversion protocol later, the parties will obtain the final output in secret-shared form, which they can do simply by not opening the output masks, and by computing (locally) the XOR of the public signal bit with the output mask. That is, an output bit is shared as

$$[\![b]\!]^{2^l} := [\![\lambda_w]\!]^{2^l} \oplus \Lambda_w.$$

8.3 Generation of daBits

In this section, the functionality $\mathcal{F}_{\mathsf{RPrep+}}$ is constructed.

Any technique for generating daBits using MPC as a black box requires some form of checking procedure to ensure consistency between the two fields. One common method of checking consistency between two sets of the (alleged) same secrets involves checking that random linear combinations of secrets produce the same result in both cases. Unfortunately, since the two instantiations of MPC are in different fields, one cannot compute the same linear combination in both fields. It is, however, possible to check XORs of bits, which in $\mathbb{F}_p$ is non-linear and so requires multiplication. (See Equation 8.1 in Section 8.2.) Minimizing communication costs here therefore means minimizing the number of multiplications. Techniques using OT such as [WRK17b] to generate authenticated bits require a lot of XORs for checking correctness, so are undesirable for generating daBits.

The protocol for generating daBits is summarized as follows. In order to generate the same bit in both fields, each party samples a bit and calls the $\mathbb{F}_p$ and $\mathbb{F}_{2^k}$ instances of $\mathcal{F}_{\mathsf{RPrep}}$ with this same input and then the parties compute the n-party XOR. To ensure all parties provided the same inputs in both fields, randomized checking procedures called *cut-and-choose* and *bucketing* are executed. While this is often an expensive procedure, the larger the number of daBits generated the lower the average cost per daBit.

The idea behind these checks is the following. First, the parties open a random subset of secrets so that with some probability all unopened bits are correct. This ensures that the adversary cannot cheat on too many of the daBits. Then the secrets are placed into "buckets", and then in each bucket one secret is designated as the one to output, all other secrets in the bucket are used to check the designated secret, and then they discard all but the designated secret. For a single bucket, the check will only pass (by construction) if either all secrets are correct or all are incorrect. Thus the adversary is forced to corrupt whole multiples of the bucket size and hope they are grouped together in the same bucket. Fortunately, (it will be shown that) there is no leakage on the bits since the parameters required for the parts of the protocol described above already preclude it.

The cut-and-choose and bucketing checks in the protocol presented here are similar to those described by Frederiksen et al. [FKOS15, App. F.3], in which "triple-like" secrets can be checked efficiently. Note that it is not necessary to remove leakage on the bits as described in [FKOS15, p.5] since no operation is performed beyond inputting the bits into the two different fields: for triples, one creates the triple, authenticates and then sacrifices, which means there is the possibility for leakage on the triple.

The protocol $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ is given in Figure 8.7.

Protocol $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$

This protocol is in the $\mathcal{F}_{\mathsf{RPrep}}$-hybrid model. To save on notation, to say that the parties compute $[\![z]\!] := [\![x]\!] \cdot [\![y]\!]$ means that they compute a new identifier id_z, call $\mathcal{F}^p_{\mathsf{RPrep}}$ with input $(\texttt{Multiply}, id_x, id_y, id_z, sid_p)$, and store the output secret-sharing of a secret identified as id_z (and analogously for the $\mathbb{F}_{2^l}$ instance).

Initialize

1. Call an instance of $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^p, sid_p)$; denote it by $\mathcal{F}^p_{\mathsf{RPrep}}$.
2. Call an instance of $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{2^l}, sid_{2^l})$; denote it by $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$.

$\mathcal{F}_{\mathsf{RPrep}}$ subroutines

Dealt with by $\mathcal{F}^p_{\mathsf{RPrep}}$ or $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$, as determined by the session identifier.

daBit subroutine

daBits To generate ℓ bits, the parties do the following:

1. **Generate daBits**
   a) Choose $C > 1$ and $B > 1$ so that $C^B \cdot \binom{B \cdot \ell}{B} > 2^\sigma$.
   b) For each $i \in [n]$,
      i. Party $\mathcal{P}_i$ samples a bit string $(b^i_1, \ldots, b^i_{C \cdot B \cdot \ell}) \leftarrow \mathcal{U}\left(\{0,1\}^{C \cdot B \cdot \ell}\right)$.

Protocol $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ (continued)

ii. The parties call $\mathcal{F}^{p}_{\mathsf{RPrep}}$ where $\mathcal{P}_i$ has input $(\texttt{Input}, i, id_{b^i_k}, b^i_k, sid_p)_{k=1}^m$ and $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ has input $(\texttt{Input}, i, id_{b^i_k}, \perp, sid_p)_{k=1}^m$.

iii. The parties call $\mathcal{F}^{2^l}_{\mathsf{RPrep}}$ and $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$ where $\mathcal{P}_i$ has input $(\texttt{Input}, i, id_{b^i_k}, b^i_k, sid_{2^l})_{k=1}^m$ and $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ has input $(\texttt{Input}, i, id_{b^i_k}, \perp, sid_{2^l})_{k=1}^m$.

iv. The parties store the returned shares $\{[\![b^i_k]\!]^p, [\![b^i_k]\!]^{2^l}\}_{k=1}^m$.

2. **Cut and Choose**

a) Call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\texttt{RSubset}, [C \cdot B \cdot \ell], (C-1) \cdot B \cdot \ell, sid)$ to obtain a set $S \subseteq [C \cdot B \cdot \ell]$ of size $(C-1) \cdot B \cdot \ell$.

b) Call $\mathcal{F}^{p}_{\mathsf{RPrep}}$ with inputs $(\texttt{Open}, 0, id_{b^i_k}, sid_p)_{k \in S}$ for all $i \in \mathcal{P}$.

c) Call $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$ with inputs $(\texttt{Open}, 0, id_{b^i_k}, sid_{2^l})_{k \in S}$ for all $i \in \mathcal{P}$.

d) If any party sees daBits which are not in $\{0,1\}$ or not the same in both fields, they call (either instance of) $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid)$, (locally) output $\perp$, and halt.

3. **Combine** For all $k \in [\ell] \setminus S$, do the following:

a) Set $[\![b_k]\!]^p := [\![b^1_k]\!]^p$ and $[\![b_k]\!]^{2^l} := [\![b^1_k]\!]^{2^l}$.

b) For i from 2 to n,

i. Set $[\![b_k]\!]^p := [\![b_k]\!]^p + [\![b^i_k]\!]^p - 2 \cdot [\![b_k]\!]^p \cdot [\![b^i_k]\!]^p$.

ii. Set $[\![b_k]\!]^{2^l} := [\![b_k]\!]^{2^l} \oplus [\![b^i_k]\!]^{2^l}$.

4. **Check Correctness**

a) Call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\texttt{RBuckets}, [B \cdot \ell], B)$ to obtain a set of sets $\{X_k\}_{k=1}^{\ell}$ to put the $B \cdot \ell$ daBits into ℓ buckets of size B.

b) For each bucket $X_k \in \{X_k\}_{k=1}^{\ell}$,

i. Relabel the bits in this bucket as $b^1, \ldots, b^B$.

ii. Set $[\![c^k]\!]^p := [\![b^1]\!]^p$ and $[\![c^k]\!]^{2^l} := [\![b^1]\!]^{2^l}$.

iii. For k' from 2 to B, compute a new identifier id_{c^k} and do the following:

A. Set $[\![c^k]\!]^p := [\![b^1]\!]^p + [\![b^{k'}]\!]^p - 2 \cdot [\![b^1]\!]^p \cdot [\![b^{k'}]\!]^p$.

B. Set $[\![c^k]\!]^{2^l} := [\![b^1]\!]^{2^l} \oplus [\![b^{k'}]\!]^{2^l}$.

iv. Call $\mathcal{F}^{p}_{\mathsf{RPrep}}$ with inputs $(\texttt{Open}, 0, id_{c^k}, sid_p)_{k=2}^{B}$.

v. Call $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$ with inputs $(\texttt{Open}, 0, id_{c^k}, sid_{2^l})_{k=2}^{B}$.

vi. If any party sees daBits which are not in $\{0,1\}$ or not the same in both fields, they call (either instance of) $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid)$, (locally) output $\perp$, and halt.

vii. Set $[\![b_k]\!]^{p,2^l} := [\![b^1]\!]^{p,2^l}$.

c) Call $\mathcal{F}^{p}_{\mathsf{RPrep}}$ with input $(\texttt{Verify}, sid_p)$.

d) Call $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$ with input $(\texttt{Verify}, sid_{2^l})$.

e) If the checks pass without aborting, output $\{[\![b_k]\!]^{p,2^l}\}_{k=1}^{\ell}$ and discard all other bits.

Figure 8.7: Protocol for Two MPC Engines, with daBits in Both, $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$.

Proposition 8.1 is required in order to show that $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ universal composability (UC)-securely realizes the functionality $\mathcal{F}_{\mathsf{RPrep+}}$ in Figure 8.3 in the $\mathcal{F}_{\mathsf{RPrep}}$-hybrid model.

