KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Secure Multiparty Computation, Inside and Out of the Head

Robin Jadoul

Supervisor:
Prof. dr. Nigel Paul Smart

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

January 2025

Secure Multiparty Computation, Inside and Out of the Head

Robin JADOUL

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

January 2025

Preface

Research is not something that happens in isolation. While it sounds like a cliché, it is a very true saying, even when the effects of a well-known pandemic could still be felt when starting out doing research. The interaction with all the great people in and around COSIC over the years has been an absolute pleasure and a privilege. From coffee and a chat, over coffee and talking cryptography, to coffee and something sweet because there's a reason to celebrate (can you tell there's been quite a bit of coffee?), the semi-regular boardgame nights, and the occasional other social events; I have enjoyed them all. Thank you.

More towards the other side of the work-life balance, there are of course more people to be acknowledged and thanked. First and foremost among them, my supervisor. Thanks, Nigel, for the inspiration, the guidance, and the occasional prodding to get things done. This journey would have looked entirely different without you. Thanks then too, to the rest of my supervisory committee: Fre Vercauteren and Danny Hughes for the questions and feedback on my intermediary checkpoints. Extending on this, more thanks to the same people and yet more: my examination committee. Thank you, Jean-Pierre Celis, Jeongeun Park and Diego Aranha, for reading this work, and challenging me with interesting and engaging questions and discussions at the defence. By extension also my thanks to you, hypothetical future reader, whoever you may be, for taking the time to read this.

As I claimed that research does not happen in isolation, I should like to thank all my co-authors as well, for the fruitful and enjoyable collaborations (ordered alphabetically; as usual, there is no metric I could use that would make sense): Carsten Baum, Lennart Braun, Barry van Leeuwen, Emmanuela Orsini, Jeongeun Park, Hilder Pereira, Cyprien de Saint Guilhem, Peter Scholl, Nigel Smart and Titouan Tanguy. Outside of the purely academic work, there is always a lot of paperwork to be taken care of as well, which would not go quite as smoothly if it weren't for the people behind the scenes taking care of the administration and finances. Thanks, Péla, Wim, Anouk and Elsy.

Of course, I would like to thank all friends and family who provided me with the necessary support along the way. Even if you did not understand what I was working on or talking about for most of the time, just the opportunity to try and explain something in an understandable way can help me clear up my own thoughts often enough. If you were one of the lucky ones who did not have to sit through one of those attempted explanations, rest assured that the interaction, the distraction, and your general presence were well appreciated too.

Finally, as I have done before, I'd like to sincerely extend my thanks to Alice, Bob, Eve, Peggy, Victor, and the entire rest of that wonderful pantheon of cryptographic characters (or who would forget all the nameless computational parties P_i). Your curiosity, your malice and your desires for security, privacy and all sorts of new cryptographic techniques will always be an inspiration to me.

Abstract

Cryptographic approaches to privacy-preserving computation problems have known a long line of research and engineering work to achieve simultaneous security and performance guarantees. However, most of the effort for arithmetic circuits has been focused on the setting for large finite fields, causing a disconnect between the properties provided by the protocols, and those expected by people more accustomed to "regular" computers. This, in turn, leads to unwanted overhead when these protocols are applied to programs written for regular computers, as is often the case with increased adoption by a wider technological audience.

In this thesis, we aim to investigate and construct efficient and practical cryptographic protocols for Multiparty Computation (MPC) and Zero-Knowledge Proofs (ZKP), as well as the intersection and combination of both settings, over the arithmetic domain conventionally used by computers: $\mathbb{Z}_{2^k}$. This domain further improves the practicality and efficiency by respectively allowing a more straightforward translation of user programs, and implementations relying on native arithmetic.

Concretely, we first provide an overview of the necessary mathematical and cryptographical background concerning the foundations of MPC and ZKPs for arithmetic circuits. We then present and examine several protocols for both MPC and ZKP over the rings $\mathbb{Z}_{2^k}$ — which can be considered to be the native evaluation domain of a modern binary computer, — sometimes through the lens of the more general form $\mathbb{Z}_{p^k}$ additionally covering small and large fields.

Along the way, we also investigate a different performance consideration and tradeoff in the form of lifting a designated-verifier ZKP into a setting we call the "distributed-verifier" setting. Here we turn the verifier into a distributed protocol that guarantees the security properties as long as no sufficiently large collaborations between the parties — including the prover — exists. In return, we gain a ZKP that inherits efficiency from the designated-verifier protocol while simultaneously becoming, in a certain sense, more publicly verifiable.

Beknopte samenvatting

Cryptografische benaderingen van problemen rond privacy-beschermende berekeningen kennen een lange lijn aan onderzoeks- en ontwikkelingswerk om tegelijkertijd garanties te kunnen bieden voor veiligheid en performantie. Voor het grootste deel van de inspanningen voor aritmetische circuits ligt de focus echter op grote eindige lichamen. Dit veroorzaakt een belangrijk verschil tussen de eigenschappen die de protocols aanbieden, en de eigenschappen die iemand meer gewend aan "gewone" computers zou verwachten. Op zijn beurt leidt dit tot ongewenste overhead wanneer deze protocols een toepassing vinden voor programma's geschreven voor gewone computers, zoals zo vaak gebeurt bij een toegenomen ingebruikname door een breder technologisch doelpubliek

In deze thesis stellen we tot doel om efficiënte en praktisch toepasbare cryptografische protocols voor Multiparty Computation (MPC) en Zero-Knowledge Proofs (ZKP) te onderzoeken en te construeren, samen met het overlap en de combinatie van beide domeinen, met berekeningen over $\mathbb{Z}_{2^k}$. Dit rekendomein zorgt verder voor verbeteringen in toepasbaarheid en efficiëntie door respectievelijk een meer directe vertaling van programma's van gebruikers toe te laten, en door implementaties in een natuurlijke rekenomgeving te plaatsen.

Concreet voorzien we eerst een overzicht van de noodzakelijke wiskundige en cryptografische achtergrond met betrekking tot de grondslagen van MPC en ZKP voor aritmetische circuits. Vervolgens presenteren en onderzoeken we verscheidene protocols voor zowel MPC als ZKP over de ringen $\mathbb{Z}_{2^k}$ — die gezien kunnen worden als het natuurlijke evaluatiedomein van een moderne binaire computer, — soms door de lens van de meer algemene vorm $\mathbb{Z}_{p^k}$ die eveneens kleine en grote lichamen vervat.

Onderweg onderzoeken we ook een andere overweging van performantie en de afweging ervan in de vorm van een vertaling van een designated-verifier ZKP naar een context die we de “distributed-verifier” setting noemen. Hier veranderen we de verifier in een gedistribueerd protocol dat de veiligheidseigenschappen garandeert zolang er geen afdoend grote samenwerking tussen de partijen — inclusief de prover — bestaat. In ruil daarvoor krijgen we een ZKP dat de efficiëntie van het designated-verifier protocol erft en tegelijkertijd, in zekere zin, meer publiekelijk verifieerbaar wordt.

List of Abbreviations

2PC	Secure Two-Party Computation.
AES	Advanced Encryption Standard.
DV-ZKPoK	Distributed Verifier ZKPoK.
ESP	Extended Span Program.
LSSS	Linear Secret Sharing Scheme.
MAC	Message Authentication Code.
MitH	MPC-in-the-Head.
MPC	Secure Multiparty Computation.
MPCitH	MPC-in-the-Head.
MSP	Monotone Span Program.
NIZK	Non-Interactive Zero-Knowledge Proof.
ORAM	Oblivious RAM.
PKI	Public-Key Infrastructure.
PoK	Proof of Knowledge.
RAM	Random-Access Memory.

RO Random Oracle.

ROM Random Oracle Model.

SNARK Succinct Non-interactive Argument of Knowledge.

SSS Secret Sharing Scheme.

STARK Scalable, Transparent Argument of Knowledge.

UC Universal Composability.

VM Virtual Machine.

ZK Zero-Knowledge.

ZKP Zero-Knowledge Proof.

ZKPoK Zero-Knowledge Proof of Knowledge.

List of Symbols

$[s]$	A secret sharing of the value s
C	A circuit
$\mathcal{M}$	A MSP or ESP
$\mathcal{P}$	A set of parties (for MPC); The prover (for ZKPs)
$\mathcal{V}$	The verifier
$\mathbb{F}$	A field
$\mathbb{F}_q$	The finite field of order q
$GR(q, d)$	The Galois ring of degree d modulo q
λ_Q	The reconstruction vector for a qualified set Q in a linear secret sharing scheme
$\mathbb{C}$	The field of the complex numbers
$\mathbb{Q}$	The field of the rationals
$\mathbb{R}$	The field of the reals
N	The number of parties
$\mathcal{O}_R$	A random oracle
Rec	The reconstruction function of a secret sharing scheme
Share	The sharing function of a secret sharing scheme
$\mathbb{Z}$	The ring of integers

$\mathbb{Z}_n$	The ring of integers modulo n
p	A prime
P_i	A party
q	A prime power, p^k
$V(\alpha_1, \ldots, \alpha_t)$	The Vandermonde matrix over the points $(\alpha_1, \ldots, \alpha_t)$

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Cryptography is a discipline with many different faces. For a kid wanting to keep their diary a secret, it could be as simple as replacing every letter by the next one in the alphabet — though this approach is also said to have been used by Caesar to communicate with his generals — or writing backwards with mirrored letters. For a film or an advertisement campaign, it could mean playfully adding some secret, lightly obfuscated information to the background of a scene to reward the observant viewer and drive engagement. For a company, it could mean securely storing the recipe for their secret sauce; ensuring its availability for the employees that need it, but keeping it away from the prying eyes of industrial espionage.

In general, one could say that the overall goal of cryptography is about managing "trust" and data. The kid writing in their diary might not trust their younger sibling to respect their privacy, but is willing to trust the sibling's lack of understanding of the code or their laziness to figure it out. When communicating over the internet, we might not trust the routers in between that are forwarding all messages, but if we assume or trust that certain mathematical problems are hard, we are able to construct a secure communication channel. Within that paradigm, we can then distinguish three major subfields, classified by the usage of the involved data. The first two will be more familiar to most readers, and are in fact clearly recognizable in the above examples: "data in transit" and "data at rest". Data in transit concerns itself with all kinds of communication situations, dealing with things like secure transactions on the internet and secure messaging (e.g. what happens when you send a message on WhatsApp). For data at rest, we care more about storing, retrieving and updating data, such as you might encounter when dealing with disk encryption on your phone.[1]

[1] In some situations, it may be tempting to think about data at rest as a special case of data in transit, as you may essentially be sending a message to the "future you". There are

The third subfield could be labelled as "data during computation", and some topics from this subfield will be the focus of this thesis. The prototypical example — and indeed one of the problems that started the field of secure multiparty computation — is Yao's millionaire problem [Yao82], which considers two millionaires who wish to determine who among them is the richest, without revealing their wealth to each other. Later, [BCD+09] showed a first real-world application of this type of protocol by running a private double auction of Danish sugar beets with it. While this focus will be more cryptographically and mathematically inspired, it may prove useful to first consider the engineering approach to securing data during computation.

1.1 From Engineering to Cryptography

The engineering way to perform private computation relies on *Trusted Execution Environments* (TEEs). These are computer chips that have been constructed in such a way that not even the owner or operator of the chip has the ability to read or write memory other than in the ways allowed by the currently running program. Additionally, the manufacturer of the TEE embeds a private signing key into the hardware that can be used to perform *remote attestation*, that can prove to the user both that the TEE is a genuine piece of hardware (and not simulated in software to give a false sense of security) and that the code running inside the TEE is what the user expects it to be. All of this functionality however requires a certain level of trust in the security guarantees offered by the hardware. In practice, hardware is a complex construction, where micro-architectural attacks and side channels are discovered frequently enough that this trust may not be guaranteed. Instead, in this thesis, we investigate protocols that can manage this trust issue by involving cryptographical hardness assumptions, collaboration between multiple parties and mathematical techniques. Regarding the security of the protocols in this thesis, we wish to stress that all building blocks are either information-theoretical or post-quantum secure primitives. This means that all protocols we present here can be considered future-proof, even when considering the hypothetical arrival of cryptographically relevant quantum computers.

Next, we distinguish a few different functionalities that a TEE could offer, and briefly discuss the cryptographic alternatives. The cryptographically aware reader may recognize in these examples that the TEE acts as a straw man for a trusted third party, or an ideal functionality. In practice, it is worth noting that cryptographic algorithms allow for more functionality and advantages than

some important differences however, including the need for securely updating the data and the lack of interactivity between the sender and the receiver.

TEEs are used for in real-world applications, as they can additionally provide resilience against denial of service attacks, unreliable networks, colluding parties, and more.

Delegated computation A "client" with low computational power (such as an IoT device that is constrained by its battery) wishes to perform an expensive computation on some private inputs and learn the corresponding output. Since the memory of the TEE cannot be read by the operator, the client can send encrypted inputs and receive the output similarly encrypted from the TEE. From a cryptographic perspective, a technique known as *Fully Homomorphic Encryption* (FHE) allows exactly this use case, at the cost of an increased computational cost for the evaluator and increased ciphertext size. We do not discuss this sort of encryption scheme in any detail in this work, but wish to make the reader aware of its existence so that they are better equipped to make informed decisions on the right kind of cryptography to use.

Private computation In a similar vein, multiple computers may each have some data that they would like to combine and learn some function of all inputs without any party learning more information about another's inputs besides what is revealed from the output. A TEE could act as a *trusted third party* that can receive all inputs (each encrypted under a party-specific key), compute the function, and send the output to each party that is allowed to learn it. A direct application of FHE does not easily apply in a situation like this, since the function evaluation requires all data to be encrypted under the same key, leading to anyone with access to the key being able to decrypt all inputs.[2] Instead, we can let all parties jointly execute a *Secure Multiparty Computation* (MPC) protocol, that will ensure data privacy as long as an insufficient amount of parties collaborates to break it. The exact amount of parties needed to break privacy depends strongly on the specific protocol being used, but is commonly a third, half or all of the parties.

Verifiable computation The above techniques by themselves can ensure that inputs remain private, but they may not yet guarantee that the computed output itself was computed honestly. The TEE chip can avoid this integrity issue through the use of remote attestation that confirms the code being executed is the code requested by the client. The cryptographic approach to verifiable

[2]There do however exist so-called *Multi-key* FHE schemes that, at noticeable increase in computational cost, enable joint computation on ciphertexts encrypted by different parties, without requiring communication between them. These schemes can even be used in some cases as a building block for MPC protocols.

computation relies on *Zero-Knowledge Proofs* (ZKPs) that can attest to the output being correctly calculated from the inputs. ZKPs can even prove the correctness of some secret data without revealing it. A common example would be proving knowledge of an encryption key that corresponds to a known pair of plaintext and ciphertext.[3] This turns out to be a very powerful functionality that finds much use in privacy-preserving applications and as a building block for even more elaborate cryptographic protocols.

1.2 Protocols and Efficiency

While the transition from hardware-based privacy to cryptography manages to alleviate our worries about the security of TEEs and bases trust instead in well-studied mathematics, it also comes with a significant downside. Hardware is *fast*; software and cryptography, less so.

Making use of cryptographic protocols often means introducing additional communication between different parties. On top of that, many mathematicians — and hence the protocols they design — usually work with so-called finite fields.[4] Modern hardware on the other hand, generally works modulo some power of two (so the ring $\mathbb{Z}_{2^k}$), which is not natively compatible with the structure of a finite field. That means that “normal” programs — i.e. those written for contemporary hardware — need to be emulated inside the finite field structure to recover the original semantics, whereas a TEE-based approach would naturally admit the same structure as the “insecure” hardware. This leads to one major contribution of this thesis: the study of MPC and ZKP protocols over the ring structure $\mathbb{Z}_{2^k}$ favoured by hardware.

Cryptographic protocols may also come in different varieties. Sometimes, a protocol is designed to perform exactly one task, and do it as optimized as possible. For instance one could attempt to design a protocol that can perform a single private comparison as efficiently as possible, so that the millionaires from before can start spending their money on drinks faster, rather than wasting time figuring out who exactly should pay for it. On the other hand, it is also possible to build a protocol that can securely compute *any* function. This approach may not be the fastest possible in every single situation, but it may be the most cost-effective as it requires no extra time to first design the protocol. In this thesis, we focus on protocols of the latter kind, in the hope that we can provide

[3]This functionality could also be achieved in a TEE (standing in for a trusted third party) by framing it as a private computation problem between two parties.

[4]This is of course not just because they like being quirky, finite fields are a common occurrence and well studied within cryptography, as they provide a lot of nice structure and properties to work with.

both fast and generic protocols, as well as provide inspiration and techniques that enable faster specialized protocols when the need arises.

1.3 Chapter Overview

The remainder of this thesis can be subdivided into two different kinds of chapter. First are some preliminary chapters that aim to introduce, in an intuitive and understandable manner, the main concepts and building blocks required for the later chapters. In turn, those later chapters correspond to papers published and presented at conferences.[5] As is customary within the field, and due to the impossibility of disentangling contributions out of fruitful discussions and collaborations, the authors on the papers are listed in alphabetical order.[6]

Chapter 2: Rings of the Form $\mathbb{Z}_{2^k}$ In this chapter, we define and explore the mathematical objects of finite fields and rings, so that we can properly quantify the difference between the mathematician's and the computer's approach discussed earlier in the introduction. We shall see that, while there are differences of significance, there are also useful parallels to draw between the two, not in the least the constructions of Galois fields and rings. Afterwards, we also introduce some methods for probabilistic equality testing of sequences over rings and fields, which will prove to be important building blocks for many protocols later on.

Chapter 3: Multiparty Computation from Linear Secret Sharing In this chapter, we first introduce and formalize the concept of secret sharing as a way to distribute private data across multiple parties without revealing any information about it. A specific kind of secret sharing scheme will then give rise to the computation of linear functions over shared secrets. Finally, we can augment these linear secret sharing schemes into fully-fledged MPC protocols through the design of multiplication protocols. The chapter includes all essential background knowledge on access structures, security and performance to understand the contributions in later chapters.

Chapter 4: Zero-Knowledge Proofs In this final preliminary chapter, we introduce the concept of Zero-Knowledge Proofs, which have — besides being

[5]Rather, the chapters correspond to the full version of those papers.

[6]For a more elaborate statement on this matter, the reader may refer to `https://www.ams.org/profession/leaders/CultureStatement04.pdf`.

an interesting primitive in their own right — several useful applications in areas such as verifiable computation and the design of signature schemes. The general goal of a ZKP is to convince a verifier of the veracity of a certain statement, without revealing any further information (or "knowledge"). We discuss some early protocols for specialized statements as a means to familiarize the reader with the definitions and the material, before moving on to constructions of ZKPs based on MPC protocols, which are essential for the understanding of later chapters.

Chapter 5: MPC for $\mathcal{Q}_2$ Access Structures over Rings and Fields This is the first chapter based on a published paper, namely [JSv22]. In it, we examine and compare several MPC protocols for so-called $\mathcal{Q}_2$ access structures. A $\mathcal{Q}_2$ access structure can be considered as a generalization of the *honest-majority* setting, in which at least half the participating parties are assumed to act honestly. Our protocol designs work for both small and large finite fields as well as the rings $\mathbb{Z}_{p^k}$, which importantly also covers the setting of interest $\mathbb{Z}_{2^k}$. Our evaluation of the protocols takes into account the expected communication costs per multiplication gate, depending on the access structure, and the round complexity depending on the multiplicative depth of the computation. We also present an open source programmatic cost estimation tool and framework for the estimation of communication costs for MPC protocols. My personal contributions to this paper are centred around the design of the different protocols and the evaluation thereof, including the case analysis of several specific secret sharing schemes and the development of the cost estimator.

Chapter 6: Feta: Efficient Threshold Designated-Verifier Zero-Knowledge Proofs This paper [BJO+22] explores the intersection between MPC and ZKPs. In particular, it seeks to find a tradeoff between public verifiability of the proof system and the computational cost for the prover. It achieves this by turning a designated-verifier proof system into a *distributed-verifier* one, where the proof is efficiently verified by a set of parties executing an MPC protocol. We present concretely efficient protocols in the threshold setting, alongside an implementation and performance analysis. I was strongly involved with the protocol design and responsible for the full implementation of our proof of concept and the experiments. Being on this intersection between protocol design and implementation allowed me to find and apply improvements across this boundary, by taking inspiration from what was problematic on one side to improve the other side.

Chapter 7: ZK-for-Z2K: MPC-in-the-Head Zero-Knowledge Proofs for $\mathbb{Z}_{2^k}$
In this paper [BdSGJ+24], we once again turn our attention towards the "computer" structure $\mathbb{Z}_{2^k}$. We build efficient Zero-Knowledge protocols for statements over this ring based on the MPC-in-the-Head paradigm, which can transform MPC protocols into proof systems. Our protocols make use of the structure afforded by Galois rings and admit different underlying secret sharing schemes and access structures. Within our protocols, we describe a method to use and verify (oblivious) memory accesses as well as two orthogonal ways to efficiently prove multiple instances of the same statement in parallel. For our resulting protocols, we examine and compare the resulting communication costs or proof sizes. My contribution was focussed on the design and integration of the various subprotocols (multiplication and ring checks) together with the introduction of the packing in the Galois domain.

CHAPTER 2

Rings of the Form $\mathbb{Z}_{p^k}$

In this chapter, we first restate some mathematical definitions regarding rings and fields, as well as the construction of Galois fields. We then have a closer look at rings of the form $\mathbb{Z}_{p^k}$, together with the issues introduced by the presence of zero divisors and how to deal with them. In doing so, we will work with Galois Rings, which follow a parallel construction to that of Galois fields, building upon $\mathbb{Z}_{p^k}$ instead of $\mathbb{F}_p$.

2.1 Finite Fields

In order to calculate arithmetic operations and perform computations, it is useful to first identify which operations can be somehow considered to be *fundamental*. For those operations, we can then determine which properties, in particular those we know from working with the "everyday numbers", we want to keep and build an abstract mathematical definition around. With the definition, we can then identify other mathematical objects that possess the same structure. After all, computer memory is finite, and we cannot — nor do we really want to, most of the time — deal with numbers that grow excessively large or numbers that require exceedingly large precision.[1]

2.1.1 Definitions

Taking inspiration from the real numbers, a first choice for addition and multiplication as mathematical operations becomes natural. For each of these, we want the usual properties to hold, being commutativity, associativity and the presence of a *neutral* element: respectively zero and one, which should be distinct. Additionally, to ensure proper interaction between these two operations, we require multiplication to distribute over addition. Finally, each of the two operations must be invertible. That is, for every element there is an element such that they add to the zero and an element such that they multiply to one. The only exception to this is zero: no inverse element for multiplication can or should exist.

Definition 2.1: Field

A field $\mathbb{F}$ is a tuple $(F, +, \cdot)$, where F is a set and $+, \cdot$ are binary operations $F \times F \to F$, such that:

(a) $+$ and $\cdot$ are associative: $a + (b + c) = (a + b) + c = a + b + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot b \cdot c$

(b) $+$ and $\cdot$ are commutative: $a + b = b + a$ and $a \cdot b = b \cdot a$

(c) There are two distinct elements $0 \neq 1$ in F such that $a + 0 = a \cdot 1 = a$

(d) Addition has an inverse: $\forall a \in F, \exists(-a) : a + (-a) = 0$

(e) Multiplication has an inverse: $\forall a \neq 0 \in F, \exists(a^{-1}) : a \cdot (a^{-1}) = 1$

[1] You could compare this to how we can calculate an arbitrary amount of digits for the number $\pi \in \mathbb{R}$, while in practice, using even just the number 3 as an approximation will often already give useful results.

(f) Multiplication distributes over addition: $a \cdot (b + c) = a \cdot b + a \cdot c$

Since we often care about the relations between mathematical objects, it can be interesting to clearly define and distinguish maps between fields. In particular maps that ensure the properties of the fields are fully preserved are of interest, which we call homomorphisms. As we shall discuss later, it is sometimes possible to find a copy of one field inside another, and this embedding can be made explicit by describing the homomorphism that takes any element of the contained field into the containing field.

Definition 2.2: Homomorphism of fields

A homomorphism (of fields) is a structure-preserving map from one field $\mathbb{F}_1 = (F_1, +, \cdot)$ to another $\mathbb{F}_2 = (F_2, \oplus, \odot)$. It is a function $\varphi : \mathbb{F}_1 \to \mathbb{F}_2$, such that $\varphi(1_1) = 1_2$, $\varphi(a + b) = \varphi(a) \oplus \varphi(b)$ and $\varphi(a \cdot b) = \varphi(a) \odot \varphi(b)$.

If a homomorphism is invertible, we call it an *isomorphism*.

2.1.2 Constructions

From the above definition 2.1, it is easy to verify that our main sources of inspiration for the definition all satisfy the required properties: the real numbers $\mathbb{R}$, the complex numbers $\mathbb{C}$ and the rational numbers $\mathbb{Q}$ all form fields. Observe that the integers $\mathbb{Z}$ fail to be a field, as condition (e) is not satisfied for any integer other than 1 and -1.

Recall now that our goal in abstracting over these structures was to find fields that could efficiently be represented by a computer, without needing unbounded memory to represent any single field element. Hence, we want to find some *finite* mathematical structure that satisfies the field axioms; a *finite field*.

It is a known result that all finite fields have a number of elements that is equal to a prime power, and that additionally only one finite field exists for each such cardinality.[2] Moreover, concrete instantiations and constructions of all finite fields are known. We now describe these in two stages. First we describe the finite fields of prime size, and then extend upon these to build finite fields with a prime power size.

[2]Of course, multiple representations of each field may exist, so alternatively one can state that all fields of a given size are isomorphic to each other.

To obtain a field of prime size, it suffices to look at the integers modulo a prime p: $\mathbb{Z}_p$ (also written $\mathbb{F}_p$ when talking about the structure as being a field). The cardinality (and finiteness) of the structure can be easily verified, as well as associativity, commutativity and distributivity. As neutral elements of addition and multiplication, 0 and 1 play their usual role. The additive inverse of an element a is $-a$ or equivalently $p-a$, since $a+(p-a) \equiv 0 \pmod{p}$. To find the multiplicative inverse of an element a, we can refer to Bézout's identity, which tells us that for any element coprime to p — so modulo p, any non-zero element — we can write $x \cdot a + y \cdot p = 1$ for some integers x and y, and consequently that $x \cdot a \equiv 1 \pmod{p}$ and as such $a^{-1} = x$.

This construction does however not generalize to any other, non-prime moduli. As a counterexample, it is sufficient to look for the multiplicative inverse of 3 in $\mathbb{Z}_6$ or $\mathbb{Z}_9 = \mathbb{Z}_{3^2}$. As respectively $3 \cdot 2 \equiv 0$ and $3 \cdot 3 \equiv 0$, such an element cannot exist. This means that to build up fields of prime power order, we will need to find a different construction.

One way in which we can construct an object of the right cardinality p^k is to take k copies of $\mathbb{F}_p$ in a tuple. Here, a lawful addition can be defined as the componentwise addition, but unfortunately the construction cannot properly support multiplication. We can however build upon this approach and add some extra structure. Rather than letting the copies of $\mathbb{F}_p$ be entirely independent, we make them the coefficients of a polynomial of degree $k-1$. Addition is still defined correspondingly, but while multiplication can be defined and no longer acts purely componentwise, the multiplication of two degree $k-1$ polynomials will have a degree that is too large. Hence, to reduce the degree of the resulting polynomial, we introduce the rule that we may write $X^k = P(X)$ where $\deg P \leq k-1$. As it turns out, when the associated polynomial $Q(X) = X^k - P(X)$ is irreducible (over $\mathbb{F}_p$), the resulting structure forms a field. An equivalent construction of this field is to let ζ be a root of $Q(X)$ and define $\mathbb{F}_{p^k} = \mathbb{F}_p[\zeta]$. This construction is known as a *Galois field*, named after the mathematician Évariste Galois. As an interesting additional note, any base field can be extended in a similar way. For instance by taking an extension of $\mathbb{F}_{2^2} = \mathbb{F}_2[\zeta]$ with $Q(X) = X^2 + X + (\zeta + 1)$, one constructs an alternative representation of the field $\mathbb{F}_{2^4}$. From this, one can additionally conclude that a homomorphism from any $\mathbb{F}_{p^k}$ to $\mathbb{F}_{p^m}$ exists whenever k is a divisor of m.

2.2 Finite Rings of the Form $\mathbb{Z}_{p^k}$

Our initial motivation to look at finite fields was to enable ourselves to implement everything in bounded storage on a computer. Computers, however, generally work over bits and bytes. While a byte can be used to represent an element of $\mathbb{F}_{2^8}$ for instance, that is not generally how arithmetic is performed, as suddenly addition is the same as a bitwise *XOR* operation, and multiplication becomes entirely disjoint from most forms of intuition a programmer may have. Instead, it is more accurate to consider computers as performing arithmetic modulo 2^k, for some values of k, most commonly 32 and 64.[3] However, as we previously established, $\mathbb{Z}_{2^k}$ does not form a field for any $k > 1$. In particular, no even element of $\mathbb{Z}_{2^k}$ is invertible (see condition (e) of definition 2.1). Related to that fact, we observe that all even elements are so-called *zero divisors*.

Even though this structure is not as well-behaved as a field, we will still want to be able to work with them. In particular, several cryptographic protocols that deal with computation — refer also to the next chapters — will want to have some form of compatibility with $\mathbb{Z}_{2^{64}}$ in order to emulate how a "normal" computer operates, and how most programmers think about computation. This compatibility is usually achieved via simulation by using only elements in some other field. While this approach is functional and correct, it also incurs an extra computational overhead that we would like to avoid by constructing protocols that work natively over this structure, despite the extra difficulties. Therefore, to be able to talk about these sorts of structures and analyse their properties, we again define an abstract mathematical structure that encompass the important aspects: a (finite) *commutative ring*.

2.2.1 Definitions

Rather than directing our attention to purely the structure of $\mathbb{Z}_{2^k}$, we consider the slight generalization to $\mathbb{Z}_{p^k}$, for some prime p and integer $k > 1$. For most cryptographic applications, we end up working in one of two situations. Either we take full advantage of the properties of fields and work in $\mathbb{F}_p$, or we work in $\mathbb{Z}_{p^k}$ for some small p — very often $p = 2$ — with a moderately sized k.

Definition 2.3: Commutative ring

A commutative ring R is a tuple $(S, +, \cdot)$, such that it satisfies all axioms of definition 2.1, other than (e). Sometimes the requirement that $0 \neq 1$

[3]Note however that this does not cover bitwise operations.

is also omitted.

Observe hence that every field is automatically a commutative ring, and all $\mathbb{Z}_{p^k}$ are commutative rings. Furthermore, whereas $\mathbb{Z}$ does not form a field, it can be verified that it is a commutative ring.

Mirroring the structure-preserving maps we defined for fields, we can analogously define homomorphisms and isomorphisms for rings too.

Definition 2.4: Homomorphism of (commutative) rings

A homomorphism (of commutative rings) is a structure-preserving map from one commutative ring $R_1 = (S_1, +, \cdot)$ to another $R_2 = (S_2, \oplus, \odot)$. It is a function $\varphi : R_1 \to R_2$, such that $\varphi(1_1) = 1_2$, $\varphi(a+b) = \varphi(a) \oplus \varphi(b)$ and $\varphi(a \cdot b) = \varphi(a) \odot \varphi(b)$.