Proposition 8.1. *For a given $\ell > 0$, choose $B > 1$ and $C > 1$ so that $C^{-B} \cdot \binom{B\ell}{B}^{-1} < 2^{-\sigma}$. Then the probability that one or more of the ℓ daBits output after **Check Correctness** by $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ is different in each field is at most $2^{-\sigma}$.*

Proof. Using $\mathcal{F}^{p}_{\mathsf{RPrep}}$ and $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$ as black boxes ensures the adversary can only possibly cheat in the input stage. It will be argued that:

1. If both sets of inputs from corrupt parties to $\mathcal{F}^{p}_{\mathsf{RPrep}}$ and $\mathcal{F}^{2^k}_{\mathsf{RPrep}}$ are bits (rather than other field elements), then the bits are consistent in the two different fields with overwhelming probability.
2. The inputs in $\mathbb{F}_{2^k}$ are bits with overwhelming probability.
3. The inputs in $\mathbb{F}_p$ are bits with overwhelming probability.

If these hold then it follows that the daBits are bits in the two fields, and are consistent.

1. Let c be the number of inconsistent daBits generated by a given corrupt party. If $c > B\ell$ then every set of size $(C-1)B\ell$ contains an incorrect daBit so the honest parties will always detect this in **Cut and Choose** and abort. Since $(C-1)B\ell$ out of $CB\ell$ daBits are opened, on average the probability that a daBit is not opened is $1-(C-1)/C = C^{-1}$, and so if $c < B\ell$ then

$$\Pr[\text{None of the } c \text{ corrupted daBits is opened}] = C^{-c}. \tag{8.2}$$

At this point, if the protocol has not yet aborted, then there are $B\ell$ daBits remaining of which exactly c are corrupt.

Suppose a daBit $[\![b]\!]^{p,2^l}$ takes the value $\tilde{b}$ in $\mathbb{F}_p$ and $\hat{b}$ in $\mathbb{F}_{2^k}$. If the bucketing check passes then for every other daBit $[\![b']\!]^{p,2^l}$ in the bucket it holds that $\tilde{b} \oplus \tilde{b}' = \hat{b} \oplus \hat{b}'$, so $\tilde{b}' = (\hat{b} \oplus \hat{b}') \oplus \tilde{b}$, and so $\tilde{b} = \hat{b} \oplus 1$ if and only if $\tilde{b}' = \hat{b}' \oplus 1$. (Recall that at this stage it is assumed that the inputs are certainly bits.) In other words, within a single bucket, the check passes if and only if either all daBits are inconsistent, or if none of them are. Thus the probability that **Check Correctness** passes without aborting is the probability that all corrupted daBits are placed into the same buckets. Moreover, this implies that if the number of corrupted daBits, c, is not a multiple of the bucket size, this stage never

passes, so $c = Bt$ for some $t > 0$. Let E be the event that all corrupted daBits are placed in the same buckets. Then

$$\begin{aligned}\Pr[E] &= \frac{\binom{Bt}{B}\cdot\binom{B(t-1)}{B}\cdots\binom{B}{B}\cdot\binom{B\ell-Bt}{B}\cdot\binom{B\ell-Bt-B}{B}\cdots\binom{B}{B}}{\binom{B\ell}{B}\cdot\binom{B\ell-B}{B}\cdots\binom{B}{B}} \\ &= \frac{(Bt)!}{B!^t}\cdot\frac{(B\ell-Bt)!}{B!^{\ell-t}}\cdot\frac{B!^\ell}{(B\ell)!} \\ &= \binom{B\ell}{Bt}^{-1}.\end{aligned}$$

Since the randomness for **Cut and Choose** and **Check Correctness** is independent, the event that both checks pass after the adversary corrupts c daBits is the product of the probabilities. To upper-bound the adversary's chance of winning, the probability is maximized by varying over t: thus it is necessary to find C and B so that

$$\max_t \left\{ C^{-Bt}\cdot\binom{B\ell}{Bt}^{-1} \right\} < 2^{-\sigma}. \tag{8.3}$$

The maximum occurs when t is small, and $t \geq 1$ otherwise no cheating occurred; thus since the proposition stipulates that $C^{-B}\cdot\binom{B\ell}{B}^{-1} < 2^{-\sigma}$, the daBits are consistent in both fields, if they are indeed bits in both fields.

2. Next, it will be argued that the check in **Cut and Choose** ensures that the inputs given to $\mathcal{F}_{\mathsf{RPrep}}^{2^k}$ are indeed bits. It follows from Equation 8.2 that the step **Cut and Choose** aborts with probability C^{-c} if any element of either field is not a bit, as well as if the elements in the two fields do not match. Moreover, in **Check Correctness**, in order for the check to pass in $\mathbb{F}_{2^k}$ for a given bucket, the secrets' higher-order bits must be the same for all shares so that the XOR is always zero when the pairwise XORs are opened. Thus the probability that this happens is the same as the probability above in Equation 8.3 since again this can only happen when the adversary is not detected in **Cut and Choose**, that it cheats in some multiple of B daBits, and that these cheating bits are placed in the same buckets in **Check Correctness**.

3. It will now be shown that all of the $\mathbb{F}_p$ components are bits. To do this, it will be shown that if the $\mathbb{F}_p$ component of a daBit is not a bit, then the bucket check passes only if all other daBits in the bucket are also not bits in $\mathbb{F}_p$.

If the protocol has not aborted, then in every bucket B, for every $2 \leq j \leq B$, it holds that

$$b^1 + b^j - 2 \cdot b^1 \cdot b^j = c^j \tag{8.4}$$

where $c^j \in \{0,1\}$ are determined by the XOR in $\mathbb{F}_{2^k}$. Note that since $c^j = \bigoplus_{i=1}^n b_i^1 \oplus \bigoplus_{i=1}^n b_i^j$ and at least one b_i^j is generated by an honest party, this value is uniform and unknown to the adversary when it chooses its inputs at the beginning.

Suppose $b^1 \in \mathbb{F}_p \setminus \{0,1\}$. If $b^1 = 2^{-1} \in \mathbb{F}_p$ then by Equation 8.4 we have $b^1 = c^j$; but c^j is a bit, so the "XOR" is not the same in both fields and the protocol will abort. Thus it may be assumed that $b^1 \neq 2^{-1}$ and so the equation above can be rewritten as

$$b^j = \frac{b^1 - c^j}{2 \cdot b^1 - 1}. \tag{8.5}$$

Now if b^j is a bit then it satisfies $b^j(b^j - 1) = 0$, and so

$$0 = \left(\frac{b^1 - c^j}{2 \cdot b^1 - 1}\right) \cdot \left(\frac{b^1 - c^j}{2 \cdot b^1 - 1} - 1\right) = -\frac{(b^1 - c^j)(b^1 - (1 - c^j))}{(2 \cdot b^1 - 1)^2}$$

so $b^1 = c^j$ or $b^1 = 1 - c^j$; thus $b^1 \in \{0,1\}$, which is a contradiction. Thus if b^1 is not a bit then b^j is not a bit for every other b^j in this bucket. Moreover, for each $j = 2, \ldots, B$, there are two distinct values $b^j \in \mathbb{F}_p \setminus \{0,1\}$ solving Equation 8.5 corresponding to the two possible values of $c^j \in \{0,1\}$, which means that if the bucket check passes then the adversary must *also* have guessed the bits $\{c^j\}_{j=1}^B$, which it can do with probability 2^{-B} since they are constructed using at least one honest party's input. Thus the chance of cheating without detection in this way is at most $2^{-Bt} \cdot C^{-Bt} \cdot \binom{B\ell}{Bt}^{-1}$.

Thus it has been shown that the probability that $b^1 \in \mathbb{F}_p \setminus \{0,1\}$ is given as output for the $\mathbb{F}_p$ component is at most the probability that the adversary corrupts a multiple of B daBits, that these daBits are placed in the same buckets, and that the adversary correctly guesses c bits from honest parties (in the construction of the bits $\{b^j\}_{j \in B}$) so that the appropriate equations hold in the corrupted buckets. Indeed, needing to guess the bits ahead of time only *reduces* the adversary's chance of winning from the same probability in the $\mathbb{F}_{2^k}$ case.

It is therefore possible to conclude that the daBits are bits in both fields and are the same in both fields with probability except with probability at most $2^{-\sigma}$. □

Theorem 8.1. *The protocol* $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ *UC-securely realizes* $\mathcal{F}_{\mathsf{RPrep+}}$ *against an active, static adversary corrupting up to* $n-1$ *out of* n *parties in the* $\mathcal{F}_{\mathsf{RPrep}}$, $\mathcal{F}_{\mathsf{CoinFlip}}$*-hybrid model.*

Proof. The simulator is given in Figure 8.8.

Simulator $\mathcal{S}_{\mathsf{Prep+}}$

Initialize

1. Await the call to $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^p, 0)$, initialize a local instance, and then call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^p, 0)$.
2. Await the call to $\mathcal{F}_{\mathsf{RPrep}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{2^l}, 1)$, initialize a local instance, and then call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Initialize}, \Gamma, [\![\cdot]\!]^{2^l}, 1)$.

$\mathcal{F}_{\mathsf{RPrep}}$ subroutines

All calls for producing preprocessing, other than what is described for the generation of daBits, below, sent from $\mathcal{A}$ to $\mathcal{F}^p_{\mathsf{RPrep}}$ or $\mathcal{F}^{2^l}_{\mathsf{RPrep}}$, should be forwarded to $\mathcal{F}_{\mathsf{RPrep+}}$. All response messages from $\mathcal{F}_{\mathsf{RPrep+}}$ are relayed directly back to $\mathcal{A}$.

daBit subroutine

daBits Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{daBits}, id_1, \ldots, id_\ell, 0, 1)$.

1. **Generate daBits** Execute **Generate daBits** from $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ with $\mathcal{A}$, sampling inputs for all honest parties.
2. **Cut and Choose** Execute **Cut and Choose** from $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ with $\mathcal{A}$.
3. **Combine** Execute **Combine** from $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ with $\mathcal{A}$.
4. **Check Correctness** Execute **Check Correctness** from $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$ with $\mathcal{A}$. If the protocol aborted, send Abort to $\mathcal{F}_{\mathsf{RPrep+}}$, and otherwise send OK.

Figure 8.8: Simulator $\mathcal{S}_{\mathsf{Prep+}}$ for $\mathcal{F}_{\mathsf{RPrep+}}$.

The simulator merely relays information between $\mathcal{A}$ and $\mathcal{F}_{\mathsf{RPrep+}}$ for the entire execution of the protocol outside of the procedure **daBits**, so the simulation here is perfect. The simulator also honestly executes the oracle $\mathcal{F}_{\mathsf{CoinFlip}}$, so this is also simulated perfectly.

Notice that the parties do not have inputs in the subroutine **daBits** in $\mathcal{F}_{\mathsf{RPrep+}}$, so there is no need for the simulator to extract any inputs from corrupt parties. It remains to show the contribution to the transcripts produced in its execution are indistinguishable, namely that the protocol aborts in the hybrid world with the same distribution as when the honest parties are emulated by the simulator, and that the outputs of all corrupt parties combined with the outputs of honest parties are "consistent" – that is, they correspond to a possible execution of $\mathcal{F}_{\mathsf{RPrep+}}$ or of the protocol.

The correctness of the simulation holds by the fact that the simulator simply executes the protocol as honest parties would, making random choices for honest parties by sampling in the same way as in the protocol.

If the adversary performs a *selective-failure attack*, then the environment may learn information. A selective failure attack is where the environment can learn some information if the protocol does not detect cheating behaviour. For example, if the environment guesses an entire bucket of bits (and thus guesses some daBit) and chooses the adversary's input so that the bucket check would pass based on these guesses, then if the protocol does not abort, then the environment learns that its guesses were correct. Moreover, in this case the simulator does not tell $\mathcal{F}_{\mathsf{RPrep+}}$ to abort and so the environment receives the outputs of honest parties. Then if the final output bit for this daBit is *not* the XOR of the honest parties' final output shares for this daBit with the XOR of the corrupt parties' shares, then the execution must have happened in the ideal world since in this world the output depends on the random tape of $\mathcal{F}_{\mathsf{RPrep+}}$ and is independent of the adversary's and honest parties' random tapes, contrasting the output in the $\mathcal{F}_{\mathsf{RPrep}}, \mathcal{F}_{\mathsf{CoinFlip}}$-hybrid world in which the final output is an XOR of bits on these tapes (which were guessed by the environment). Since this happens with probability $\frac{1}{2}$, in expected 2 executions, the environment can distinguish. However, by Proposition 8.1, the environment can only mount a selective failure attack by making such guesses with success with probability at most $2^{-\sigma}$ by the choice of parameters.

Thus the only way to distinguish between worlds is if the transcript leaks information on the honest parties' inputs. However, the only time data regarding the honest parties' inputs are revealed is in **Check Correctness**, in which XORs are computed in both fields and the result is opened. This reveals no information on the final daBit outputs as the linear dependence between the secret and the public values is broken by discarding all secrets in each bucket except the designated (i.e. first) bit. Thus the overall distributions of the two executions are statistically indistinguishable. □

8.4 Switching and Modified Garbling

In this section, the switching procedures are explained in detail. The modified SPDZ-BMR protocol Π_{CABB} is given in Figures 8.9, 8.10 and 8.11.

Protocol Π_{CABB}

This protocol is realized in the $\mathcal{F}_{\mathsf{RPrep+}}$-hybrid model. To save on notation, to say that the parties compute $[\![z]\!] := [\![x]\!] \cdot [\![y]\!]$ means that they compute a new identifier id_z, call $\mathcal{F}^p_{\mathsf{RPrep}}$ with input $(\texttt{Multiply}, id_x, id_y, id_z, sid_p)$, and store the output secret-sharing of a secret identified as id_z (and analogously for the $\mathbb{F}_{2^l}$ instance). Recall that $\ell := \lfloor \log p \rfloor$.

Initialize The parties call $\mathcal{F}_{\mathsf{RPrep+}}$ with inputs $(\texttt{Initialize}, \mathbb{F}_p, sid_p)$ and $(\texttt{Initialize}, \mathbb{F}_{2^k}, sid_{2^l})$.

Arithmetic Circuit

Input For $\mathcal{P}_i$ to provide input $x \in \mathbb{F}_p$, the parties compute a new identifier id_x, $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Input}, i, id_x, x, sid_p)$ and all other parties call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Input}, i, id_x, \perp, sid_p)$.

Add To add secrets x and y, parties compute a new identifier id_z and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Add}, id_x, id_y, id_z, sid_p)$ and then each $\mathcal{P}_i$ provides input $[\![x]\!]^p_{\mathcal{P}_i}$ and $[\![y]\!]^p_{\mathcal{P}_i}$.

Multiply To multiply secrets x and y, parties call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Multiply}, id_x, id_y, id_z, sid_p)$ where id_z is a new identifier.

Output To receive output x with identifier id_x, parties do the following:

1. Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Verify}, sid_p)$.
2. Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, 0, id_x, sid_p)$.
3. Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Verify}, sid_p)$.

Boolean Circuit (All of the following procedures are performed, in order.)

Initialize garbling To garble a Boolean circuit C with identifiers W for wires, G_{AND} for AND gates and G_{XOR} for XOR gates, the parties do the following:

1. The parties compute new identifiers $\{id_{\lambda_w}\}_{w \in W_o}$ and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{daBits}, \{id_{\lambda_w}\}_{w \in W_o}, sid_p, sid_{2^l})$ where W_o denotes the set indexing circuit output wires.
2. For each $i \in [n]$, the parties compute a new identifier id_{Δ^i} and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{RElt}, id_{\Delta^i}, sid_{2^l})$ and then call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, i, id_{\Delta^i}, sid_{2^l})$ to reveal Δ^i to $\mathcal{P}_i$.

Input layer Let the number of $\mathbb{F}_p$ inputs to the circuit be t. The parties do the following:

1. For $k = 1$ to t,
   a) Compute new identifiers $\{id_{r_{k,j}}\}_{j=0}^{\ell-1}$ and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{daBits}, \{id_{r_{k,j}}\}_{j=0}^{\ell}, sid_p, sid_{2^l})$ to obtain $\{[\![r_{k,j}]\!]^{p,2^l}\}_{j=0}^{\ell}$.
   b) Compute a new identifier id_{r_k} and set $[\![r_k]\!]^p := \sum_{j=0}^{\ell-1} 2^j \cdot [\![r_{k,j}]\!]^p$.
   c) Create the circuit $\texttt{ADDMOD}(a_k, b_k, p)$ and prepend the circuit to C to be garbled, augmenting G_{AND} and G_{XOR} as appropriate. See Section 8.4.1 for details.
   d) For every input wire w corresponding to an input $a_{k,j}$ of $\texttt{ADDMOD}(a_k, b_k, p)$, compute a new identifier $id_{\lambda_{w_{k,j}}}$ and set $[\![\lambda_{w_{k,j}}]\!]^{2^l} := [\![0]\!]^{2^l}$.
   e) For every input wire w corresponding to an input $b_{k,j}$ of $\texttt{ADDMOD}(a_k, b_k, p)$, compute a

Protocol Π_{CABB} (continued)

new identifier $id_{\lambda_{w_{k,j}}}$ and set $[\![\lambda_{w_{k,j}}]\!]^{2^l} \coloneqq [\![r_{k,j}]\!]^{2^l}$.

2. For each input wire $w \in W$, for each $i \in [n]$,
   a) Compute a new identifier $id_{k^i_{w,0}}$ and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{RElt}, id_{k^i_{w,0}}, sid_{2^l})$.
   b) Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, i, id_{k^i_{w,0}}, sid_{2^l})$ to reveal $k^i_{w,0}$ to $\mathcal{P}_i$.
   c) $\mathcal{P}_i$ sets $k^i_{w,1} \coloneqq k^i_{w,0} \oplus \Delta^i$ and the parties set $[\![k^i_{w,1}]\!]^{2^l} \coloneqq [\![k^i_{w,0}]\!]^{2^l} \oplus [\![\Delta^i]\!]^{2^l}$.

Garble Refer to $\Pi_{\mathsf{BMRGarble}}$ in Figure 8.10.

Output layer For every wire w that is an (external, circuit) output wire, the parties do the following

1. Retrieve a daBit $[\![\lambda_{w'}]\!]^{p,2^l}$ from memory, generated in **Initialize**, with identifier $id_{\lambda_{w'}}$.
2. Compute a new identifier $id_{\lambda_{w_0}}$ and set $[\![\lambda_{w_0}]\!]^{2^l} \coloneqq [\![\lambda_w]\!]^{2^l} \oplus [\![\lambda_{w'}]\!]^{2^l}$.
3. Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, 0, id_{\lambda_{w_0}}, sid_{2^l})$; all parties store this locally in memory as the value Λ_{w_0}.