We already alluded to even numbers in $\mathbb{Z}_{2^k}$ being able to multiply together to zero. For this property, we can also provide a full definition.

Definition 2.5: Zero divisor

An element $a \in R$ in a commutative ring R is a zero divisor when an element $b \neq 0 \in R$ exists, such that $a \cdot b = 0$.

It is easy to see that any element divisible by p is a zero divisor in $\mathbb{Z}_{p^k}$, as multiplying by p^{k-1} will always result in a multiple of p^k, which is congruent to zero in the ring. Conversely, we can see by Bézout's identity that all other elements are invertible and hence not a zero divisor. An invertible element $a \in R$ is also called a *unit*. A unit cannot be a zero divisor, but lacking an inverse element does not automatically make for a zero divisor. Consider $2 \in \mathbb{Z}$ as an example of a number that is not a unit — after all, 2^{-1} does not exist in $\mathbb{Z}$ — but is not a zero divisor.

2.2.2 Extensions

Just like we built larger fields starting from a field, we can build larger commutative rings starting from a commutative ring. We will look at two concrete ways of achieving this, the respective utility of each method becoming clear in the upcoming section (2.3) on probabilistic testing of the equality of sequences over rings and fields. First, we mirror the construction of field extensions to arrive at *Galois rings*. Afterwards, we modify $\mathbb{Z}_{p^k}$ directly by adding extra "digits" to arrive at $\mathbb{Z}_{p^{k+s}}$. Both approaches can also be further combined, leading to a clean combination of their respective properties. We do hence not have to elaborate further on this composition after discussing the individual approaches.

Similar to how we constructed Galois fields, we can take polynomials of degree $d-1$ over $\mathbb{Z}_{p^k}$ modulo a monic irreducible polynomial $Q(X)$ to construct a *Galois ring* $GR(p^k, d)$. This has the same effect as before, where it enables us to rewrite any term with a coefficient greater than d by substituting $X^d = P(X) = X^d - Q(X)$. The values in $\mathbb{Z}_{p^k}$ form a subset of those in $GR(p^k, d)$, where the arithmetic operations between the two are compatible, and so we call $\mathbb{Z}_{p^k}$ a *subring* of $GR(p^k, d)$. Consequently, there exists a trivial homomorphism mapping $\mathbb{Z}_{p^k}$ onto $GR(p^k, d)$.

The other way in which we can make our rings bigger is by adding extra "digits". That is, we work in $\mathbb{Z}_{p^{k+s}}$ for some $s > 0$ to compute or ensure properties of $\mathbb{Z}_{p^k}$.[4] The reason we can do this and retain correct results over $\mathbb{Z}_{p^k}$ is that the "truncation" operation — that is, we cut off the s extra digits we added — is a proper ring homomorphism mapping $\mathbb{Z}_{p^{k+s}} \to \mathbb{Z}_{p^k}$.

For any homomorphism $\varphi : \mathbb{Z}_{p^k} \to R$, we can derive the related homomorphism $\psi : GR(p^k, d) \to R[X]/\langle\varphi(f)\rangle$, by applying φ componentwise. As long as the defining polynomial $f(X)$ of $GR(p^k, d)$ is irreducible over R, we could write — with some abuse of notation — $\psi : GR(p^k, d) \to GR(R, d)$. The utility of this becomes apparent when we now consider this in the context of the truncation homomorphism: the homomorphism $\mathsf{trunc}_k : \mathbb{Z}_{p^{k+s}} \to \mathbb{Z}_{p^k}$ gives us a related homomorphism $\mathsf{trunc}'_k : GR(p^{k+s}, d) \to GR(p^k, d)$. For the specific case of trunc'_1, this means that the reduction of $GR(p^k, d)$ modulo p is equivalent to the finite field $\mathbb{F}_{p^d}$.

2.3 Testing Equality

Like we hinted at earlier, in many cases, we want to devise ways in which to compare sequences of values over some ring or field to each other, without necessarily comparing every element of one sequence directly with the corresponding element in the other. A common reason to do this would for example be a situation where multiple parties have some part of the values — a share of the secret, as will be further explained in chapter 3 — on which they can perform linear operations, but no comparisons. In such a situation, the actual comparison would require communication between the parties, which is

[4]We sometimes call this a *p-adic* extension, due to its apparent similarity to truncations of the p-adic integers.

often costly. Instead, the parties will first want to compress the two sequences into some shorter form, often a single element, by making use of some random compression. If the results are equal, we will then want to have only a small probability that the initial sequences were in fact not equal. Such small probabilities can then be further boosted through standard application of repetition to be entirely negligible.

In this section, we examine two such compression techniques. The first one will be based on polynomial evaluation, while the second one will deal with random linear combinations of the sequence elements.

2.3.1 Equality of Polynomials

We first recall that a polynomial $f(X) \in R[X]$ with coefficients taken from some ring R can be written as $f(X) = f_0 + f_1 \cdot X + f_2 \cdot X^2 + \cdots + f_d \cdot X^d$, where we call d the degree of f, $\deg f$. When interpreted as a function, we can look at f as the function $f : R' \to R'$, where we would commonly take $R' = R$, but any ring such that R is a subring of R' is acceptable.

To represent a sequence $(x_1, \ldots, x_n) \in R^n$ as a polynomial in $R[X]$, we have two common approaches. Both approaches allow computing the polynomial with only linear operations, so either fits the context we sketched above. For the first case, we can simply take each element as a coefficient of the polynomial, and end up with the degree $n-1$ polynomial $f(X) = x_1 + x_2 \cdot X + \cdots + x_n \cdot X^{n-1}$. In the other case, we can embed the sequence elements as evaluations of the polynomial, such that $f(\alpha_1) = x_1, f(\alpha_2) = x_2, \ldots$ for some n fixed, distinct elements α_i. This process of computing a polynomial given a set of evaluations is known as *polynomial interpolation*.

In order to see why interpolation is possible using only linear operations over the values x_i, we first describe how to evaluate a polynomial at multiple values with the Vandermonde matrix. If you fix a constant value $\alpha \in R$ and evaluate $f(\alpha)$, it is clear that all powers of α are also constant, and this evaluation can be represented as a linear combination of the coefficients of the polynomial: $f_0 + \alpha \cdot f_1 + \cdots + \alpha^d \cdot f_d$. As such, when we fix $d+1$ distinct values α_i and consider the evaluation of f in each of these points, we can write this as the matrix-vector product

$$\begin{bmatrix} f(\alpha_1) \\ f(\alpha_2) \\ \vdots \\ f(\alpha_{d+1}) \end{bmatrix} = \begin{bmatrix} \alpha_1^0 & \alpha_1^1 & \alpha_1^2 & \cdots & \alpha_1^d \\ \alpha_2^0 & \alpha_2^1 & \alpha_2^2 & \cdots & \alpha_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{d+1}^0 & \alpha_{d+1}^2 & \alpha_{d+1}^2 & \cdots & \alpha_{d+1}^d \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_d \end{bmatrix}.$$

This matrix is a constant depending solely on our choices for the evaluation points α_i, and is known as the *Vandermonde matrix* $V(\alpha_1, \dots, \alpha_{d+1})$.

It is known that the Vandermonde matrix has an inverse when its evaluation points α_i form an *exceptional sequence*. From there, it is enough to observe that multiplication by the inverse of the Vandermonde matrix results in a polynomial that evaluates to the given values at these points.

Definition 2.6: Exceptional Sequence

An *exceptional sequence* (of length n) in a commutative ring R is a sequence of values $(\alpha_1, \dots, \alpha_n)$, such that for all pairs α_i, α_j (for $i \neq j$), the value $\alpha_i - \alpha_j$ has an inverse in R.

For a field, any sequence of n distinct elements is automatically an exceptional sequence, and hence interpolation is possible over any field that is sufficiently large to contain n elements. In a ring, this is unfortunately no longer a guarantee. Indeed, for the ring $\mathbb{Z}_{p^k}$, the maximal length of an exceptional sequence is only p, which is particularly bad for some situations of interest like $\mathbb{Z}_{2^{64}}$. For the Galois rings $GR(p^k, d)$, the maximal length of an exceptional sequence is p^d, which we can achieve for example by choosing representatives of the field $\mathbb{F}_{p^d}$. This can be formalized with the following lemma.

Lemma 2.1

The Galois ring $R = GR(p^k, d)$ contains a maximal exceptional sequence of length p^d.

Proof.
Existence Consider the reduction of R modulo p, i.e. $\mathbb{F}_{p^d}$. Since this is a field, it is an exceptional sequence of length p^d. Any choice of representatives of this field in R will result in an exceptional sequence of the same length, since any element of R is invertible when its reduction modulo p is.

Maximality Assume for the purposes of contradiction an exceptional sequence of length $n > p^d$. Then two elements, say $\alpha, \beta \in R$, will have the same reduction modulo p,

$$\alpha \equiv \beta \pmod{p} \Rightarrow \alpha - \beta \equiv 0 \pmod{p}.$$

Since their difference has no inverse in $\mathbb{F}_{p^d}$, no inverse can exist in R either. $\square$

Before we return to our goal of testing the equality of two polynomials, we recall one more fact about polynomials over fields.

Lemma 2.2

A non-constant-zero polynomial f of degree $\deg f = d$, over a field, has at most d roots.

Proof. Assume that we have $d+1$ distinct roots $(z_1, z_2 \ldots, z_{d+1})$. Then we can interpolate f by computing $(V(z_1, z_2, \ldots, z_{d+1}))^{-1} \cdot \mathbf{0} = \mathbf{0}$, and arrive at the contradiction that f is the zero polynomial. □

Now we are ready to state a result, known as the *Schwartz-Zippel* lemma, over fields, that will give us a bound on the probability of two polynomials being equal when they evaluate to the same value at a randomly chosen point.[5]

Lemma 2.3: Schwartz-Zippel for fields

Let $f \neq g$ be polynomials over a field $\mathbb{F}$ of degree at most d. Let $r \in \mathbb{F}$ be uniformly random. Then

$$\Pr_{r \leftarrow \mathbb{F}}[f(r) = g(r)] \leq \frac{d}{|\mathbb{F}|}.$$

Proof. Consider the polynomial $h = f - g \neq 0$. Since neither f nor g has any coefficients of degree greater than d, $\deg h \leq d$. When $f(r) = g(r)$, r is a root of h, of which we know at most d exist, by lemma 2.2. Since r was uniformly random over $\mathbb{F}$, the probability of finding a root of h is hence at most $\frac{d}{|\mathbb{F}|}$. □

When widening our view to the rings we've been considering in this chapter, the proof for lemma 2.2 fails because there may be collections of roots over which we cannot interpolate to the zero polynomial. Indeed, for an arbitrary collection of evaluation points, it may not be possible to interpolate at all. An intuitive reason to see why we may have many more roots for a polynomial over e.g. $\mathbb{Z}_{p^k}$ is to consider an evaluation point $\alpha = \alpha' \cdot p \neq 0$ and the polynomial $f(X) = p^{k-1} \cdot X$. Clearly, $\alpha \cdot p^{k-1}$ is zero by construction, and hence $f(\alpha) = 0$. There are however p^{k-1} possible choices for α', which for non-trivial k clearly exceeds the single possible root we'd have over a field.

[5]The probability of the polynomials being equal when they evaluate to different values is, of course, zero.

Lemma 2.4: Schwartz-Zippel for rings

Let $f \neq g$ by polynomials over a ring R with an exceptional sequence S. Let $r \in S$ be uniformly random. Then

$$\Pr_{r \leftarrow S}[f(r) = g(r)] \leq \frac{d}{|S|}.$$

This lemma can be seen to be a generalization of the case for fields. Its proof is analogous to the one for lemma 2.3, so we do not repeat it here.

In practice, we can see that the probability of incorrectly determining that two different polynomials f and g are identical over $GR(p^k, d)$ is entirely independent of k, and hence that larger k makes elements larger without noticeable advantage towards equality testing. Hence, we usually try to avoid large p-adic extensions when applying Schwartz-Zippel equality tests.

A final application of testing polynomial equality that should be noted is concerned with testing equality of (multi)sets. The goal here is to construct a polynomial that can represent a set of values without caring about their order. Luckily, the roots of a polynomial form a set, so if we construct a polynomial $f(X) = \prod_i (X - x_i)$ that has exclusively our elements as roots, we can achieve this goal. It should be noted, however, that this polynomial cannot be computed by exclusively linear operations.

2.3.2 Random Linear Combinations for Sequence Equality

The second common method for sequence equality testing we discuss will not introduce extra structure to the sequence. We can get some first intuition by starting with a Schwartz-Zippel test, and then removing dependence between the randomness we see. Consider a polynomial $f(X)$ with the sequence's elements as coefficients, and an evaluation of that polynomial in a random value r:

$$f(r) = r^0 \cdot x_1 + r^1 \cdot x_2 + r^2 \cdot x_3 + \cdots + r^{n-1} \cdot x_n.$$

With an appropriate shift in viewpoint, this is simply a linear combination of the sequence elements with some random values r^i that happen to not be independent. If we replace all r^i with independent values r_i instead, we can analyse the resulting probability of compressing two distinct sequences to an identical value. Equivalently, and to simplify the analysis, we can look at the equality of two sequences $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ as having all of $(x_1 - y_1, \ldots, x_n - y_n)$ equal to zero. In a field, we then get the following result.

Lemma 2.5: Random linear combinations for fields

Let $(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n$, with at least one $x_i \neq 0$. With $(r_1, r_2, \ldots, r_n)$ uniformly random over $\mathbb{F}^n$, the probability

$$\Pr_{r \leftarrow \mathbb{F}^n}[r_1 \cdot x_1 + r_2 \cdot x_2 + \cdots + r_n \cdot x_n = 0] \leq \frac{1}{|\mathbb{F}|}.$$

Proof. Let i be an index such that $x_i \neq 0$. Then we can write the condition for the linear combination to be zero as

$$r_i = -\frac{\sum_{j \neq i} r_j \cdot x_j}{x_i},$$

where the inverse of x_i exists due to $x_i \neq 0$. Since r_i is uniformly random over $\mathbb{F}$, the probability of the right-hand size being equal to r_i is exactly $\frac{1}{|\mathbb{F}|}$. □

When considering this lemma over rings instead, we run into the problem that any non-zero x_i may not be invertible. Instead, we can work over a p-adic extension by s bits instead, working over $\mathbb{Z}_{p^{k+s}}$, while only guaranteeing that the lower k digits are zero, and allowing the upper s digits to be different. This approach is entirely compatible with Galois extensions as well, working over $GR(p^{k+s}, d)$, which in turn makes the space to sample randomness from larger.

Lemma 2.6: Random linear combinations for Galois rings

Let $(x_1, x_2, \ldots, x^n) \in GR(p^{k+s}, d)^n$, with at least one $x_i \not\equiv 0 \pmod{p^k}$. With $(r_1, r_2, \ldots, r_n)$ uniformly random over $GR(p^{s+1}, d)^n$, the probability

$$\Pr_{r \leftarrow \mathbb{F}^n}[r_1 \cdot x_1 + r_2 \cdot x_2 + \cdots + r_n \cdot x_n = 0] \leq \frac{1}{p^{d \cdot (s+1)}}.$$

Proof. Let i be an index such that $x_i \not\equiv 0 \pmod{p^k}$, and $w < k$ maximal such that p^w divides x_i. Then we can write the condition for the linear combination to be zero as

$$r_i \equiv -\frac{\sum_{j \neq i} r_j \cdot x_j}{p^w} \cdot \left(\frac{x_i}{p^w}\right)^{-1} \pmod{p^{k+s-w}},$$

where the inverse now exists thanks to the maximality of w. Since r_i is uniformly random and $k + s - w \geq s + 1$, the probability becomes exactly as claimed. □

We can observe here that only extending to a Galois ring results in a slightly better probability, by the disappearance of the factor n in the numerator, as compared to the Schwartz-Zippel check. The extension to a Galois ring is however far more expensive in terms of the amount of data needed to represent a single element, when compared to the p-adic extension. Hence, extending p-adically can provide more value here, in contrast to the situation when dealing with polynomials.

CHAPTER 3

Multiparty Computation from Linear Secret Sharing

In this chapter, we first describe secret sharing: a way to distribute or share information with multiple people such that no single person can gain any further knowledge about it. We then cover a few general methods in which multiple people can come together to jointly perform private computation that does not reveal any person's inputs to another. Finally, we describe ways in which such private computation can be instantiated based on secret sharing.

3.1 Linear Secret Sharing Schemes

The first step to take before going directly into secure distributed computations, is to consider how the data for these computations will be represented among all participating parties. Since our goal is to avoid any single party — and as we shall discuss, this even extends to entire sets of parties — learning anything about this data in the intermediate steps, the information should be spread across all participants, in such a way that everyone can contribute some part, that is meaningless on its own. To tease some notation without fully defining it just yet, we shall write a sharing of a secret value s as $[s]$, which you could interpret visually as putting s in a (distributed) box.

In order to do so, we discuss *Secret Sharing Schemes* in this section, and in particular a specific kind known as *Linear Secret Sharing Schemes (LSSS)*. We shall define these in an abstract sense, based on the *Access Structures* they allow. That is, which sets of parties are allowed or disallowed to learn the shared secret. From there, we build a concrete representation of LSSS in the form of *Monotone Span Programs (MSP)*, which will enable us to use tools from linear algebra to manipulate the associated LSSS.

Consider the following example. We will come back to this setting a few times in this chapter to further clarify newly introduced concepts.

Example 3.1

The queen of a far-off realm wishes to ensure the future wealth of the kingdom, and has secured the crown jewels along with a large amount of gold in a vault. This vault is protected behind several traps and a secret code, which, if entered incorrectly, will result in the entire treasure being destroyed on the spot. She is currently the only person to know the code, but wants to make sure that all wealth will not be lost in the event of her death, so she looks for a way to share the code between her three children (the crown prince *C*, princess Alice *A* and prince Bob *B*) and the castle-keeper *K*. Since the treasure should only be accessed after her death, she wants this sharing to be in such a way that C needs at least one other person with him to open the vault, or A, B, and K have to come together (just in case something happens to the crown prince too).

3.1.1 Access Structures

Two of the main characteristic properties we want from secret sharings are privacy, certain sets of shares do not reveal any information about the shared secret, and being able to reconstruct, where at least *some* set of shares can recover the shared secret by working together. We can formalize the sets corresponding to these properties as respectively *unqualified sets* and *qualified sets*.

Observe that simply based on availability of information, if a set S is a superset of some qualified set Q, S is qualified too, and similarly any subset T of an unqualified set U must be unqualified. For qualified sets, we call this *monotonically increasing*, and for unqualified sets *monotonically decreasing*.

Definition 3.1: Monotonicity

Let S be a set and $\Theta \in 2^S$. We call Θ monotonically *increasing* when $T \in \Theta \land T \subseteq R \subseteq S \implies R \in \Theta$. We call Θ monotonically *decreasing* when $T \in \Theta \land R \subseteq T \subseteq S \implies R \in \Theta$.
Since the subset relation defines a partial order, we can identify minimal elements of a monotonically increasing set and maxima of a monotonically decreasing set. In this context, we refer to those as *minimally qualified sets* and *maximally unqualified sets* respectively.

Definition 3.2: Access Structure

Let $\mathcal{P}$ be a set of parties (or equivalently shares). When $\Gamma, \Delta \in 2^{\mathcal{P}}$, $\Gamma \cap \Delta = \emptyset$, Γ monotonically increasing, Δ monotonically decreasing, we call (Γ, Δ) an *Access Structure* (for $\mathcal{P}$). The elements of Γ are called qualified sets, while the elements of Δ are called unqualified sets.

For ease of use and notation, we will always assume so-called *complete* access structures. That is, every possible set of shares is in exactly one of Γ and Δ. Any access structure that is not complete can be turned into a complete access structure by arbitrarily assigning every ambiguous set of shares to either of the two sets.

Example 3.2

Recall the setting from example 3.1. We can denote the set of parties as $\mathcal{P} = \{C, K, A, B\}$, and interpret the queen's wishes as requiring the qualified sets to be $\Gamma \supseteq \{\{C, K\}, \{C, A\}, \{C, B\}, \{K, A, B\}\}$, i.e.

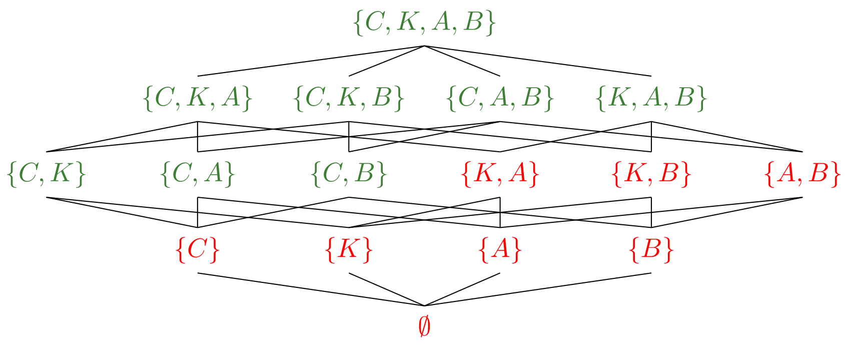
the minimally qualified sets. Additionally, Γ should be monotonically increasing, so we obtain

$$\begin{aligned}\Gamma = \{&\{C,K\},\{C,A\},\{C,B\},\\ &\{K,A,B\},\{C,K,A\},\{C,K,B\},\{C,A,B\},\\ &\{C,K,A,B\}\}.\end{aligned}$$

Since we wish to work with a complete access structure, the unqualified sets then become

$$\begin{aligned}\Delta &= 2^{\mathcal{P}} \setminus \Gamma\\ &= \{\emptyset,\{C\},\{K\},\{A\},\{B\},\{A,B\},\{K,A\},\{K,B\}\},\end{aligned}$$

and (Γ, Δ) is the access structure we want to achieve.
To visualize an access structure, it can sometimes be enlightening to look at the lattice structure inherent to the subset relation, and see that green (being qualified) propagates upwards, while red (being unqualified) propagates downwards.



When dealing with more parties, explicitly writing out the access structure becomes infeasible, as we would need to specify 2^N sets (with $N = |\mathcal{P}|$). Even restricting this to only describing the minimally qualified sets can still lead to an exponential number of sets to specify. Therefore, we often infuse some extra structure. The most common occurrence of this is in t-threshold access structures, where all sets with more than t parties are qualified. For values of $t < \frac{N}{2}$, this is also referred to as *honest majority*.

Definition 3.3: t-Threshold Access Structure

Let (Γ, Δ) be an access structure. It is called a *t-threshold access structure* when

$$X \in \Gamma \iff |X| > t.$$

Another structural property that is commonly encountered is a $\mathcal{Q}_\ell$ *access structure*. In contrast to threshold access structures, $\mathcal{Q}_\ell$ structures do not provide a description of the access structure, but rather a condition to be fulfilled. Specifically, it is concerned with how many unqualified sets in Δ it takes to obtain the entire set of parties $\mathcal{P}$.

Definition 3.4: $\mathcal{Q}_\ell$ Access Structure

An access structure (Γ, Δ) for $\mathcal{P}$ is a $\mathcal{Q}_\ell$ access structure when no union of ℓ sets $X_i \in \Delta$ covers all of $\mathcal{P}$.

From this definition, it is clear that any $\mathcal{Q}_\ell$ access structure is also $\mathcal{Q}_k$ for $k \leq \ell$. Additionally, this condition subsumes the family of threshold access structures where $t < \frac{N}{\ell}$, as in that case $\ell \cdot t < N$ and no ℓ unqualified sets can cover all of $\mathcal{P}$ by cardinality. In this thesis, the $\mathcal{Q}_2$ condition will be of central importance, as it is a natural generalization of the honest majority setting by the earlier argument.

Example 3.3

Resuming our running example, observe that while the access structure is not a threshold structure and hence not "honest majority", it does satisfy the $\mathcal{Q}_2$ property. By contrast, it is not $\mathcal{Q}_3$ (or indeed $\mathcal{Q}_k$ for $k > 2$), as for instance the three unqualified sets $\{K, A\}$, $\{K, B\}$ and $\{C\}$ have a union equal to $\{C, K, A, B\}$.

3.1.2 LSSS for a Given Access Structure

Having defined access structures, we are now ready to give a definition of a secret sharing scheme. For now, it will be mostly an abstract definition of what it *does*, without any details on the *how*. In the next section, we then cover a concrete mathematical construction that provides instantiations for any *linear* secret sharing scheme.

Definition 3.5: Secret Sharing Scheme (SSS)

Let $\mathcal{P} = \{P_i \mid 1 \leq i \leq N\}$ be a set of parties. A secret sharing scheme consists of the following two functionalities:

Share: takes as input some secret value s and outputs share s_i to party P_i. s_i need not be a single value, it could also be a vector of values. We also write $(s_i)_i = [s]$, with the understanding that the shares are held in a distributed manner.

Rec: A subset of parties $S \subseteq \mathcal{P}$ inputs their shares, and *reconstructs* the shared secret. The output is either s, if the S is qualified, or $\perp$ if S is unqualified.

If (Γ, Δ) is an access structure for $\mathcal{P}$, every set in Γ is qualified for Rec, and every set in Δ is unqualified for Rec, the secret sharing scheme is said to *realize* the access structure (Γ, Δ).

Commonly — and especially further on in this chapter — we not only want to *share* some data, we also want to be able to perform some computation on the shared data, without having to reveal it. Therefore, we restrict our view to a specific family of secret sharing schemes: *Linear Secret Sharing Schemes*. As the name already implies, this family of sharing schemes will allow us to compute linear functions over secret shared values without any communication beyond the initial sharing. That is, we want to achieve a linear secret sharing scheme such that $\alpha \cdot [x] + \beta \cdot [y] = [\alpha \cdot x + \beta \cdot y]$.

To achieve this property, we only require a single restriction to the reconstruction process Rec, namely that it is a linear combination of the input shares. Concretely, this means that for every qualified set $Q \in \Gamma$, there is vector λ_Q such that if we let $\mathbf{s}_Q$ represent the shares collectively held by the parties in Q, reconstruction consists of computing only the inner product $s = \langle \lambda_Q, \mathbf{s}_Q \rangle$.[1] A straightforward verification confirms that this restriction suffices to obtain the desired property:

$$\begin{aligned}\mathsf{Rec}(\alpha \cdot \mathbf{x}_Q + \beta \cdot \mathbf{y}_Q) &= \langle \lambda_Q, \alpha \cdot \mathbf{x}_Q + \beta \cdot \mathbf{y}_Q \rangle \\ &= \alpha \cdot \langle \lambda_Q, \mathbf{x}_Q \rangle + \beta \cdot \langle \lambda_Q, \mathbf{y}_Q \rangle \\ &= \alpha \cdot \mathsf{Rec}(\mathbf{x}_Q) + \beta \cdot \mathsf{Rec}(\mathbf{y}_Q).\end{aligned}$$

[1]We let $\mathbf{s}_Q$ be a vector of ring elements here. So if any s_i consists of more than a single element, each element of s_i occurs as a separate element of $\mathbf{s}_Q$.

Definition 3.6: Linear Secret Sharing Scheme (LSSS)

An SSS for which reconstruction is a linear combination of the input shares is a *Linear Secret Sharing Scheme*.

Example 3.4

We briefly consider a sample construction of a secret sharing scheme for the setting of example 3.1. To compute the shares, we first sample some uniformly random values $r_2, t_2, t_3 \in \mathbb{F}$ from the (finite) field over which we are working. Then we compute $r_1 = s - r_2$ and $t_1 = s - t_2 - t_3$. To C we give as share r_1, to K we give the shares (r_2, t_1), to A the shares (r_2, t_2) and to B the shares (r_2, t_3). In this way, we ensure that each of the minimally qualified sets can either compute $s = r_1 + r_2$ or $s = t_1 + t_2 + t_3$, while each of the unqualified sets lacks at least one uniformly random value for both of those computations.

Example 3.5: Shamir Secret Sharing

Leaving behind our running example for a moment, we can also consider a secret sharing scheme that can be instantiated for any t-threshold access structure, as long as the ring of definition allows an exceptional sequence $S = (\alpha_0, \alpha_1, \ldots, \alpha_N)$ of length $N + 1$. For ease of construction, we also assume (without loss of generality) that $\alpha_0 = 0 \in S$. Observe that for the case of $R = \mathbb{F}$ a finite field, this requirement reduces to $\mathbb{F}$ being large enough to contain $N + 1$ distinct values.
For a sharing with N parties and threshold t, each party is publicly assigned a distinct value $\alpha_i \neq 0$ from S. In order to share a value s, a polynomial of degree t

$$f(X) = s + \sum_{i=1}^{t} r_i \cdot X^i$$

is generated with uniformly random coefficients r_i. Each party P_i then receives as share $f(\alpha_i)$. Observe that the secret $s = f(0)$, which is not distributed to any party as per our earlier definition of α_i.
Once $t+1$ or more parties combine their shares, any $t+1$ points uniquely define the polynomial $f(X)$. Interpolation can then recover it and compute $s = f(0)$ for reconstruction. Interpolation and evaluation at 0 can be immediately combined into a single linear combination, showing that the Shamir secret sharing scheme is an LSSS.

3.1.3 Monotone Span Programs

In order to make more mathematically clear statements and proofs later on, it would be good to have an object we can manipulate more liberally, while still keeping an equivalence with the LSSS structure we defined earlier. The linearity properties of LSSS quickly point us towards the use of linear algebra and matrices to describe the secret sharing scheme. We can generate all share values from a single matrix-vector multiplication with a random vector, as long as that random vector satisfies the correct linear relation to allow for reconstruction. Once we have all share values, we assign them to the correct players. This idea is what brings us a formal definition: the *Monotone Span Program.*[2]

Definition 3.7: Monotone Span Program (MSP)

A monotone span program is a 4-tuple $(\mathbb{F}, M, \varepsilon, \psi)$, such that:

- $\mathbb{F}$ is a field
- M is an $m \times d$ full-rank matrix over $\mathbb{F}$, with $d \leq m$
- $\varepsilon \in \mathbb{F}^d$ is a non-zero vector
- ψ is a surjective mapping of $\{1, 2, \ldots, m\} \to \mathcal{P}$ mapping rows of M to parties in $\mathcal{P}$.