Open To open the circuit, the parties do the following:

1. For every $g \in G_{\mathrm{AND}}$, for every $j \in [n]$, for every $(\alpha, \beta) \in \{0,1\}^2$,
   a) Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, 0, id_{\widetilde{g}^j_{\alpha,\beta}}, sid_{2^l})$. If $\mathcal{F}_{\mathsf{RPrep+}}$ sends the message Abort, then call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Broadcast}, \mathsf{Abort}, sid_p)$, locally output $\perp$, and halt; otherwise continue.
2. Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Verify}, sid_{2^l})$. If $\mathcal{F}_{\mathsf{RPrep+}}$ sends the message Abort, then (locally) output $\perp$ and halt; otherwise, locally output $(id_g, (\widetilde{g}^j_{0,0}, \widetilde{g}^j_{0,1}, \widetilde{g}^j_{1,0}, \widetilde{g}^j_{1,1})^n_{j=1})_{g \in G}$ and the input mask identifiers $id_{r_1}, \ldots, id_{r_t}$ and associated shares $[\![r_1]\!]^p, \ldots, [\![r_t]\!]^p$.

Evaluate Refer to $\Pi_{\mathsf{BMREvaluate}}$ in Figure 8.11.

Figure 8.9: Protocol for Garbling and Evaluating a Circuit, Π_{CABB}.

Subprotocol $\Pi_{\mathsf{BMRGarble}}$

Garble Traversing the circuit in topological order, for every gate $g \in G$ with (internal) input wires u and v and (internal) output wire w,

- If g is an XOR gate, i.e. $g \in G_{\mathrm{XOR}}$,
  1. The parties set $[\![\lambda_w]\!]^{2^l} \coloneqq [\![\lambda_u]\!]^{2^l} \oplus [\![\lambda_v]\!]^{2^l}$.
  2. For each $i \in [n]$, $\mathcal{P}_i$ computes $k^i_{w,0} \coloneqq k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} \coloneqq k^i_{w,0} \oplus \Delta^i$ and all parties set $[\![k^i_{w,0}]\!]^{2^l} \coloneqq [\![k^i_{u,0}]\!]^{2^l} \oplus [\![k^i_{v,0}]\!]^{2^l}$.
- If g is an AND gate, i.e. $g \in G_{\mathrm{AND}}$,
  1. Compute a new identifier id_{λ_w} and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{RBit}, id_{\lambda_w}, sid_{2^l})$ to obtain $[\![\lambda_w]\!]^{2^l}$.
  2. For each $i \in [n]$,

Subprotocol $\Pi_{\mathsf{BMRGarble}}$ (continued)

a) Compute a new identifier $id_{k^i_{w,0}}$ and call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{RElt}, id_{k^i_{w,0}}, sid_{2^l})$ to obtain $[\![k^i_{w,0}]\!]^{2^l}$.

b) Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, i, id_{k^i_{w,0}}, sid_{2^l})$ to reveal $k^i_{w,0}$ to $\mathcal{P}_i$.

c) Party $\mathcal{P}_i$ sets the one key as $k^i_{w,1} := k^i_{w,0} \oplus \Delta^i$.

d) For all four distinct elements $(\alpha,\beta) \in \{0,1\}^2$, and for every $j \in [n]$, the parties compute a new indentifier $id_{F^{g,i,j}_{\alpha,\beta}}$, and then $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{RPrep+}}$ with input $\left(\texttt{Input}, i, id_{F^{g,i,j}_{\alpha,\beta}}, F_{k^i_{u,\alpha},k^i_{v,\beta}}(id_g \| j), sid_{2^l}\right)$ and the other parties with input $\left(\texttt{Input}, i, id_{F^{g,i,j}_{\alpha,\beta}}, \perp, sid_{2^l}\right)$.

3. For all $j \in [n]$ and all $(\alpha,\beta) \in \{0,1\}^2$, the parties compute

$$[\![\tilde{g}^j_{\alpha,\beta}]\!]^{2^l} := \left(\bigoplus_{i=1}^n [\![F^{g,i,j}_{\alpha,\beta}]\!]^{2^l}\right) \oplus [\![k^j_{w,0}]\!]^{2^l} \oplus [\![\Delta^j]\!]^{2^l} \cdot \left((\alpha \oplus [\![\lambda_u]\!]^{2^l}) \cdot (\beta \oplus [\![\lambda_v]\!]^{2^l}) \oplus [\![\lambda_w]\!]^{2^l}\right).$$

Figure 8.10: Subprotocol for Garbling a Circuit, $\Pi_{\mathsf{BMREvaluate}}$.

Subprotocol $\Pi_{\mathsf{BMREvaluate}}$

Evaluate The parties, holding the outputs $(id_g(\tilde{g}^i_{0,0}, \tilde{g}^i_{0,1}, \tilde{g}^i_{1,0}, \tilde{g}^i_{1,1})^n_{i=1})_{g\in G}$, $id_{r_1},\ldots,id_{r_t}$ and $[\![r_1]\!]^p,\ldots,[\![r_t]\!]^p$ of $\Pi_{\mathsf{BMRGarble}}$, evaluate in the following way, traversing the circuit in topological order:

1. For each input $\{[\![x_k]\!]^p\}_{k\in[t]}$, the parties do the following:
   a) Retrieve from memory the secret mask $[\![r_k]\!]^p$ (locally) output in $\Pi_{\mathsf{BMRGarble}}$.
   b) Compute a new identifier id_{a_k} and set $[\![a_k]\!]^p := [\![x_k]\!]^p - [\![r_k]\!]^p$.
   c) Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Open}, 0, id_{a_k}, sid_p)$.
   d) Denote the corresponding input wires by $\{w_{k,j}\}_{j=0}^{\lfloor \log p \rfloor -1}$. Bit-decompose the public value a_k and let the bits be $\{a_{k,j}\}_{j=0}^{\lfloor \log p \rfloor -1}$.
   e) For each $j = 0,\ldots,\lfloor \log p \rfloor - 1$, set $\Lambda_{w_{k,j}} := a_{k,j} \oplus 0$.
   f) For each $i \in [n]$, $\mathcal{P}_i$ sends $\{k^i_{w_{k,j},\Lambda_{w_{k,j}}}\}_{j=0}^{\lfloor \log p \rfloor -1}$ to all other parties.
   g) The parties call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\texttt{Verify}, sid_p)$.
2. For every $g \in G$,
   a) If g is an XOR gate,
      i. Party P_i computes $\Lambda_w := \Lambda_u \oplus \Lambda_v$.
      ii. Party P_i computes all n output keys indexed by $j \in [n]$, as $k^j_{w,\Lambda_w} := k^j_{u,\Lambda_u} \oplus k^j_{v,\Lambda_v}$.
   b) If g is an AND gate,
      i. Each party computes the n keys indexed by $j \in [n]$ as

$$k^j_{w,\Lambda_w} := \tilde{g}^j_{\Lambda_u,\Lambda_v} \oplus \left(\bigoplus_{i=1}^n F_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g \| j)\right)$$

      and compares its keys $k^i_{w,0}$ and $k^i_{w,1}$ to the i^{th} key obtained to determine the global

Subprotocol $\Pi_{\mathsf{BMREvaluate}}$ (continued)

signal bit Λ_w.

3. For every external output wire w,
   a) Retrieve from memory the corresponding public signal bit Λ_{w_0} produced in **Output layer**.
   b) Locally compute $\Lambda_{w'} := \Lambda_{w_0} \oplus \Lambda_w$.
   c) Locally compute the secret output as

$$[\![b_w]\!]^p := \Lambda_{w'} + [\![\lambda_{w'}]\!]^p - 2 \cdot \Lambda_{w'} \cdot [\![\lambda_{w'}]\!]^p.$$

4. Send the message $(\texttt{Verify}, sid_{2^l})$ to $\mathcal{F}_{\mathsf{RPrep+}}$.

Figure 8.11: Subprotocol for Evaluating a Garbled Circuit, $\Pi_{\mathsf{BMREvaluate}}$.

8.4.1 Conversion from LSSS to GC

In brief, the parties open $x - r$ in MPC where $r = \sum_{j=0}^{\lfloor \log p \rfloor - 1} 2^j \cdot [\![r_j]\!]^p$ is constructed from daBits $\left\{[\![r_j]\!]^{p,2^l}\right\}_{j=0}^{\lfloor \log p \rfloor - 1}$, and $\mathcal{F}_{\mathsf{RPrep}}$ is called with input $(\texttt{Verify}, sid_p)$ either at this point or later on, and then these public values are taken to be signal bits to the circuit. To correct the offset r, the circuit simply takes in the $\mathbb{F}_{2^k}$ parts of the daBits of r as input, and the circuit $(x - r) + r \mod p$ is computed inside the garbled circuit.

In more detail, consider the following Boolean circuit

$$\texttt{ADDMOD}(a, b, p) := (a + b) - p \cdot \left((a + b) \stackrel{?}{\geq} p\right)$$

where the computation takes place over the integers and the inputs a and b are supplied as a string of bits.

Let the input wires of $\texttt{ADDMOD}(a, b, p)$ be $(u_j)_{j=0}^{\lfloor \log p \rfloor - 1}$ for the bits of a, $(v_j)_{j=0}^{\lfloor \log p \rfloor - 1}$ for the bits of b, and the output wires be $(w_j)_{j=0}^{\lfloor \log p \rfloor - 1}$, and let the input wires of C be $(u'_j)_{j=0}^{\lfloor \log p \rfloor - 1}$. Then the circuit that the parties garble in Π_{CABB} is the circuit obtained by associating wire w_j with wire u'_j for every $j = 0$ to $\lfloor \log p \rfloor - 1$. Now if $a = x - r$ and $b = r$ then clearly $(C \circ \texttt{ADDMOD})(a, b, p) = C(x)$ where $\circ$ denotes the wiring association as above.