If we can give a proper definition of the Share and Rec operations of an LSSS starting from an MSP, this shows every MSP defines (or "induces") an LSSS. To Share a value s, the dealer samples a random vector $\mathbf{r} \in \mathbb{F}^d$, such that the inner product $\langle \mathbf{r}, \varepsilon \rangle = s$. Each party P_i then receives the share vector $\mathbf{s}_i = ((M \cdot \mathbf{r})_j \mid \psi(j) = P_i)_j$. By the surjectivity of ψ, each share vector for each party is non-empty. To Rec the value s given the shares $\mathbf{s}_Q$ from a qualified set of parties Q, consider the matrix $\tilde{M}$ as the matrix formed by the rows of M that ψ maps to parties in Q. We set the recombination vector λ_Q to be such

[2]The "monotone" part of the name comes from the same monotonicity condition that we encountered before when introducing access structures.

that $\tilde{M}^\top \cdot \lambda_Q = \varepsilon$, satisfying

$$\begin{aligned} s &= \langle \varepsilon, \mathbf{r} \rangle \\ &= \langle \tilde{M}^\top \cdot \lambda_Q, \mathbf{r} \rangle \\ &= \left(\tilde{M}^\top \cdot \lambda_Q\right)^\top \cdot \mathbf{r} \\ &= \lambda_Q^\top \cdot \tilde{M} \cdot \mathbf{r} \\ &= \lambda_Q^\top \cdot \mathbf{s}_Q \\ &= \langle \lambda_Q, \mathbf{s}_Q \rangle. \end{aligned}$$

This automatically gives a definition of the access structure realized by the LSSS induced by the MSP: $Q \in \Gamma \iff \varepsilon \in \mathsf{Im}(\tilde{M}^\top)$. The converse direction, constructing an MSP that induces a given LSSS, is not as simple to describe, as an LSSS gives far less structure to work with — which was of course our motivation to introduce MSPs in the first place. The key observation to make is that we can take the condition $\tilde{M}^\top \cdot \lambda_Q = \varepsilon$, fix a value for ε and solve a big linear system for the matrix M.

Also observe that while every MSP induces a unique LSSS, any LSSS may admit a large number of MSPs. Indeed, the choice of ε can be seen to be arbitrary and an equivalent MSP can be obtained for every choice of ε. Similarly, the ordering of the rows of M can be changed, leading to only a change in the definition of ψ to keep the same LSSS.

Example 3.6

We revisit the LSSS from example 3.4 and construct one of the MSPs realizing it. Both the scheme as presented earlier and the MSP we construct now are independent of the field being used, so any choice of $\mathbb{F}$ will work. As mentioned earlier, the choice of ε is somewhat arbitrary, so we will choose the first standard basis vector $\varepsilon = (1, 0, 0, 0)$ for it. This means that our random vector $\mathbf{r}$ will be of the form (s, r_1, t_1, t_2), and that we should construct the matrix M in such a way that they get correct share. The following matrix achieves this goal:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & -1 \end{bmatrix}.$$

Observe here that M contains repeated rows whenever multiple parties receive the same value. For readability, we present the value of ψ as a sequence of players, assigning rows of M from top to bottom: $\psi = (C, K, K, A, A, B, B)$.

Example 3.7

Finally, consider the Shamir secret sharing scheme introduced earlier. If we let the random vector $\mathbf{r}$ represent the coefficients of the polynomial $f(X)$, we require the value of $\varepsilon = (1, 0, 0, \ldots, 0)$, as $f(0) = s$. As secret sharing corresponds to polynomial evaluation at fixed points α_i, the $N \times (t+1)$ matrix M becomes the Vandermonde matrix $V(\alpha_1, \ldots, \alpha_N)$, and the assignment function ψ will map $i \mapsto P_i$. It is easy to verify that the recombination vectors obtained from submatrices of M correspond to those obtained from the polynomial interpolation approach.

3.2 Secure Multiparty Computation (MPC)

Now that we know how to distribute data among multiple parties, it is time to consider how to perform computation with multiple parties. In this section, we will first give a short overview of the general setting: we discuss how to represent computation and “programs” in ways that are amenable to multi-party computation, present a few of the different high-level approaches to build MPC and how to evaluate the performance of MPC protocols. In the next section, we can then have a closer look at how we can use linear secret sharing techniques to enable computation.

3.2.1 Representing Computation

If you ask a software developer how they would represent a piece of computation, they might reply with their favourite programming language. A more theoretically inspired computer scientist may talk about Turing Machines or even lambda calculus. And if you manage to talk to the right kind of mathematician, you could even hear the opinion that computation is irrelevant, because you only need to prove existence or non-existence of a solution.

All of these options go to illustrate that picking a single "correct" representation for computation is not a trivial matter. Generally speaking, there is common structure that is usually chosen for the purposes of MPC: circuits. We can think of a circuit as an abstract representation of what happens in a piece of hardware. There are "wires" that carry values around, and those wires connect the outputs of some "gates" to the input of the next gates. Additionally, we enforce the structure that no gate's output can eventually end up at its own input again. From the viewpoint of a computer scientist, we could say that the gates form a directed acyclic graph.

From this description, the question of what a *gate* actually is or does remains. We can find a first option by reconsidering the inspiration from hardware: boolean gates. In a *boolean circuit*[3], the gates consist of *XOR* and *AND* gates, that, respectively, compute the XOR and the AND operation on their inputs, with all wires carrying single-bit values. It is known — and indeed used in hardware — that any computation can be constructed from a limited choice of gates, be it NAND or XOR+AND for instance. In this particular case, the choice for XOR and AND is more attractive as the XOR gate performs a linear operation over the bits.

A second, though similar, option appears when we try to generalize these boolean circuits to a larger domain. When we consider bits as being values of the finite field $\mathbb{F}_2$, we can frame the XOR operation as addition, and AND as multiplication. Then, letting our field be larger, say $\mathbb{F}_q$, we can build circuits with wires carrying $\mathbb{F}_q$ values, and *ADD* and *MUL* gates computing addition and multiplication over the field respectively. When restricting values to come from $\{0, 1\} \subseteq \mathbb{F}_q$, and letting $a \oplus b = a + b - a \cdot b$, we can recover boolean circuits, hence proving that these *arithmetic circuits* are as powerful.

Additionally, when designing cryptographic protocols, it is oftentimes possible to achieve certain functionalities more efficiently than with a generic decomposition into ADD and MUL gates. If, for instance, you need exponentiations in your protocol, and you find a way to compute those more efficiently than a generic

[3]Sometimes also called a binary circuit.

decomposition into multiplications — or even if you have a way to more efficiently compute squares for the square-and-multiply approach to exponentiation — you could add that as a custom gate or *gadget* to your circuit description.

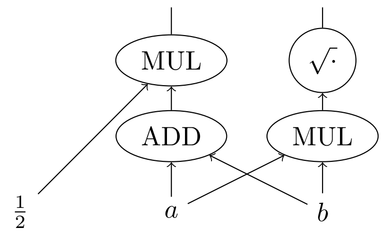
While boolean and arithmetic circuits are able to describe any computation, there are some downsides when you compare them to some other possible descriptions of computations. However, they can be seen to closely align with the "oblivious" evaluation of the computation we aim for in MPC, where the computation can be performed without having to "look under the hood" at the actual values carried by the wires. All you need to evaluate a circuit is a way to evaluate its gates. The other side of that coin unfortunately means that some things become hard or even impossible to express or implement. For example, when some part of the computation is only used conditionally, based on an input or other intermediate value, it becomes impossible to skip the computation when the condition is false, and both paths are taken "simultaneously". It is impossible to tell whether the condition is true or false, after all. An extension of this problem also appears when trying to repeat some computation. Conditional loops run into the same problem as conditions, and without adding extra structure, fixed-length loops result in repeated structure in the resulting circuit description.

As an alternative model, one can also look at virtual machines (VMs) that repeatedly execute a single step of computation, based on the current state of some *register* values, and a single instruction. The evaluation of such an instruction can then for instance be represented by a circuit. Additionally, such a VM[4] might be able to read and write to some random-access memory (RAM), where the accesses are performed with arbitrary values. Much cryptographic research has gone into trying to make this RAM access simultaneously efficient and oblivious.

Example 3.8

As a final example, consider the following arithmetic circuit — enhanced with a gadget to calculate the square root — that takes two inputs and computes their arithmetic and geometric means.

[4]Although with appropriate gadget design, this is also possible in a circuit model.



3.2.2 Flavours of MPC

With circuits in our tool belt and the realization that being able to add and multiply values obliviously is sufficient to compute *anything* obliviously[5], we can now turn our attention to the common settings for MPC. For this, we will consider things like distinguishing which parties provide inputs and which receive the output, which parties — or subsets of parties — might be malicious and work together to learn things from the protocol they should not be able to, in which way parties may cheat, and more. We then also look at the high-level idea behind some of the common forms of MPC and how they correspond to those differences in setting, delaying the discussion of concrete constructions and techniques to the next section.

When multiple parties come together to perform some computation, we can consider how many parties contribute to the protocol, as well as identify some different roles that these parties may perform. To compute something, the first thing we need is data to compute over, so some parties — the *input-bearing parties* — should provide some input to the protocol. Then the computation itself needs to be performed over these inputs, which the *computing parties* do. And finally the output should be revealed, to the *output-receiving parties*. These roles need not be exclusive, and indeed, more than just occasionally, all parties will take up all three roles within the same protocol.

MPC protocols can also differ in their adversary model: what a theoretical attacker is “allowed” to do — usually by *corrupting* some of the parties, hence gaining knowledge of their internal state and influencing them to behave in certain ways — and what guarantees the protocol offers under those conditions. This can usually be classified along some orthogonal axes. On the first axis, we can distinguish a *passive* (sometimes also called *semi-honest* or *honest but curious*) adversary from an *active* adversary. A passive adversary will correctly

[5]As long as we have our inputs in *some* right form.

follow the protocol, but try to learn secrets from the communication it can observe and the data held by the corrupted parties, while an active adversary can freely deviate from the protocol, and for instance learn secret information by introducing errors and observing how those affect the public communication. In the case of an active adversary, we can then distinguish whether we want to simply detect their presence and stop the protocol (called *active security with abort*), or be able to correct for the malicious behaviour and complete the protocol with the correct output regardless (known as *guaranteed output delivery*).

On a second axis, we can determine how the adversary is allowed to choose the parties to be corrupted. For *static* corruptions, the adversary has to make a choice at the start of the protocol, but in the case of *dynamic* adversaries, they can make the decision to corrupt a party based on the observed communication or data gathered from previous corruptions. In some cases, we could even allow parties to only be temporarily corrupted, after which the adversary can no longer influence the party, but may choose to corrupt a different one. Additionally, and this could be considered as a third orthogonal axis, we often want to restrict the possibilities of parties that could be corrupted by the adversary. After all, if every party is corrupted, it is trivially impossible to have any security guarantees. As such, we describe an access structure (such as in definition 3.2) that tells us which sets of corruptions should guarantee security. In some cases, this could be enhanced to have different access structures for active and passive corruptions. We then of course have similar terminology concerning these access structures as before: t-threshold structures, $\mathcal{Q}_\ell$ conditions and honest majority all still apply.

Finally, it is important to be aware of the "environment" in which the parties interact. When the network can be relied upon to deliver sent messages within a certain timespan, we speak of a *synchronous* network, and we can build simpler protocols that rely on this property. On the other hand, in an *asynchronous* network, we need to take into account that messages may be lost or severely delayed somewhere along the way to the recipient. Related to this, the availability of a *broadcast channel* — where all (honest) parties are guaranteed to receive the same message — can drastically change the protocol design and the theoretical guarantees that can be made.

3.2.3 Cost and Efficiency

When choosing a protocol for an application, one has to be able to compare it to alternatives and decide which is better for the given situation. During the design of a protocol however, any concrete application may not yet be known. It is nevertheless necessary to already have some metrics that will commonly affect the concrete performance and enable comparisons between different protocols.

As a first intuition, it might be tempting to analyse the computational complexity of an individual computing party and try to minimize that. While it should of course not be entirely neglected, this measure completely ignores the fact that the computing parties need to communicate over the network, and form a distributed system. Therefore, we should investigate the impact of any protocol on the amount of communication required. We use two complementary measures for this, the relative importance of which will be affected by the type of network the protocol is deployed in. In networks with low latency, a single round of communication — in which every party potentially sends a message to every other party — may not take up much time, leaving the majority of the cost of communication to depend on the amount of data being sent. On the other hand, if the impact of the network latency becomes more pronounced, the plain cost of having an extra round of communication may well justify reducing the number of rounds at the cost of having to send more bits over the wire overall. In practice, minimizing the number of communication rounds and the amount of communication will indeed also work towards having smaller computational complexity. Most operations performed tend to be fairly simple; and having less communication means that less data needs to be computed before being sent.

With a focus on optimizing for certain classes of applications, it is sometimes also possible to define a more narrow metric, and optimize even more towards that. The most common example for this is the *preprocessing model*, in which the parties can start executing the protocol *before* they know their inputs or even the specific computation to be performed. Their goal is then to precompute useful, but random, data so that the *online* phase of the protocol — once the inputs and computation are known — can be performed as fast as possible. The cost of the preprocessing phase is of lesser importance compared to the speed at which results are known once the actual computation can be performed. We can again count the number of communication rounds and the amount of communicated data as measures for the efficiency of this online phase. In the next section, we will see multiplication triples as an example of useful preprocessing data.

3.3 Building MPC from Secret Sharing

Different "families" of MPC exist, and can serve different purposes, as well as achieve different performance characteristics. One approach depends on *Fully-Homomorphic Encryption* (FHE), which enhances ciphertext with the capability of obliviously performing ring operations on the underlying plaintext. The final ciphertext can then be sent to the output-receiving parties who have the corresponding decryption key. While this enables very efficient data and round complexity — after all, only the inputs and the output need to be communicated — it has the downside that performing the homomorphic computation is computationally rather expensive, and the computing party must be distinct from the output-receiving parties, as otherwise they could decrypt the inputs immediately.

Another approach, which is specialized for 2-party computation, and also achieves a constant number of rounds, is that of *garbled circuits*. Here, one of the two parties acts as the "garbler", and obfuscates (garbles) the circuit under consideration, with their own inputs baked in. Then the second party acts as the "evaluator" and uses the information received from the garbler to compute the circuit on their own inputs, and finally obtain the output.[6]

And finally, there is MPC based on LSSS, which supports an arbitrary number of parties and which will be the focus for the rest of this chapter. Traditionally, all parties are considered as simultaneously input-bearing, computing and output-receiving, but the roles can be distributed differently without much trouble. All parties secret-share their inputs such that everyone holds a share of all values in the computation at all times. Linear operations can be performed locally, as the secret sharing scheme is linear, but communication is required to evaluate multiplications. As a result, the number of required rounds tends to scale with the *multiplicative depth* of the circuit, and the data complexity with the total number of multiplications. To obtain the final output, each party can broadcast their shares to all other parties, and everyone can then locally perform the reconstruction of the LSSS.

3.3.1 Passive Security

As mentioned before, the main challenge when building an MPC protocol from an LSSS is to achieve multiplication. In this section, we briefly present a few common techniques, specialized to the Shamir secret sharing scheme (see example 3.5) with honest majority and a semi-honest adversary. A more elaborate treatment for general $\mathcal{Q}_2$ access structures can be found later on in Chapter 5.

Consider what happens when the parties have shares $[a]$ and $[b]$, and compute the local product of their shares. Recall that $a_i = f(\alpha_i)$ and $b_i = g(\alpha_i)$, while

[6]In practice, to have inputs provided by the evaluator, we additionally need to use a cryptographic primitive known as *oblivious transfer*.

the secrets $a = f(0)$ and $b = g(0)$ are embedded in the constant term of the polynomials. The products $c_i = a_i \cdot b_i = f(\alpha_i) \cdot g(\alpha_i) = (f \cdot g)(\alpha_i)$ now lie on the degree $2t$ polynomial $h = f \cdot g$. This polynomial does satisfy that $h(0) = (f \cdot g)(0) = a \cdot b$, as we would want from a multiplication operation, but importantly, is no longer random and hence not perfectly secure,[7] nor indeed of the correct degree. This means that at least $2t + 1$ parties are required to reconstruct the underlying value — which is still possible thanks to the assumption of honest majority — and we could only perform multiplications up to a depth of 1 without losing the information required to reconstruct. Therefore, we aim to have a step of communication that will simultaneously reduce the degree of this polynomial h back to t and re-randomize its coefficients (other than the constant term) such that it becomes a "regular" secret-shared value again.