Optimization With some tweaking of the garbling protocol, it is possible to reduce communication and computation costs further. Since $x - r$ is a public value, there is no need to have masks for the corresponding input wires, so they can be set to 0. Furthermore, since r is independent of the online inputs, the bits used to construct r can be hard-wired into the circuit, reducing preprocessing *and* online costs: if the masking

bits are set to be equal to the daBits used to construct r, i.e. $\{r_j\}_{j=0}^{\lfloor \log p \rfloor - 1}$, then when computing the ciphertexts, for the input u corresponding to the bit r_j, it holds that $\alpha = \Lambda_u = \lambda_u \oplus r_j = 0$, so the ciphertexts are

$$[\![\tilde{g}^j_{0,\beta}]\!]^{2^l} := \left(\bigoplus_{i=1}^{n} [\![F_{k^i_{u,0},k^i_{v,\beta}}(id_g, j)]\!]^{2^l} \right) \oplus [\![k^j_{w,0}]\!]^{2^l} \oplus [\![\Delta^j]\!]^{2^l} \cdot \left((0 \oplus [\![\lambda_u]\!]^{2^l}) \cdot (\beta \oplus [\![\lambda_v]\!]^{2^l}) \oplus [\![\lambda_w]\!]^{2^l} \right)$$

for $\beta \in \{0,1\}$. Thus there are only $2n$ ciphertexts instead of $4n$, and consequently also only $2n$ keys must be opened online instead of $4n$. Note that in order to avoid cluttering the description, in the protocol in Figure 8.9, all four ciphertexts are computed and opened.

8.4.2 Conversion From GC to LSSS

The output of a multi-party garbled circuit is one or more keys and corresponding public signal bits. In SPDZ-BMR, the output wire masks are revealed after garbling so that all parties can learn the final outputs. A simple way of retaining shared output of the circuit, as required in mixed protocols, is for the parties not to reveal the masks for output wires after garbling and instead to compute the XOR of the secret-shared mask with the public signal bit, in MPC. In other words, for output wire w they obtain a sharing of the secret output bit b by computing

$$[\![b]\!]^{2^l} := \Lambda_w \oplus [\![\lambda_w]\!]^{2^l}.$$

However, the output must be in $\mathbb{F}_p$. If the circuit output wires are daBits, then the parties can (locally) compute a sharing of each output bit as

$$[\![b]\!]^p := \Lambda_w + [\![\lambda_w]\!]^p - 2 \cdot \Lambda_w \cdot [\![\lambda_w]\!]^p.$$

If such modifications to the garbling protocol are not possible, then the following approach gives a cheap conversion procedure. One can define an additional layer to the circuit after the output layer which converts output wires with masks only in $\mathbb{F}_{2^l}$ to output wires with masks as daBits, without changing the real values on the wire. To do this, parties do the following: for every output wire w,

1. In the garbling stage,
   a) Take a new daBit $[\![\lambda_{w'}]\!]^{p,2^l}$.
   b) Set $[\![\lambda_{w_0}]\!]^{2^l} := [\![\lambda_w]\!]^{2^l} \oplus [\![\lambda_{w'}]\!]^{2^l}$.

c) Call $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\mathtt{Open}, 0, id_{\lambda_{w_0}}, sid)$ an call this $\Lambda_{w_0} = 0 \oplus \lambda_{w_0}$.

2. In the evaluation stage, upon obtaining Λ_w,

   a) Compute $\Lambda_{w'} \coloneqq \Lambda_w \oplus \Lambda_{w_0}$.

   b) Compute the final ($\mathbb{F}_p$-secret-shared) output as $[\![b]\!]^p \coloneqq \Lambda_{w'} + [\![\lambda_{w'}]\!]^p - 2 \cdot \Lambda_{w'} \cdot [\![\lambda_{w'}]\!]^p$.

Observe that $\Lambda_{w_0} \equiv \lambda_{w_0}$ so this procedure is just adding a layer of XOR gates where the masking bits are daBits and the other input wire is always 0 (so the gate evaluation doesn't change the real wire value), as shown in Figure 8.12. This was the labelled "Mask Conversion" layer in Figure 8.2. Note that since the signal bits for XOR gates are determined from input signal bits and not the output key, there is no need to generate an output key for wire w_0.

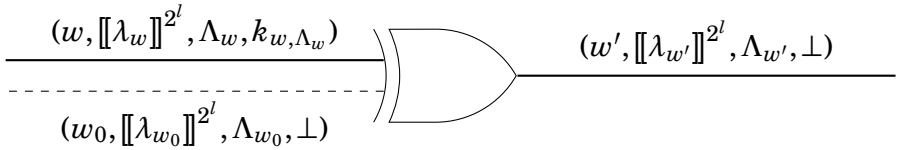


Figure 8.12: Circuit Output Wires.

For correctness, observe that:

$$\begin{aligned}\Lambda_{w'} \oplus \lambda_{w'} &= (\Lambda_w \oplus \Lambda_{w_0}) \oplus (\lambda_w \oplus \lambda_{w_0}) \\ &= \big((b \oplus \lambda_w) \oplus (0 \oplus \lambda_{w_0})\big) \oplus (\lambda_w \oplus \lambda_{w_0}) \\ &= b.\end{aligned}$$

8.4.3 Security

Correctness of the actual garbling was outlined in Section 8.2. The proof of Theorem 8.2 follows from the security of SPDZ-BMR [LSS16] and [KY18], and the fact that the additional input and output procedures perfectly hide the actual circuit inputs and outputs.

Theorem 8.2. *The protocol* Π_{CABB} *UC-securely realizes the functionality* $\mathcal{F}_{\mathsf{CABB}}$ *against a static, active adversary in the* $\mathcal{F}_{\mathsf{RPrep+}}$*-hybrid model.*

Proof (Sketch). Security essentially follows immediately from the security of the garbling protocol, the fact that $\mathcal{F}_{\mathsf{Prep}}$ can be used to realize $\mathcal{F}_{\mathsf{ABB}}$ UC-securely via the protocol Π_{Online} (which means that $\mathcal{F}_{\mathsf{RPrep}}$ can also be used to realize it also using Π_{Online}

since it is merely an extension of $\mathcal{F}_{\mathsf{Prep}}$), and the fact that the only communication in the switching procedure involves revealing inputs masked by random values in $\mathbb{F}_p$. A sketch of the proof is now given highlighting these key points.

The only secrets to which the simulator is not privy are the secret inputs of honest parties. Everything else in the protocol involves calls to $\mathcal{F}_{\mathsf{RPrep+}}$ which is locally emulated by the simulator. Moreover, preprocessing is not output by $\mathcal{F}_{\mathsf{CABB}}$, which means that any preprocessing generated to perform the garbling or circuit evaluation can be emulated perfectly by the simulator.

Suppose there are h honest parties in total, and that they are indexed by $[h] \subseteq [n]$, and let **Hybrid i** be defined as follows:

Hybrid i The $\mathcal{F}_{\mathsf{RPrep+}}$-hybrid world in which the simulator is handed the inputs of all honest parties $\mathcal{P}_j \in \mathcal{P} \setminus \mathcal{A}$ where $j \leq i$.

The simulator is given in Figure 8.13.

Simulator $\mathcal{S}_{\mathsf{CABB}}$

Initialize Initialize a local copy of $\mathcal{F}_{\mathsf{RPrep+}}$ and await the inputs $(\texttt{Initialize}, \mathbb{F}_p, sid_p)$ and $(\texttt{Initialize}, \mathbb{F}_{2^k}, sid_{2^l})$ from $\mathcal{A}$.

Arithmetic Circuit

Calls to $\mathcal{F}_{\mathsf{RPrep+}}$ with $sid = sid_p$ have the same format as corresponding calls in $\mathcal{F}_{\mathsf{CABB}}$, so the simulator just relays these calls. For any calls to preprocessing, the simulator executes a local copy of $\mathcal{F}_{\mathsf{RPrep+}}$ and emulates honest parties honestly.

Boolean Circuit (All of the following procedures are performed, in order.)

Initialize garbling Run **Initialize** from Π_{CABB} with $\mathcal{A}$.

Input layer Run **Input layer** from Π_{CABB} with $\mathcal{A}$.

Garble Run $\Pi_{\mathsf{BMRGarble}}$ with $\mathcal{A}$.

Output layer Run **Output layer** from Π_{CABB} with $\mathcal{A}$.

Open Run **Open** from Π_{CABB} with $\mathcal{A}$.

Evaluate Call $\mathcal{F}_{\mathsf{CABB}}$ with input $(\texttt{EvaluateCircuit}, C, id_{x_1}, \ldots, id_{x_t}, id, sid_p)$ and then do the following:

1. For each $k = 1$ to t do the following:
   a) Retrieve the masks $[\![r_k]\!]^p$ generated during $\Pi_{\mathsf{BMREvaluate}}$ as well as the shares of the input $[\![x_k]\!]^p$.

Simulator $\mathcal{S}_{\mathsf{CABB}}$ (continued)

b) Compute a new identifier id_{a_k} and set $[\![a_k]\!]^p := [\![x_k]\!]^p - [\![r_k]\!]^p$.
c) Await the call to $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\mathtt{Open}, 0, id_{a_k}, sid_p)$ from $\mathcal{A}$ and execute it honestly to obtain some $a_k \in \mathbb{F}_p$.
d) (This step requires no simulation.)
e) (This step requires no simulation.)
f) Await the calls to $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\mathtt{Open}, 0, id_{k_{w_{k,j},a_{k,j}}}, sid_{2^l})_{j=0}^{\ell-1}$ from $\mathcal{A}$, where $w_{k,j}$ is the wire for the j^{th} input bit $a_{k,j}$ of a_k, and execute the procedure honestly.
g) Await the call to $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\mathtt{Verify}, sid_p)$ and execute it honestly, aborting if (emulated) honest parties would.