A first possible approach is due to [Mau06]. It relies on the fact that a reconstruction for the degree $2t$ polynomial h exists, using only linear combinations of the shares. While the product cannot be publicly reconstructed without violating privacy, the reconstruction can happen in the basic t-threshold Shamir scheme. Every party takes their value $a_i \cdot b_i$ and shares it to all parties in the base LSSS. Since the reconstruction of h is linear, it can be computed as a linear function

$$[a \cdot b] = \mathsf{Rec}_{2t}([a_1 \cdot b_1], \ldots, [a_N \cdot b_N]).$$

All "re-sharings" were random and of degree (at most) t, hence the resulting share of the product $a \cdot b$ will be of the correct degree and fully random.

An alternative approach, due to [DN07], uses the same properties, but additionally consumes some independently generated random pairs $([r]_t, [r]_{2t})$. Here we use the notation $[x]_d$ to represent a random degree d polynomial with x as constant coefficient. These pairs can be generated in a preprocessing phase with some techniques based on so-called *superinvertible* matrices, such as the Vandermonde matrix. Given such a pair and the values $a_i \cdot b_i$, one can interpret $a_i \cdot b_i - r_{2t,i}$ as a random degree $2t$ sharing of $\sigma = a \cdot b - r = \mathsf{Rec}([a \cdot b]_{2t} - [r]_{2t})$. Thanks to the randomness of r, this value σ reveals no information, and can safely be publicly reconstructed. Given σ, the parties can then locally compute $[a \cdot b]_t = [r]_t + \sigma$, thus obtaining a uniformly random share of degree t of the product $a \cdot b$. To ensure privacy and avoid linear relations between products from leaking, no such random pair $([r]_t, [r]_{2t})$ should be used more than once.

As a final common approach, we present the idea from [Bea92]. Here, we no longer compute the local product, but instead rely on randomness from a preprocessing phase to directly enable multiplication through linear operations

[7]For example, h cannot be irreducible.

and a single round of reconstruction. This means that, as long as the preprocessing functionality can be achieved, the technique works for arbitrary access structures, including a full threshold structure. From the preprocessing, we assume that we can obtain *random* multiplication triples[8] $([x], [y], [z] = [x \cdot y])$. Such triples could be generated in several of different ways, including the above approaches for $\mathcal{Q}_2$ access structures. When given such a triple $([x], [y], [z])$ and two values $[a]$ and $[b]$ to multiply, the parties open two values $\rho = [a + x]$ and $\sigma = [b + y]$. Observe that both values are uniformly random as they are masked by the uniform random values x and y respectively. With these values, the parties can then locally compute

$$[c] = [z] + \rho \cdot [b] + \sigma \cdot [a] - \rho \cdot \sigma.$$

One can confirm that indeed

$$a \cdot b = x \cdot y + (a + x) \cdot b + (b + y) \cdot a - (a + x) \cdot (b + y).$$

Here too, the preprocessing material should not be reused between multiple multiplication calculations, as that would reveal linear relations between the inputs.

3.3.2 Security with Abort

Now that we have some approaches to compute multiplications in a semi-honest setting, the natural next step is to see if they can be transformed to deal with active adversaries. Our first target is security with abort: the parties stop executing the protocol when any malicious behaviour is detected, and before any information can leak towards the adversary.[9] We present some general ideas here, and revisit the topic for a more technical overview as applied to $\mathcal{Q}_2$ access structures in chapter 5.

In contrast to the case for passive security, there are now several subprotocols where we need to watch out for malicious behaviour, rather than only the multiplication. The first one of these is the sharing of the inputs. Here the adversary could choose to distribute shares that are not consistent with the secret sharing scheme chosen.[10] This could for instance result in openings of the

[8]These are often called Beaver triples, after Donald Beaver who first proposed them.

[9]There also exists the concept of *identifiable* abort, in which not only the malicious behaviour is detected, but the honest parties can also pinpoint at least one malicious party. This allows for further deterrence against acting dishonestly, as well as enabling the exclusion of dishonest parties from further protocol executions.

[10]Keep in mind that we do not worry about parties sending “incorrect” input values, as even with a magical black box that would perform the computation with perfect security this would be possible. We instead simply want to perform the computation on the *given* inputs.

shares later giving different results for different parties. The common solution is to first generate *random* secret sharings, for which several approaches exist. One can check the consistency of shares by opening several randomly chosen sharings, for instance, without leaking information, as the underlying value is meaningless. Such a random sharing can then be opened towards only the input party, who can compute the difference of the opened value with their input and communicate — in the clear — a linear adjustment that can be used to update the shares.

The second subprotocol we need to worry about deals with the opening of shares. In order to ensure that everyone obtains the same opened value, we want to check that all parties received the same shares. This can be done with extra communication, but that comes at a significant overhead in cost. Instead, the parties will accumulate a checksum over all shares that have been opened so far, and only perform a single round of communication to verify the correctness of that checksum whenever a "meaningful"[11] value is opened.

The reader with a passing knowledge about code theory may observe that there exists a close relation between detecting errors in the shares during reconstruction like this and error detection in linear codes. This is no coincidence and is explored for instance in [SW19]. Such insights can also further provide intuition towards building protocols with guaranteed output delivery from error *correcting* codes, such as Reed-Solomon codes, which can be framed as a reinterpretation of Shamir secret sharing.

Lastly, we can turn our focus to the final aspect of secure-with-abort MPC: multiplications. If the parties have access to actively secure preprocessing material, we can verify that the degree reduction pairs of [DN07] or Beaver triples provide active security with the above approach to securely reconstructing shares. However, to securely generate this preprocessing, or to perform multiplications without it, different methods need to be established. A very generic and costly approach could be to have every party prove their honesty with a zero-knowledge protocol (which we will cover next, in a general sense, in chapter 4).

More specialized methods can however provide better performance, both in terms of communication and computational costs. In one such approach, the parties perform the same computation twice, once with shares of the actual values $[x]$ and once with shares of the value $[\alpha \cdot x]$.[12] They can then perform a final check that all values have the correct correspondence in both executions. This can be seen as a specific instantiation of a general concept where values

[11]By this, we mean that the value does not come from the uniform distribution, such as the final outputs of the computation.

[12]These computations actually occur in parallel, both in order to not double the number of rounds needed and to be able to compute $[\alpha \cdot x \cdot y]$ as the multiplication of $[\alpha \cdot x]$ with $[y]$.

have an attached *message authentication code* (MAC) that can be validated to ensure validity of the computation. In some protocols, these MACs can for instance be held collectively by all parties, or by each party holding a MAC for every other party.

An alternative family of specialized methods makes the parties compute additional, random multiplications — just like they would do in the passively secure preprocessing for multiplication triples — and “sacrifice” these triples to verify the validity of other triples. Such a sacrifice masks the target triple with the random one, and can then securely perform a reconstruction to check the correctness. A target triple could either have the actual computation’s value embedded, or be another random triple in some preprocessing phase providing active security. Further optimizations also exist that improve upon the number of random triples needed per target triple. These techniques often rely on the fact that many multiplications need to be performed or checked at the same time.

CHAPTER 4

Zero-Knowledge Proofs

In this chapter, we explore what it means to prove a statement, and how one could prove a statement such that it does not reveal any further information, while simultaneously being convincing. We look at a few such *zero-knowledge proofs* for specialized statements before diving into constructions for proving more general statements based upon the multiparty computation schemes from linear secret sharing schemes that were introduced in the previous chapter.

4.1 Proof and Arguments

In the previous chapter, we hinted at the possibility of *proving* honest behaviour to enable security against active adversaries. While it may not be the most efficient approach, it is something that can be made possible through cryptographic techniques. In this case, we not only see the want for a way to prove things, but even to do so in a manner that ensures privacy: part of the statement being proven involves secret data that should remain known to only the prover. Any party verifying the proof should learn nothing about this secret data; hence it is given the name *Zero-Knowledge Proof* (ZKP).

ZKPs find many applications as building block in more elaborate cryptographical protocols, such as our example for MPC. Another use case involves identification schemes, where the prover wishes to convince a verifier that they are who they say they are, usually by proving the knowledge of some secret, such as the preimage for a one-way function. From such identification schemes, one can additionally derive signature schemes that combine the identification aspect (binding to the identity represented by a public key) and a message to be signed. In the recent call by NIST for post-quantum digital signature schemes[1] several of the proposals use this transformation, and even rely on the specific type of ZKP (MPC-in-the-Head schemes) that will be the focus of the last part of this chapter.

4.1.1 Definitions and Properties

Before we can properly define what exactly a Zero-Knowledge Proof is, and what exactly makes it have *zero knowledge*, we first define what we understand by a proof, and what properties we require from such a mathematical object.

Ordinarily, one would try to define a proof as some sequence of steps, where each follows logically from the previous ones and a set of axioms. The goal, then, is to convince some *verifier*[2] of the correctness of some statement. Zooming out a bit and forgetting about the structure of a proof for a moment, we can then try to define it by its intended function.

Definition 4.1: Proof

A proof is an interaction between a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$, where $\mathcal{P}$ attempts to convince $\mathcal{V}$ of the veracity of some statement.

[1] https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals
[2] Whether that would be an external entity, or merely oneself.

The details on how exactly such an interaction looks (which we could call a *proof system*) — for instance the prover sending a list of logical steps to the verifier; or $\mathcal{P}$ and $\mathcal{V}$ engaging in a cryptographic protocol — become of secondary importance compared to the concept of a proof and the properties of a proof system.

In order for a proof system to correspond to our initial mathematical intuition and common sense, it needs to satisfy a few criteria. Ideally, if a statement is true, it should be possible to prove it.[3] This property, we call *completeness*.

Definition 4.2: Completeness

A proof system is called complete if every proof for a true statement is accepted by the verifier, except for a negligible probability.

Additionally, we would like false statements to not be provable, or alternatively, for the verifier to reject false proofs. This is known as the *soundness* of the proof system.

Definition 4.3: Soundness

A proof system is called sound if every proof for a false statement is rejected by the verifier, except for a negligible probability.

Either of these properties can either hold unconditionally, or be relaxed to hold except for some negligible probability. When soundness is allowed to only hold when the prover is in some sense restricted, we rather speak of *arguments* in an *argument system*. Common examples of such prover restrictions include computational hardness assumptions or limiting the amount of interaction with some external resource.

4.1.2 Knowledge

In discussions of (cryptographic) proof systems, the word "knowledge" appears frequently. However, it tends to have two separate meanings in this context. The first meaning usually appears as part of the phrase "proof of knowledge" or "knowledge-soundness". The second meaning is normally negated as "zero-knowledge", which opens up some bigger questions as to how we should think of

[3]Note that this is a superficial and intuitive request that will not always hold up to further scrutiny. For instance, Gödel's incompleteness theorems may worry the observant reader here. In practice, for our cryptographic applications, the statements being proven will be restricted enough that this property can still be achieved.

proofs. While the two meanings are not entirely unrelated and both still reference the colloquial use of the word, it is interesting to note that zero-knowledge is not exactly the opposite of knowledge in the first sense.

The term *proof of knowledge (PoK)* is used for proof systems that exhibit — in addition to the properties from before — *knowledge-soundness*. In intuitive terms, the prover is proving that they *know* some value that satisfies some condition. For practical situations, this condition will most often involve some one-way function. Intuition is of course not enough for mathematical rigour, so we should ask ourselves what it means, formally, for the prover to *know* something. For this, we imagine a hypothetical party or algorithm: a *(knowledge) extractor*. The knowledge extractor is allowed to interact with the prover, and *rewind* them to some previous state or point in time. With this interaction, the goal of the extractor is then to output a value satisfying the condition placed of the "known" value. If every prover that has a non-negligible chance of making the verifier accept admits a successful extractor (with non-negligible success probability), we call the proof system *knowledge-sound*.

Definition 4.4: Knowledge-Soundness

A proof system is called knowledge-sound if for every prover, there exists a knowledge extractor that, upon interaction with the prover, outputs a valid *witness* to the proof's statement with non-negligible probability. A proof for a knowledge-sound proof system is also called a proof of knowledge.

Note that the existence of an extractor does not necessarily mean that the prover has the value under consideration explicitly stored in memory. It merely indicates that the prover has enough information to efficiently compute the value should the need arise.

Besides having the prover show that they know some "useful" value, we also often want to make sure that the verifier does not learn that value. Indeed, if the prover is e.g. somehow proving their identity, it would likely be a bad idea to enable the verifier to replicate that proof somehow and convince some second verifier that they were the prover instead. To strengthen this requirement a bit, we want the verifier to learn *nothing at all*, other than the veracity of the prover's claim. If we can achieve this, we call it the *Zero-Knowledge (ZK)* property. Before properly formalizing this notion, we can first try to draw a first conclusion based solely on our intuitive understanding: ZK proofs *must* involve interaction between the prover and the verifier. Consider to the contrary a proof system where the prover simply outputs a bit string π as proof and sends it to the verifier. If the verifier accepts the proof, they are also able to "replay" it towards a second verifier, who would naturally also accept it using the same verification procedure. A PoK with the ZK property is commonly referred to as a *ZKPoK*.

To get a proper definition for the ZK property, we can start from this idea and expand on it a bit. If we lift the concept of a single bit string as proof to a setting with interaction, we can define the *transcript* of a protocol as the ordered sequence of messages sent between the parties. In a Zero-Knowledge proof, we require that this transcript cannot, in turn, be used as a (non-interactive) proof of the statement. That is, it should not be sufficient to check that a transcript is consistent with a proof system to verify that the proof is correct. Hence, it must be possible to forge a transcript that cannot be told apart from a genuine interaction. This, finally, leads us to a more formal definition of ZK.

Definition 4.5: Zero-Knowledge

A proof system is Zero-Knowledge if for every verifier, there exists a *simulator* that, without interaction with the prover and in expected polynomial time, outputs a forged transcript that is indistinguishable from a transcript of a genuine interaction between prover and verifier. This indistinguishability can sometimes be relaxed to notions of statistical or computational security guarantees, rather than merely information theoretical.

At this point, it may be a natural reaction to wonder how soundness and zero-knowledge are not mutually exclusive. After all, shouldn't the ability to forge a transcript mean that it is possible to "fool" the verifier? The answer here lies once again in the interaction inherent in the protocol. Because the verifier is an active participant in the protocol, and therefore knows that the transcript was created in the correct order, they can be confident that the proof was correct. The simulator is in contrast not bound to the same honest ordering of the transcript, and can instead generate the transcript in any way they'd like, as long as the final result is indistinguishable from a real one.

To now try and crystallize our intuition for all these properties some more, we examine two low-tech examples of zero-knowledge proofs.

Example 4.1: Wine tasting

Imagine you are trying to learn about wine tasting. To you, the two glasses in front of you taste exactly alike, but your instructor exclaims "clearly these two are different, can you not recognize the difference in tannins?"

You are unconvinced by this explanation, and so you want to put it to the test. While the instructor (taking the role of prover) turns around, you (the verifier) will shuffle the two glasses around, leaving them either in their original places or swapped around in the end. Then it is the task of the instructor to determine which of these two actions you took. To argue completeness, we can see that if the instructor can indeed taste the difference, they should be able to determine which glass is which, and hence determine if the order is the same as before or not. For soundness, we observe that if the instructor is unable to tell the wines apart, they only have a one in two chance of guessing your action correctly. If this experiment is repeated enough times, after k attempts, a dishonest prover will only have a $\frac{1}{2^k}$ probability of correctly guessing every action. As long as we assume you choose either action with equal probability, of course. And, as this experiment teaches you nothing at all about wine tasting and the two wines still taste exactly alike to you, this could arguably be called zero-knowledge. A simulator could for instance first choose which action to take, and then simply choose the corresponding correct answer for the instructor. The original statement only makes a claim about an ability of the prover, rather than about knowledge, so there is no knowledge to be extracted from the prover and this is not a proof of knowledge.

Example 4.2: Sudoku

Now imagine you just solved a hard sudoku, and you want to challenge your friend to also solve it. They try for a while, without success, and start complaining that it cannot be solved. Of course, you want to prove them wrong, but you do not want to simply reveal the solution, as that would take away the challenge.
In this case, you will take the role of prover, while your friend will take the role of verifier. As a first step, you take your solution to the sudoku, and permute the digits. After all, the logic of sudoku does not rely on the value of the digits, only on a distinguishable identity, and a permutation of the digits does not change this underlying identity. For the second step, you cover the entire (permuted) sudoku with scratch-off foil, and show it to your friend. They then can make a choice between a few options. Either, you reveal the cells that were already filled for the blank sudoku, you reveal a specific column, you reveal a specific row, or you reveal a specific three by three box. In the first case, your friend can check that you revealed a permuted version of the original sudoku, and

in all other cases, they can check that you revealed nine distinct digits. Completeness follows easy, as the steps guarantee that the cells you reveal will have the correct properties. Soundness is a bit more difficult, but since you do not know the challenge your friend will give you, you cannot satisfy all requests simultaneously without knowing a solution. Hence, you only have a bounded probability of the revealed digits being correct, which can again be improved by repeating the experiment multiple times. The uniformly random permutation and the limited amount of revealed digits ensure that, if the challenge is known beforehand, it is easily answered without knowing a solution: either choose a random digit permutation of the original sudoku, or choose a random order of all nine digits to place in a row, column or box. A simulator can again apply the same approach of first choosing the challenge and then choose the taped-over sudoku.

In this example, we are dealing with a proof of knowledge, as your claim is to know a solution to the given sudoku. To show this is indeed the case, we can describe a knowledge extractor that can rewind the prover and outputs a solution to the original sudoku. The essence of the rewinding capability is that it allows the extractor to challenge the prover multiple times for the same digit permutation. This means that in only nine queries, it can reveal the entire (permuted) sudoku, say by querying all nine rows. In combination with the original blank sudoku, the digit permutation can be recovered, which in turn results in a recovered solution.

There are some further subtleties that we do not fully address in this high level overview. For instance, the verifier could decide which challenge to choose based on previous communication with the prover. This complicates the simulation argument, as used in the previous two examples, a bit, since a simulator must exist for *any* verifier. Each verifier can however have its own dedicated simulator, which allows for some leeway, and the problem can often still be solved by having the simulator interact with the verifier. The simulator then often first guesses the challenge, generates a corresponding message to the verifier, and checks if it guessed correctly. If so, it can add the output to the forged transcript, otherwise it simply starts over. Another issue can arise when we do not expect the prover to be perfect. That is, they only prove the statement correctly with non-negligible probability. This complicates the job of the knowledge extractor, and again needs some further technical depth to ensure that it succeeds in expected polynomial time.

4.1.3 Eliminating Interaction

In the previous section, we claimed that interaction was a strict necessity for ZK proofs. However, in practical settings such as the construction of signature schemes from zero-knowledge protocols, interaction may not always be possible or desirable. Therefore, we may want to relax our definition a bit towards *non-interactive zero-knowledge proofs (NIZKs)*. While technical definitions exist, for our intuitive purposes, it is enough to think of them as non-interactive proofs, where the verifier learns nothing, other than the proof string.

This change in setting does not mean that all the work on interactive proof systems is lost or useless. In many cases, it is possible to take an interactive proof system, and convert it to remove the requirement of interaction. The most common such case is when the verifier is a *public-coin verifier*.

Definition 4.6: Public-Coin Verifier

A public-coin verifier is a verifier that only sends messages that have been sampled from a known distribution. That is, it "flips some coins" and sends the result to the prover as some form of challenge.

Public-coin verifiers are fairly common, and in fact, both examples from the previous section have one.

Given a zero-knowledge proof system with a public-coin verifier, the *Fiat-Shamir transformation* (introduced by Fiat and Shamir in [FS87]) constructs a NIZK by replacing the verifier with a *random oracle (RO)*[4] that takes as input the entire transcript up to that point, and outputs the verifier's random coins. One can think of a random oracle as a shared function, that takes arbitrary bit strings as input and outputs uniformly random, fixed-length bit strings as output, ensuring that identical inputs map to identical outputs. For practical applications, this random oracle will usually be instantiated as a fixed chosen hash function, which is not entirely theoretically correct, but "good enough" according to the *ROM heuristic*. There are several security considerations to make when dealing with the Fiat-Shamir transformation, such as the number of queries we allow the prover to make to the RO, and how this interacts with the composition of multiple repetitions with constant soundness error, but a deeper discussion is outside the scope for this chapter. Some further details on specifically the problems with sequential repetition are covered later in chapter 6.

[4]Thus moving the proof system into the so-called *random oracle model or ROM*.

4.2 Knowledge of Homomorphism Preimages

In this section, we give an example of a ZKPoK that falls within the category of *sigma protocols*, following [Mau15]. The prover wants to prove knowledge of the preimage of a homomorphism (of groups, one can think of this through the definitions of chapter 2 but having only one binary operation), for which computing the preimage is assumed to be computationally hard. To do so, we first look at Schnorr's identification protocol [Sch90] and how its non-interactive variant gives rise to Schnorr signatures, before briefly describing how Maurer's scheme generalizes and subsumes it. This section illustrates the construction of ZK protocols for specialized or restricted statements, such that in the next section, we can shift our focus to ZK proof systems for general computation that build upon MPC protocols.

4.2.1 Proving the Discrete Logarithm

We first briefly give a definition of a sigma protocol, so that the reader can recognize them where they show up. We will not always explicitly point out which protocols are sigma protocols or not, but the specific structure occurs commonly enough in the literature that being aware of it is generally useful.

Definition 4.7: Sigma Protocol

A sigma protocol is an interactive proof system in which the prover and the verifier exchange three messages. The prover first sends a commitment to the verifier; the verifier responds with a random challenge; and finally, the prover sends one last message that depends on the challenge and opens the commitment.

The name sigma protocol comes from the resemblance of visual diagrams of this interaction to the Greek letter Σ.

Now we describe Schnorr's sigma protocol to prove knowledge of a discrete logarithm. Assume prime numbers p and q, along with a number $g \neq 1$, such that $g^q \equiv 1 \pmod p$.[5] Now let $y \equiv g^x \pmod p$ be the public statement, with $0 \leq x < q$ unknown except to the prover. The goal of the prover is then to convince a verifier that they indeed know such a value x, without revealing any further information. Since the process of finding x, given y, g, p and q (known as the *discrete logarithm* problem) is assumed to be computationally hard for

[5] So g is a generator for a cyclic group of prime order q.

large enough q and p, the verifier cannot independently find x in polynomial time, nor any other discrete logarithm the prover may send.

In the protocol, the prover first sends a random commitment $t \equiv g^\alpha \pmod{p}$, for some uniformly random $0 \leq \alpha < q$. The verifier responds with a uniform challenge $c \in \{0, 1\}$, and is hence a public-coin verifier. For the final message, the prover answers with $z \equiv \alpha + c \cdot x \pmod{q}$. The verifier can now check whether $g^z \equiv t \cdot y^c \pmod{p}$, and accept the proof if so. This protocol can be repeated multiple times to improve on the soundness. It is left as an exercise to the reader to verify that the properties from section 4.1.1 and section 4.1.2 are correctly satisfied.

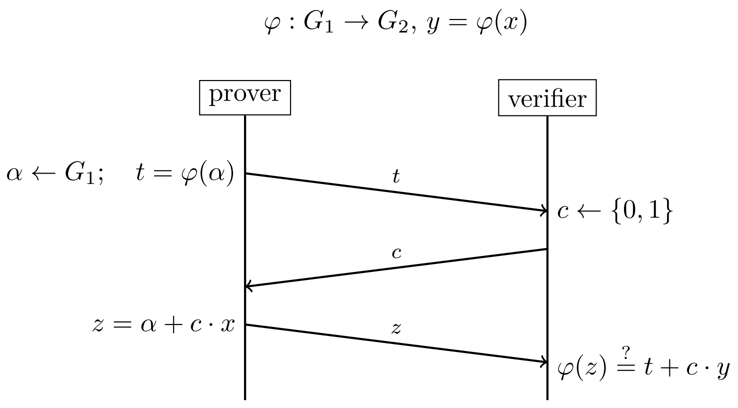
As described in section 4.1.3, we can also turn this sigma protocol into a NIZK, by choosing $c = \mathcal{O}_R(t)$ as the output of a random oracle. Additionally, we can bind this challenge to further "context", such as a session identifier or a message, to avoid the proof being replayed in a different context. To achieve this, one can simply feed more input into the random oracle, and to bind the proof to a message m, it suffices to let $c = \mathcal{O}_R((t, m))$, with some appropriate encoding of the tuple (t, m) into a bit string. It can be proven that the resulting scheme is in fact a secure signature scheme for public key y, private key x and message m. Several commonly used signature schemes such as Schnorr signatures and (EC)DSA are either based on this exact construction, or closely related.

4.2.2 Exponentiation is a Homomorphism

By taking an extra step in abstraction, it suddenly becomes easy to build upon Schnorr's protocol to build sigma protocols for a larger class of statements. The key observation is that one can see the exponentiation $g^x \pmod{p}$ as a homomorphism φ of $(\mathbb{Z}_q, +)$ onto $(\mathbb{Z}_p, \cdot)$, for which it is (assumed to be) computationally hard to compute a preimage.[6] If one replaces every exponentiation in Schnorr's protocol with an evaluation of the homomorphism, it can be shown that this indeed makes for a ZKPoK that proves knowledge of a preimage of y.

To briefly summarize, we can represent the resulting protocol visually.

[6] Here, a preimage of y is exactly any value x such that $\phi(x) = y$.


$\varphi : G_1 \to G_2,\ y = \varphi(x)$
prover
verifier
$\alpha \leftarrow G_1; \quad t = \varphi(\alpha)$
t
$c \leftarrow \{0, 1\}$
c
$z = \alpha + c \cdot x$
z
$\varphi(z) \overset{?}{=} t + c \cdot y$

4.3 Zero-Knowledge Proofs from MPC

In the previous section, we discussed a construction for a proof system dealing with a specific class of statements, discrete logarithms, and expanded it to a wider range of statements. However, by itself, even that wider class of homomorphism preimage statements remains rather limited in what it can express. We would like to be able to construct proof systems in which even more powerful statements can be proven. Some examples would be to prove the correct evaluation of a program (referred to as *verifiable computation*), or to prove knowledge of an encryption key — for example to transform it into a post-quantum signature scheme. In this section, we first briefly revisit the discussion from chapter 3 on how we want to represent such a general computation. Then we discuss some constructions of this functionality by building upon MPC protocols, the result of which are called *MPC-in-the-Head (MPCitH or MitH)* schemes.

4.3.1 Proofs for General Computation

To shift our view from more specialized proofs, such as those in the preceding section, towards more general proof systems that can show the correctness of arbitrary computations, we first determine a useful structure for the statements we now wish to prove. If we look at the statement $y = \varphi(x)$ from before, and abstract it a bit further, we could choose to represent any statement as $y = C(x)$, or with a more common choice of variables $x = C(w)$. Here x is often referred to as the *statement*, while w is the *witness*, and C can be any circuit to represent an arbitrary piece of computation, just like we described in section 3.2.1. Like before, once we start considering the concrete efficiency of specific instantiations and protocols, these circuits could be further enhanced with special-purpose *gadgets*, such as memory elements that can be read or written (known as the *oblivious RAM (ORAM)* model). Of course, this shift in viewpoint loses the homomorphic properties that enabled the earlier sigma protocol, so in the later subsections, we shall explore some alternative constructions.

While this model may seem to be somewhat restrictive and to cover only verifiable computation, it is not hard to apply it to a situation where the proof is merely concerned with asserting that some properties hold for the witness. If we let the output of the circuit $x \in \{0, 1\}$ mean respectively whether the required properties hold or not, the computation $x = C(w)$ with publicly known statement $x = 0$ correctly models this situation too. Additionally, if we want the circuit to take both public inputs y and a private witness w, that is $x = C(y, w)$, this can be represented in two different ways. As a first option, we can embed the public input y into the final circuit description, by defining the new circuit $C'(w) = C(y, w)$. For the second choice, we can absorb y into both the statement x and the witness w: $C'((w, y)) = (C(w), y) = (x, y)$, where the presence of y in the public statement establishes the required consistence.

Example 4.3

Consider as a simple example of these manipulations the goal of proving that some public element y in a ring R is invertible. To show this, the prover can choose as witness $w = y^{-1}$, which only exists if the statement is indeed true. To express this in a circuit, one could choose to prove $y = C(w) = w^{-1}$, but this may be a costly computation to perform. Rather, if we let the circuit simply verify that $w \cdot y = 1$, we can express it as $1 \overset{?}{=} C((w, y)) = w \cdot y$, requiring only a single multiplication to be proven.

From a public-coin verifier proof system for such statements, we can, just like before, construct non-interactive proof systems through the Fiat-Shamir transform. Such NIZKs for one-way functions can in turn again be made into signature schemes by additionally binding the proof to a message. Common choices for the one-way functions include asymmetric primitives such as the discrete logarithm from the previous section, and symmetric primitives such as block ciphers like AES or hash functions. If the proof system can be constructed from symmetric primitives — as is the case for the MPCitH schemes we will discuss next — and the choice of one-way function is a symmetric primitive, this constructs a signature scheme that solely depends on symmetric primitives, and as such would be very plausibly quantum-resistant.[7]

4.3.2 Building on MPC

When considering interactive (ZK) proofs, it is natural to interpret it as a computation performed by two parties, and hence that it can be seen as an instance of two-party MPC (or 2PC). Indeed, given the above setting, one can take an actively secure 2PC protocol that computes $C(w)$ for some input w held by the prover and call it an interactive proof. The privacy guarantees of the protocol ensure that the zero-knowledge property holds, while soundness is provided by the active security and completeness by the MPC protocol providing evaluation for arbitrary circuits. Indeed, such constructions have been proposed before using MPC protocols specialized to the 2PC setting, such as in [JKO13].

However, this general construction has several downsides. The first one is the necessity for *active* security of the 2PC protocol, which comes at a significant computational and communicational cost. The second downside arises from the active role the verifier takes in the protocol evaluation. Since the 2PC protocol makes no guarantees about the computation of the verifier, it cannot generically enable a public-coin verifier, which in turn prevents straightforward translation of the protocol into a NIZK.

Instead, in [IKOS07], Ishai et al. introduce a black-box transformation for any — only passively secure — MPC protocol into a ZK proof system. The general idea is that the prover can, without any interaction, simulate the execution of an N-party MPC protocol and commit to the *views* of the parties — the view of a party is a localized transcript of the MPC protocol that contains all the messages sent and received by a single party. After receiving this commitment, the verifier then chooses a (public-coin) challenge consisting of a privacy-preserving subset of the MPC parties that the prover must open. If the prover is dishonest, there necessarily exists a pair of parties with conflicting views, as otherwise the MPC protocol would correspond to an honest execution and the statement would be true. Depending on the exact access structure of the MPC protocol and the number of parties, this results in a fixed probability of a dishonest prover being caught that can, as before, be amplified through repetition. This approach succeeds in mitigating both downsides of the 2PC approach, allowing for the MPC protocol to be only secure against semi-honest adversaries and the resulting ZK protocol to have a public-coin verifier.

[7]The Fiat-Shamir transform itself is known to occasionally have some issues in the adaptation of the ROM to the quantum setting. There is however an alternative due to Unruh [Unr15] that avoids these issues at a somewhat higher computational cost.