2. Execute the circuit evaluation honestly on behalf of (emulated) honest parties, aborting if they would abort. to $\mathcal{F}_{\mathsf{CABB}}$. If an honest party would have aborted then the simulator sends Abort to $\mathcal{F}_{\mathsf{CABB}}$.
3. Execute the output layer procedures honestly on behalf of (emulated) honest parties.
4. Await the call to $\mathcal{F}_{\mathsf{RPrep+}}$ with input $(\mathtt{Verify}, sid_{2^l})$. If at any point an (emulated) honest party aborted, send the message Abort to $\mathcal{F}_{\mathsf{CABB}}$, and otherwise send the message OK.

Figure 8.13: Simulator $\mathcal{S}_{\mathsf{CABB}}$ for $\mathcal{F}_{\mathsf{CABB}}$.

The world **Hybrid h** is exactly the ideal world in which the simulator receives the inputs of no honest parties. The world **Hybrid 0** is exactly the $\mathcal{F}_{\mathsf{RPrep+}}$-hybrid world and is simulated perfectly since the simulator just executes the protocol on behalf of honest parties. No extraction of inputs is required since the ideal calls of $\mathcal{A}$ to $\mathcal{F}_{\mathsf{RPrep+}}$ contain all necessary information to pass on messages to $\mathcal{F}_{\mathsf{CABB}}$ so that the functionality will provide the same outputs to real honest parties.

Claim 8.1. **Hybrid i** is indistinguishable from **Hybrid i+1** for $i = 0, \ldots, h-1$.

Proof. For the emulation of $\mathcal{F}_{\mathsf{RPrep+}}$ with $sid = sid_p$ and for the garbling the simulation is perfect since honest parties' inputs are not required.

The ability of the simulator to equivocate outputs in the simulation $\mathcal{S}_{\mathsf{ABB}}$ of the protocol Π_{Online} that realizes $\mathcal{F}_{\mathsf{ABB}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid world implies the simulator here can do the same when $\mathcal{F}_{\mathsf{CABB}}$ provides output and the adversary calls $\mathcal{F}_{\mathsf{RPrep+}}$ to obtain outputs of secrets. The upshot of this is that the values on which the garbled circuit is evaluated in the protocol do not matter, since circuit evaluation does not leak information about the inputs (as proved in [LPSY15]), and $\mathcal{F}_{\mathsf{CABB}}$ only outputs elements of $\mathbb{F}_p$, so whatever is given as output from $\mathcal{F}_{\mathsf{CABB}}$ can be used to replace the simulated output: it only matters that the protocol aborts with the same distribution as in a real execution, which it does because the simulator emulates the behaviour of honest parties.

However, there is one point in the protocol in which the transcript is dependent on the real inputs of honest parties (which are unknown to the simulator), which is when secrets in $\mathbb{F}_p$ are opened after being masked with daBits. However, since the secret masks $[\![r]\!]^p$ are constructed from uniformly-sampled bits, by Lemma 8.1 the distribution of the uniformly sample $r' \leftarrow \mathcal{U}(\mathbb{F}_p)$ is statistically close to the distribution of $a - r \bmod p$ where a is the input of an honest party and $r \leftarrow \mathcal{U}(\{0,1\}^{\lfloor \log p \rfloor})$. ■

Since there are a polynomial number of hybrid worlds that are statistically indistinguishable, **Hybrid 0** and **Hybrid h** are also statistically indistinguishable – that is, the $\mathcal{F}_{\mathsf{RPrep+}}$-hybrid world is indistinguishable from the ideal world. □

8.5 Realizing the Protocol

When implementing the daBit generation, there are various caveats and possible optimizations to the protocol described, a few of which are outlined in this section.

Choice of prime It is necessary that p be close to a power of 2 so that $x - r$ is (statistically) indistinguishable from a uniform element of the field, as discussed in Section 8.2. If SPDZ is used to perform the MPC, a technique known as *packing* is used to amortize the costs; this technique requires that p be congruent to 1 mod N where $N = 2^{15} = 32768$. This means that the prime is always different from a power of two by at least 15 bits since the l-bit prime must be of the form $2^{l-1} + k \cdot 2^{15} + 1$ for some k where $1 \leq k \leq 2^{l-16} - 1$, so the secret masks r constructed from a sequence of bits "miss" at least this much of the field.

Cut and choose optimization Parties only need to input bits (instead of full field field elements) into $\mathcal{F}_{\mathsf{RPrep}}$ during $\mathcal{F}_{\mathsf{RPrep}} \| \Pi_{\mathsf{daBits}}$, which means that for the instance of $\mathcal{F}_{\mathsf{RPrep}}$ over $\mathbb{F}_{2^l}$, only authenticated "bit" masks need to be generated for the input phase: for $\mathcal{F}_{\mathsf{RPrep}}$ over $\mathbb{F}_{2^l}$, full field element masks are generally constructed by generating a set of l authenticated bits. This trick cannot be applied to the instance of $\mathcal{F}_{\mathsf{RPrep}}$ over $\mathbb{F}_p$.

Share conversion To reduce the amount of garbling when converting an additive share to a GC one, if the $\mathbb{F}_p$ inputs to the garbled circuit are assumed to be bounded

by $p/2^\sigma$, then a uniform r in $\mathbb{F}_p$ is 2^σ times larger than x so $x-r$ is statistically-indistinguishable from a uniform element of $\mathbb{F}_p$; consequently, one need only garble $(x-r)+r$ and not $(x-r)+r \bmod p$, which makes the circuit marginally smaller

8.6 Application: Computation of a Multi-Class SVM

The application and implementation as described here was completed by the coauthor of this work. It is included for the sake of completeness.

A (multi-class) Linear Support Vector Machine (SVM) is a method for classifying images using machine learning. It involves deciding on a set of classes and computing a matrix A and a vector $\mathbf{b}$ from a large dataset generated by some well-defined method in such a way that when a new input "feature vector", $\mathbf{x}$, is provided as input, the index of the largest component of the vector $A \cdot \mathbf{x} + \mathbf{b}$ determines the class. The key computation is therefore the function

$$\mathsf{ARGMAX}(A \cdot \mathbf{x} + \mathbf{b}),$$

that is, computing the index of the first component of the vector $A \cdot \mathbf{x} + \mathbf{b}$ attaining the maximum value in the vector (i.e. $\|A \cdot \mathbf{x} + \mathbf{b}\|_\infty$), for a given vector $\mathbf{x}$.

Computing an SVM can be done in MPC. The application for this is that an organization or company that trains the model (i.e. computes A and $\mathbf{b}$) on a large private dataset may want to allow clients to query the model on their own private input (i.e. $\mathbf{x}$). In this case, the MPC computation is that of computing

$$\mathsf{ARGMAX}\left([\![A]\!] \cdot [\![\mathbf{x}]\!] + [\![\mathbf{b}]\!]\right),$$

where $[\![A]\!]$ and $[\![\mathbf{b}]\!]$ are input by the company with the private dataset, and $[\![\mathbf{x}]\!]$ by the client that wishes to classify their image.

The efficiency of the switching protocols (called "circuit marbling") described in this chapter, using daBits, was tested for an SVM. This particular circuit was chosen because it is clear that it uses a combination of arithmetic computation, namely $[\![A]\!] \cdot [\![\mathbf{x}]\!] + [\![\mathbf{b}]\!]$, and bit-wise operations, namely ARGMAX, which are better suited to LSSS-based and GC-based MPC, respectively. In the experiments, the online phase of an SVM with 100 classes and 128 features was benchmarked by simulating a WAN with a round-trip ping time of 100ms and 50Mb/s bandwidth with two parties. Concretely, this means that $\mathbf{x} \in \mathbb{F}_p^{128}$, $A \in \mathbb{F}_p^{102\times 128}$ and $\mathbf{b} \in \mathbb{F}_p^{102}$. This is the same SVM structure used by Makri

et al. [MRSV19] to classify the Caltech-101 dataset [LAR03] which contains 102 different categories of images such as aeroplanes, dolphins, helicopters, and others. The particular SVM has bounded inputs so that $\mathbf{x}$ can be viewed as a vector $\mathbf{x} \in (\mathbb{Z}/2^{25}\mathbb{Z})^{128}$, a field size p where $\log p = 128$ and statistical security $\sigma = 64$.

The results are given in Tables 8.2 and 8.3, where "Marbled SPDZ" refers to the protocols as described in this chapter. The costs are given in terms of data items (i.e. triples, bits, and AND gates) as the concrete costs depend on the implementation of MPC and GC that is used. Timings were computed using [Kel19].

The online phase using Marbled-SPDZ is one order of magnitude faster than both SPDZ-BMR and SPDZ. The price paid for this efficiency in the online phase is that approximately 2.6 times the number of triples are required when compared to SPDZ; however, Marbled-SPDZ requires significantly fewer garbled AND gates, essentially because arithmetic modulo p is expensive in a Boolean circuit, saving by a factor of almost 400.

Protocol	Sub-Protocol	Preprocessing cost		
		$\mathbb{F}_p$ triples	$\mathbb{F}_p$ bits	AND gates
SPDZ	n/a	19015	9797	n/a
SPDZ-BMR	n/a	n/a	n/a	14088217
Marbled-SPDZ	SPDZ	0	13056	n/a
	daBit convert	63546	0	27030
	SPDZ-BMR	n/a	n/a	8383

Table 8.2: Two-party linear SVM: single-threaded (non-amortized) preprocessing phase costs with $\sigma = 64$.

It is clear that by relegating computation to the preprocessing phase by generating daBits, there is considerable speed-up in the online phase. Thus in situations in which preprocessing is outsourced, daBits are a useful form of data. However, the total amount of preprocessing required is significantly more than for plain SPDZ.

In conclusion, the method presented here for generating daBits is very costly and there may be situations in which parties can generate them with authentication without treating the garbling and secret-sharing in a "completely" black-box way. Despite this, it seems that they are indeed a useful form of preprocessed data.

Protocol	Sub-Protocol	Online cost		
		Comm. rounds	Time (ms)	Total (ms)
SPDZ	n/a	54	2661	2661
SPDZ-BMR	n/a	0	2786	2786
Marbled-SPDZ	SPDZ	1	133	272
	daBit convert	2	137	
	SPDZ-BMR	0	2	

Table 8.3: Two-party linear SVM: single-threaded (non-amortized) online phase costs with $\sigma = 64$.

Chapter 9

Conclusions

After a little over 30 years, multi-party computation (MPC) is finally starting to become practical, and is deployed in real-world applications. Nonetheless, there is still a lot to be done. This chapter lists a few of the areas of research that would follow well from the topics covered in this thesis.

Asynchronous MPC The protocols in this work have assumed synchronous communcation, in which every party receives all messages in a given round before sending any messages for the next round. In asynchronous protocols, the adversary is permitted to delay messages, which models real-world communication over wide-area networks (WANs) such as the Internet much more closely.

It would be interesting to investigate how the error-detection properties of a $\mathscr{Q}_2$ access structure could be used to develop asynchronous protocols. For example, if the parties were under a $\mathscr{Q}_3$ access structure, then when opening secrets in each round of communication they could wait until they received enough shares to reconstruct and then continue, but meanwhile wait to receive enough shares to perform error-detection, and then abort if they detect that a party is sending erroneous shares.

Examining Share-Reconstructability It turned out that share-reconstructability of a monotone span program was a useful property not only for error-detection, but also for randomness extraction. It would be interesting to see if this property is related to any other properties of secret-sharing schemes in the literature.

In a similar vein, it would be interesting to know for which complete monotone non-redundant $\mathscr{Q}_2$ access structures there exists a partition indexed by $[n]$, as required by the protocol in Chapter 6. Considerable effort went into finding Example 6.2.

Generalizing Outsourcing The outsourcing protocol only explored outsourcing to a set of parties using the same "type" of authentication. It would be interesting to develop a methodology for mixing the different types. Indeed, recently, Smart and Tanguy [ST19] showed how to use resharing techniques to outsource preprocessing from a $\mathscr{Q}_2$ access structure to a full-threshold access structure, giving "Triples-as-a-Service". It would be informative to implement and compare these techniques with the "naïve" methods for generating preprocessing (i.e. without outsourcing).

Better daBit Generation The method involving cut and choose for generating daBits in Chapter 8 is not very efficient, requiring several Beaver triples to check correctness of a single daBit. One of the reasons for this is that, prior to this thesis, it was not possible to talk about the explicit shares of secrets when using MPC (namely, when using $\mathcal{F}_{\mathsf{Prep}}$ to realize $\mathcal{F}_{\mathsf{ABB}}$ or to execute the SPDZ-BMR protocol) since secrets were always assumed to be stored in an "authenticated dictionary" to which parties only had black-box access, as was discussed in Section 4.1. Now that there is a methodology for $\mathcal{F}_{\mathsf{Prep}}$ involving shares explicitly, there is no need to use secrets in a completely black-box way, and consequently there may be more efficient ways to generate daBits. Improving the cost to generate daBits to the extent that the overall cost of preprocessing *and* online time was less than the same time for executing either SPDZ or a multi-party garbling protocol would be a major breakthrough.

To complement the support vector machine example, it would also be interesting to find a good application for evaluating a garbled circuit first and then linear secret-sharing scheme (LSSS)-based MPC.

Mixing Homomorphic Encryption and MPC The protocol in Chapter 8 allowed conversion between the two of the main pillars of computing on private data – LSSS and garbled circuits (GCs). However, there remains one further expansive area of research in this field in the form of homomorphic encryption. Mixed protocols involving these *three* techniques may lead to some interesting results, as indeed has been shown by Henecka et al. in the case of two parties with passive security [HKS+10].

Bibliography

[ABF+17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein.
Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier.
In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[ABF+18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida.
Generalizing the SPDZ compiler for other protocols.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 880–895, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[ACK+19] A Aly, D Cozzo, M Keller, E Orsini, D Rotaru, P Scholl, NP Smart, and T Wood.
Scale–mamba v1. 4: Documentation.
2019.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara.
High-throughput semi-honest secure three-party computation with an honest majority.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 805–817, Vienna, Austria, October 24–28, 2016. ACM Press.

[Bar86] David A. Mix Barrington.

Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1.
In *18th Annual ACM Symposium on Theory of Computing*, pages 1–5, Berkeley, CA, USA, May 28–30, 1986. ACM Press.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft.
Secure multiparty computation goes live.
In Roger Dingledine and Philippe Golle, editors, *FC 2009: 13th International Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343, Accra Beach, Barbados, February 23–26, 2009. Springer, Heidelberg, Germany.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai.
Compressing vector OLE.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 896–912, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BDK+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider.
HyCC: Compilation of hybrid protocols for practical secure computation.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 847–861, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias.
Semi-homomorphic encryption and multiparty computation.
In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[Bea92] Donald Beaver.
Efficient multiparty protocols using circuit randomization.

In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

[Ben18] Aner Ben-Efraim.
On multiparty garbling of arithmetic circuits.
In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 3–33, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky.
Near-linear unconditionally-secure multiparty computation with a dishonest minority.
In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[BGP95] Amos Beimel, Anna Gál, and Mike Paterson.
Lower bounds for monotone span programs.
In *36th Annual Symposium on Foundations of Computer Science*, pages 674–681, Milwaukee, Wisconsin, October 23–25, 1995. IEEE Computer Society Press.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan.
(Leveled) fully homomorphic encryption without bootstrapping.
In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson.
Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract).
In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway.
Efficient garbling from a fixed-key blockcipher.

In *2013 IEEE Symposium on Security and Privacy*, pages 478–492, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[Bla79] G. R. Blakley.
Safeguarding cryptographic keys.
Proceedings of AFIPS 1979 National Computer Conference, 48:313–317, 1979.

[Blu81] Manuel Blum.
Coin flipping by telephone.
In Allen Gersho, editor, *Advances in Cryptology – CRYPTO'81*, volume ECE Report 82-04, pages 11–15, Santa Barbara, CA, USA, 1981. U.C. Santa Barbara, Dept. of Elec. and Computer Eng.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson.
Sharemind: A framework for fast privacy-preserving computations.
In Sushil Jajodia and Javier López, editors, *ESORICS 2008: 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway.
The round complexity of secure protocols (extended abstract).
In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek.
Garbling gadgets for boolean and arithmetic circuits.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 565–577, Vienna, Austria, October 24–28, 2016. ACM Press.

[BPSW07] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel.
Privacy-preserving remote diagnostics.
In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007: 14th Conference on Computer and Communications Security*, pages 498–507, Alexandria, Virginia, USA, October 28–31, 2007. ACM Press.

[BR93] Mihir Bellare and Phillip Rogaway.
Random oracles are practical: A paradigm for designing efficient protocols.
In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[BW98] Donald Beaver and Avishai Wool.
Quorum-based secure multi-party computation.
In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 375–390, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany.

[Can00] Ran Canetti.
Universally composable security: A new paradigm for cryptographic protocols.
Cryptology ePrint Archive, Report 2000/067, 2000.
http://eprint.iacr.org/2000/067.

[CCD88a] David Chaum, Claude Crépeau, and Ivan Damgård.
Multiparty unconditionally secure protocols (abstract) (informal contribution).
In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, page 462, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany.

[CCD88b] David Chaum, Claude Crépeau, and Ivan Damgård.
Multiparty unconditionally secure protocols (extended abstract).
In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[CDG+05] Ronald Cramer, Vanesa Daza, Ignacio Gracia, Jorge Jiménez Urroz, Gregor Leander, Jaume Martí-Farré, and Carles Padró.
On codes, matroids and secure multi-party computation from linear secret sharing schemes.
In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 327–343, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai.

Share conversion, pseudorandom secret-sharing and applications to secure computation.
In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer.
General secure multi-party computation from any linear secret-sharing scheme.
In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish.
Universally composable security with global setup.
In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85, Amsterdam, The Netherlands, February 21–24, 2007. Springer, Heidelberg, Germany.

[CF01] Ran Canetti and Marc Fischlin.
Universally composable commitments.
In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi.
The random oracle methodology, revisited (preliminary version).
In *30th Annual ACM Symposium on Theory of Computing*, pages 209–218, Dallas, TX, USA, May 23–26, 1998. ACM Press.

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof.
Fast large-scale honest-majority MPC for malicious adversaries.
In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou.

On the security of the “free-XOR” technique.
In Ronald Cramer, editor, *TCC 2012: 9th Theory of Cryptography Conference*, volume 7194 of *Lecture Notes in Computer Science*, pages 39–53, Taormina, Sicily, Italy, March 19–21, 2012. Springer, Heidelberg, Germany.

[CP17] Ashish Choudhury and Arpita Patra.
An efficient framework for unconditionally secure multiparty computation.
IEEE Transactions on Information Theory, 63(1):428–468, 2017.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen.
Asynchronous multiparty computation: Theory and implementation.
In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart.
Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits.
In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

[DMP11] Emiliano De Cristofaro, Mark Manulis, and Bertram Poettering.
Private discovery of common social contacts.
In Javier Lopez and Gene Tsudik, editors, *ACNS 11: 9th International Conference on Applied Cryptography and Network Security*, volume 6715 of *Lecture Notes in Computer Science*, pages 147–165, Nerja, Spain, June 7–10, 2011. Springer, Heidelberg, Germany.

[DN03] Ivan Damgård and Jesper Buus Nielsen.
Universally composable efficient multiparty computation from threshold homomorphic encryption.
In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[DN07] Ivan Damgård and Jesper Buus Nielsen.
Scalable and unconditionally secure multiparty computation.
In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias.
Multiparty computation from somewhat homomorphic encryption.
In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[DSZ14] Daniel Demmler, Thomas Schneider, and Michael Zohner.
Ad-hoc secure two-party computation on mobile devices using hardware tokens.
In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014: 23rd USENIX Security Symposium*, pages 893–908, San Diego, CA, USA, August 20–22, 2014. USENIX Association.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner.
ABY - A framework for efficient mixed-protocol secure two-party computation.
In *ISOC Network and Distributed System Security Symposium – NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society.

[Dur64] Richard Durstenfeld.
Algorithm 235: Random permutation.
Commun. ACM, 7(7):420–, July 1964.

[DZ13] Ivan Damgård and Sarah Zakarias.
Constant-overhead secure computation of Boolean circuits using preprocessing.
In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641, Tokyo, Japan, March 3–6, 2013. Springer, Heidelberg, Germany.

[Feh99] Serge Fehr.
Efficient construction of the dual span program.

1999.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl.
A unified approach to MPC with preprocessing using OT.
In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 711–735, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein.
High-throughput secure three-party computation for malicious adversaries and an honest majority.
In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[FY48] Ronald Aylmer Fisher and Frank Yates.
Statistical tables for biological, agricultural and medical research.
Statistical tables for biological, agricultural and medical research., (3rd ed), 1948.

[GI99] Niv Gilboa and Yuval Ishai.
Compressing cryptographic resources.
In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 591–608, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson.
How to play any mental game or A completeness theorem for protocols with honest majority.
In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg.
TASTY: tool for automating secure two-party computations.

In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010: 17th Conference on Computer and Communications Security*, pages 451–462, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.

[HM97] Martin Hirt and Ueli M. Maurer.
Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract).
In James E. Burns and Hagit Attiya, editors, *16th ACM Symposium Annual on Principles of Distributed Computing*, pages 25–34, Santa Barbara, CA, USA, August 21–24, 1997. Association for Computing Machinery.

[HM00] Martin Hirt and Ueli M. Maurer.
Player simulation and general adversary structures in perfect multiparty computation.
Journal of Cryptology, 13(1):31–60, January 2000.

[HOSS18a] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez.
Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT).
In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 86–117, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

[HOSS18b] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez.
TinyKeys: A new approach to efficient multi-party computation.
In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 3–33, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez.
Low cost constant round MPC combining BMR and oblivious transfer.
In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Com-*

puter Science, pages 598–628, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[Hua12] Yan Huang.
Practical secure two-party computation.
PhD thesis, University of Virginia, 2012.

[IMZ19] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas.
Efficient mpc via program analysis: A framework for efficient optimal mixing.
Cryptology ePrint Archive, Report 2019/651, 2019.
To appear at ACM CCS2019. https://eprint.iacr.org/2019/651.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai.
Founding cryptography on oblivious transfer - efficiently.
In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[ISN87] M. Ito, A. Saito, and Takao Nishizeki.
Secret sharing schemes realizing general access structure.
In *Proc. IEEE Global Telecommunication Conf. (Globecom'87)*, pages 99–102, 1987.

[ISN93] M. Ito, A. Saito, and Takao Nishizeki.
Multiple assignment scheme for sharing secret.
Journal of Cryptology, 6(1):15–20, March 1993.

[KBPB19] John Kelsey, Luís T. A. N. Brandão, René Peralta, and Harold Booth.
A reference for randomness beacons, format and protocol version 2.0, Apr 2019.

[Kel19] Marcel Keller.
Multi-Protocol SPDZ, 2019.
https://github.com/n1analytics/MP-SPDZ.

[Kil88] Joe Kilian.
Founding cryptography on oblivious transfer.
In *20th Annual ACM Symposium on Theory of Computing*, pages 20–31, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[Knu97] Donald E. Knuth.

The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl.
MASCOT: Faster malicious arithmetic secure computation with oblivious transfer.
In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru.
Overdrive: Making SPDZ great again.
In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[KRSW18] Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood.
Reducing communication channels in MPC.
In Dario Catalano and Roberto De Prisco, editors, *SCN 18: 11th International Conference on Security in Communication Networks*, volume 11035 of *Lecture Notes in Computer Science*, pages 181–199, Amalfi, Italy, September 5–7, 2018. Springer, Heidelberg, Germany.

[KS08] Vladimir Kolesnikov and Thomas Schneider.
Improved garbled circuit: Free XOR gates and applications.
In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498, Reykjavik, Iceland, July 7–11, 2008. Springer, Heidelberg, Germany.

[KSS14] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer.
Automatic protocol selection in secure two-party computations.
In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14: 12th International Conference on Applied Cryptography and Network Security*, volume 8479 of *Lecture Notes in Computer Science*, pages 566–

584, Lausanne, Switzerland, June 10–13, 2014. Springer, Heidelberg, Germany.

[KW93] Mauricio Karchmer and Avi Wigderson.
On span programs.
In *Proceedings of Structures in Complexity Theory*, pages 102–111, 1993.

[KY18] Marcel Keller and Avishay Yanai.
Efficient maliciously secure multiparty computation for RAM.
In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 91–124, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[LAR03] Fei-Fei Li, Marco Andreetto, and Marc 'Aurelio Ranzato.
Caltech101 image dataset.
2003.

[LP07] Yehuda Lindell and Benny Pinkas.
An efficient protocol for secure two-party computation in the presence of malicious adversaries.
In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78, Barcelona, Spain, May 20–24, 2007. Springer, Heidelberg, Germany.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai.
Efficient constant round multi-party computation combining BMR and SPDZ.
In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 319–338, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease.
The byzantine generals problem.
ACM Trans. Program. Lang. Syst., 4(3):382–401, July 1982.

[LSS16] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez.
More efficient constant-round multi-party computation from BMR and SHE.

In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 554–581, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany.

[Mau06] Ueli M. Maurer.
Secure multi-party computation made simple.
Discrete Applied Mathematics, 154(2):370–381, 2006.

[MR18] Payman Mohassel and Peter Rindal.
ABY^3: A mixed protocol framework for machine learning.
In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[MRSV19] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren.
EPIC: Efficient private image classification (or: Learning from the masters).
In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 473–492, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra.
A new approach to practical active-secure two-party computation.
In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams.
Secure two-party computation is practical.
In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner.
Phasing: Private set intersection using permutation-based hashing.

In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 515–530, Washington, DC, USA, August 12–14, 2015. USENIX Association.

[RW19] Dragos Rotaru and Tim Wood.
MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security.
Cryptology ePrint Archive, Report 2019/207, 2019.
https://eprint.iacr.org/2019/207.

[RWT+18] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar.
Chameleon: A hybrid secure computation framework for machine learning applications.
In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*, pages 707–721, Incheon, Republic of Korea, April 2–6, 2018. ACM Press.

[Sch96] Bruce Schneier.
Applied Cryptography.
John Wiley & Sons, New York, second edition, 1996.

[Sha79] Adi Shamir.
How to share a secret.
Communications of the Association for Computing Machinery, 22(11):612–613, November 1979.

[SSW17] Peter Scholl, Nigel P. Smart, and Tim Wood.
When it's all just too much: Outsourcing MPC-preprocessing.
In Máire O'Neill, editor, *16th IMA International Conference on Cryptography and Coding*, volume 10655 of *Lecture Notes in Computer Science*, pages 77–99, Oxford, UK, December 12–14, 2017. Springer, Heidelberg, Germany.

[ST19] Nigel P. Smart and Titouan Tanguy.
Taas: Commodity mpc via triples-as-a-service.
Cryptology ePrint Archive, Report 2019/957, 2019.
https://eprint.iacr.org/2019/957.

[SW19] Nigel P. Smart and Tim Wood.
Error detection in monotone span programs with application to communication-efficient multi-party computation.
In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 210–229, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz.
Authenticated garbling and efficient maliciously secure two-party computation.
In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 21–37, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz.
Global-scale secure multiparty computation.
In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 39–56, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[Yao86] Andrew Chi-Chih Yao.
How to generate and exchange secrets (extended abstract).
In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans.
Two halves make a whole - reducing data transfer in garbled circuits using half gates.
In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

Index