Proof by Computation The landscape of MPCitH proof systems can be roughly divided into two families. The first of these families, with [KKW18] as notable example, take the witness as input and compute the statement circuit "forward". This means that the simulated MPC protocol computes all gates in the circuit in the usual order and reconstructs all outputs. With MPC protocols built upon LSSS techniques, the linear gates are free of communication, and hence virtually free in terms of cost towards the proof size as well. Non-linear gates, such as multiplications, however, take more effort and communication to compute, leading to a focus on efficiently preprocessing random multiplication triples that can be used to minimize the cost of the actual multiplication gates.

Proof by Verification The second family, with notable representatives like Banquet [BDK+21] and Limbo [DOT21], puts the burden of computing the circuit's wires on the prover — so outside the MPC protocol, — and then has the MPC protocol perform a verification of the correctness of this computation. The input to the MPC protocol now consists of the original witness, along with all outputs of multiplication gates,[8] which is commonly referred to as the *extended witness*. The chief advantage of this is that all multiplications have become, in a sense, "decoupled", since the extended witness ensures that no multiplication depends on the output of a previous multiplication any more. All multiplications are now at the same depth in the (transformed) circuit.

With all multiplication and their results available simultaneously, the verification MPC protocol can now perform a "batch" check of these multiplications and ensure that all are correct *at the same time*. In most cases, this batch check will, for performance reasons, be a probabilistic check. Due to the nature of such probabilistic checks, the prover should not be allowed to know the used randomness beforehand, so this should be provided by the verifier after receiving a commitment to (the secret sharing of) the extended witness. Additionally, the probabilistic nature of the check introduces an extra term to the soundness error or cheating probability, as it is possible for the check to output a false negative result[9] with probability depending on the exact check and the field or ring over which the check is performed.

MPCitH in Context

Finally, we provide some context on how MPCitH proof systems compare to other contemporary proof systems (for arbitrary statements or circuits) based on

[8]This is sometimes also described as the prover "injecting" the results of multiplication gates into the protocol, as if it were an oracle query.

[9]Negative here meaning that no *wrong* multiplications were found.

other techniques. Observe that the proof size for the protocols as outlined above scales at least linearly with the number of multiplication gates in the statement circuit. It turns out this is an asymptotic lower bound that can actually be achieved, and the computational time taken by the prover and the verifier scale linearly in the circuit size too. A line of work based on MPCitH, starting with Ligero [AHIV17], achieves a sublinear proof size and verification time ($\mathcal{O}\left(\sqrt{|C|}\right)$), with a somewhat larger prover time. The asymptotic improvement provides concrete advantages only once the circuit size becomes large enough, so the linear MPCitH schemes generally still yield smaller proofs for small to medium-sized circuits.

For proofs that need to be verified often, a verification time that scales linearly with the circuit size may be restrictive. In the literature, schemes such as *SNARKs* (Succinct Non-interactive Arguments of Knowledge) and *STARKs* (Scalable, Transparent Arguments of Knowledge) have been proposed that achieve proofs of constant size or scaling polylogarithmically in the circuit size. These constructions can rely on a variety of hardness assumptions, with both asymmetric and symmetric underlying primitives, resulting in schemes that are not always quantum-resistant. The exact technical details of such schemes are out of scope for this work, but we mention their existence as they may be more commonly known among a general audience (with an interest in cryptography). The improvement in proof size and verification time that SNARKs and STARKS achieve comes, however, at a noticeable computational cost to the prover. Where a concretely efficient prover is desirable, and the circuits being proven do not become excessively large, a proof system based on MPC-in-the-Head techniques will often be the right choice.

CHAPTER 5

MPC for $\mathcal{Q}_2$ Access Structures over Rings and Fields

Robin Jadoul[1], Nigel P. Smart[1], and Barry Van Leeuwen[1]

[1]imec-COSIC, KU Leuven, Leuven, Belgium.

[JSv22] Robin Jadoul, Nigel P. Smart, and Barry van Leeuwen. MPC for Q_2 access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 131–151. Springer, Cham, September / October 2022.

Abstract: We examine Multi-Party Computation protocols in the active-security-with-abort setting for $\mathcal{Q}_2$ access structures over small and large finite fields $\mathbb{F}_p$ and over rings $\mathbb{Z}_{p^k}$. We give general protocols which work for any $\mathcal{Q}_2$ access structure which is realized by a multiplicative Extended Span Program. We generalize a number of techniques and protocols from various papers and compare the different methodologies. In particular, we examine the expected communication cost per multiplication gate when the protocols are instantiated with different access structures.

My contributions: Main author
I contributed the protocols in sections 4 through 7 and the performance analysis, including the cases in appendix B.

5.1 Introduction

Secure multiparty computation (MPC) considers the situation where some set of parties $\mathcal{P}$ come together to compute a function, each with their own inputs. The security requirement is that no party is able to learn more than what the output of this computation and their own input would allow them to. From another perspective, this can be seen as a protocol that emulates a perfectly honest, trusted third party that obtains each party's input, performs the computation, and outputs the result.

We can distinguish different security notions based on the power an adversary can have. One axis along which to distinguish is whether the adversary is active or passive. A passive adversary, also sometimes called *honest but curious*, follows the protocol correctly, but tries to obtain more information from the parts of the transcript of the execution it can see. An active adversary on the other hand, is able to arbitrarily deviate from the protocol. In this situation we either require that the honest parties still obtain the correct output from the function, in which case we say that the protocol is robust, or we require that the honest parties abort the protocol with overwhelming probability, in which case we say the protocol is *actively-secure-with-abort*. In this chapter we concentrate on protocols which are actively-secure-with-abort, as they are relatively fast and practical in a large number of situations. Those readers who are interested in robust active security should consult [ACD+20, CRX19].

Another axis to consider is how many or which subsets of parties the adversary can corrupt. If we have n parties then a full threshold adversary is one who is able to corrupt at most $n-1$ parties. In such a situation we can achieve active-security-with-abort, however this comes at the expense of a costly preprocessing phase; see [DPSZ12, CDE+18] for the case of MPC over finite fields, or over finite rings. Simpler protocols can be obtained if one restricts the adversary to corrupt less parties. The classic restriction is that of threshold adversaries who are allowed to corrupt up to $t < n$ parties. When $t < n/2$ very efficient MPC protocols can be realized, using a variety of methodologies to obtain active-security-with-abort. The natural generalization of the threshold $t < n/2$ case is that of so-called $\mathcal{Q}_2$ adversary structure. A $\mathcal{Q}_2$ adversary structure is one where the union of no two unqualified sets contains the whole set of players $\mathcal{P}$. For threshold structures the set of unqualified sets are all subsets of $\mathcal{P}$ of size t, thus clearly no two sets can contain all of $\mathcal{P}$ when $t < n/2$. In this chapter we will focus on $\mathcal{Q}_2$ access structures, again as they are relatively fast and practical in a large number of situations.

A third axis to consider is the underlying field or ring over which the MPC protocol is implemented. Traditionally the focus has been on MPC protocols over fields $\mathbb{F}_p$, either large finite fields or small ones (in particular $\mathbb{F}_2$). However, recently interest has shifted to also considering finite rings such as $\mathbb{Z}_{p^k}$, and in particular $\mathbb{Z}_{2^k}$. In this setting sometimes, to obtain active security, underlying protocols require the players to work in the extended ring $\mathbb{Z}_{2^{k+s}}$, for some security parameter s, and sometimes this is avoided. In this work we will consider all such possibilities.

The final axis to consider is the precise protocol to use. Almost all practical protocols which are actively-secure-with-abort for $\mathcal{Q}_2$ access structures divide the protocol into two, and sometimes three stages. The first stage, called the offline or pre-processing stage, is function independent and generates various forms of correlated randomness amongst the parties. A second stage, called the online stage, uses the pre-processing to compute the output of the function in a secure manner. Sometimes a third stage, called the post-processing stage, is required to ensure active-security.

The investigation of the combination of the second, third and fourth axes forms the basis of this work. We generalize, where needed, prior works in order to investigate as many prior protocol variants as possible, when instantiated over finite rings or fields. We also generalize results from specific $\mathcal{Q}_2$ access structures to general $\mathcal{Q}_2$ access structures so as to obtain a complete smorgasbord of options. We then analyse the different options, as it is unclear in which situation which protocol is to be preferred (even in the case of finite fields).

Prior Related Work:

The majority of the literature has focused on the case where the underlying arithmetic is a finite field. These are often based, for general finite fields and $\mathcal{Q}_2$ access structures, on the classic multiplication protocol of Maurer [Mau06], which works for an arbitrary multiplicative secret sharing scheme. In the case of small finite fields and small numbers of parties, for example $\mathbb{F}_2$ and three players it is common to utilize a multiplication protocol based on replicated secret sharing, which originally appeared in the Sharemind software [BLW08]. The generalization of this specific multiplication protocol to arbitrary fields and $\mathcal{Q}_2$-access structures implemented by replicated secret sharing [KRSW18], the generalization to an arbitrary $\mathcal{Q}_2$ MSP was done in [SW19]. Both of these multiplication protocols we shall refer to as KRSW. There is a third passively secure multiplication protocol due to Damgård and Nielsen [DN07], which we shall refer to as DN multiplication. The DN multiplication protocol is often combined with a "king-paradigm" for opening a sharing, this reduces the total amount of data sent at the expense of doubling the number of rounds. As round complexity has often a bigger impact on execution time than data complexity we assume no king paradigm is used in our protocols.[1] Thus, before one even considers the various protocols, one has (at least) three base passively secure multiplication protocols to consider. In this work we will concentrate on these three, Maurer or KRSW or DN. The one which is more efficient depends on the precise context as we will show. From these, when using multiplication triples, one can derive a third passively secure multiplication triple which we shall call Beaver multiplication.

In more recent works, research has started to focus on MPC over finite rings, such as $\mathbb{Z}_{p^k}$, and $\mathbb{Z}_{2^k}$ in particular. For many cases, this choice is more natural, as it more closely aligns with the bitwise representation of numbers found in standard computing, and it can enable efficient high level operations such as bit-decomposition (which are very useful in practice). For example, working over $\mathbb{Z}_{2^{64}}$ would closely mimic the behaviour we have on most currently used CPUs. The main problem with working with such rings is the presence of zero-divisors.

A method to avoid the problem of zero-divisors in secret sharing schemes over rings with zero-divisors was presented in the SPD$\mathbb{Z}_{2^k}$ protocol of [CDE+18]. Originally, this was presented in the case of a full threshold adversary structure, but the basic trick used applies to any access structure. To avoid the problem of zero divisors when working modulo 2^k, the authors extend (for some protocols) the secret sharing to a large modulus 2^{k+s}, for some statistical security parameter s. This idea was extended to the case of simple $\mathcal{Q}_2$ access structures, using a replicated secret sharing schemes, in [EKO+20]. With some of the resulting protocols for $n = 3$ and $n = 4$ parties implemented in the MP-SPDZ framework [Kel20].

Across the many papers on $\mathcal{Q}_2$ MPC we identify three forms of actively secure pre-processing used in the literature, which we generalize[2] to an arbitrary setting of p^k. The first, which we denote by $\mathsf{Offline}_1$, uses a passively secure multiplication protocol to obtain $2 \cdot N$ triples. These are then made actively secure using the classic technique of sacrificing (which effectively uses internally a Beaver multiplication), resulting in an output of N triples. This variant has been used in a number of papers, e.g. [SW19]. A second variant, which we denote by $\mathsf{Offline}_2$, generates N passively secure triples, and then checks these are correct using a different checking procedure, based on the underlying passively secure multiplication protocol of choice. This variant was used in

[1]Note the king-paradigm can be used not only in DN multiplication but in any protocol which involves opening shares to all players, as long as suitable additional checks are performed to ensure active security.

[2]There are a few others which we do not consider, as they do not easily fit into our protocol descriptions below. For example the protocol of [ADEN19] looks at threshold structures and uses the multiplication protocol of [DN07] using a king paradigm.

[EKO+20].

A third offline variant, which we shall denote by $\mathsf{Offline}_3$, uses a passively secure multiplication protocol to obtain triples in the offline phase. These are then made actively secure using a cut-and-choose method, as opposed to sacrificing. The reason for this is that they are interested in MPC over $\mathbb{F}_2$ and classical sacrificing has a soundness error of one over the field size, and using cut-and-choose allows one to perform an actively secure offline phase without needing to pass to a ring of the form $\mathbb{Z}_{2^k}$. This methodology was presented in [ABF+17], and we shall also call this ABF pre-processing. This method seems very well suited to situations when p^k is small as it does not require extending the base ring to $\mathbb{Z}_{p^{k+s}}$.

From these one can derive a number of complete protocol variants. The first variant, which we shall denote $\mathsf{Protocol}_1$, exploits the error-detecting properties of a $\mathcal{Q}_2$ access structure to obtain a protocol which uses an actively secure offline phase, and then uses an online phase based on the classical Beaver multiplication method. Active-security-with-abort is achieved using the error detecting properties of the underlying secret sharing scheme. This has been considered in a number of papers in the case of threshold structures with $(n, t) = (3, 1)$, with the generalization to arbitrary $\mathcal{Q}_2$ structures in the case of large finite fields being done in [SW19].

In [EKO+20] a three party protocol is presented which makes use of a different methodology, which we generalize to arbitrary $\mathcal{Q}_2$ access structures. Here the online phase is executed optimistically using a passively secure multiplication protocol. The multiplications are then checked to be correct at the end of the protocol using a post-processing phase. Depending on the method used to perform this checking, we can either generate auxiliary, passively secure triples in an offline phase, that can be used in a form of sacrificing in the post-processing phase (which we dub $\mathsf{Protocol}_2$), or we can completely remove the need for a preprocessing step (which we dub $\mathsf{Protocol}_3$).

The paper [ABF+17] also uses an optimistic passively secure online phase with a post-processing step, but combines this with an actively secure offline phase. By doing this the post-processing check is always checking possibly incorrect multiplications (from the online phase) against known-to-be-correct multiplications (from the offline phase). This means the post-processing check can be done using a method which is close to that of classical sacrificing, without the need to worry about the small field size. We call this variant $\mathsf{Protocol}_4$.

The final protocol variant we consider, which we dub $\mathsf{Protocol}_5$, comes from [CGH+18]. In this paper the authors dispense with the offline phase, and instead generate a shared MAC-key $[\alpha]$, a bit like in SPDZ, and evaluate the circuit on both $[x]$ and $[\alpha \cdot x]$ using a passively secure multiplication protocol. Thus, in some sense, the circuit is evaluated twice in the online phase. The correctness of the evaluation is then established using the MAC-Check protocol from the SPDZ protocol. Thus, there is a post-processing step, but it is relatively light-weight, however the online phase is more expensive than other techniques.

We summarize these in five protocol variants in Table 5.1 as a means for the reader to maintain a quick overview as they read the chapter.

	Offline Phase		Online	Post-Processing	
Protocol	Passive	Active	Phase	Heavy	Light
$\mathsf{Protocol}_1$	-	✓	Beaver	-	-
$\mathsf{Protocol}_2$	✓	-	Passive	✓	-
$\mathsf{Protocol}_3$	-	-	Passive	✓	-
$\mathsf{Protocol}_4$	-	✓	Passive	✓	-
$\mathsf{Protocol}_5$	-	-	$2 \times$ Passive	-	✓

Table 5.1: Summary of our five protocol variants. A "heavy" post-processing phase denotes a phase akin to sacrificing, whereas a "light" post-processing denotes a phase akin to SPDZ-like MAC checking. A Passive online phase refers to an online phase using either Maurer or KRSW multiplication.

Our Contribution:

In this work we unify all these protocols; in prior work they may have been presented for finite fields, or for rings of the form $\mathbb{Z}_{2^k}$, or for specific access structures. We consider, in all cases, the general case of MPC over rings of the form $\mathbb{Z}_{p^k}$; i.e. where we consider both the case of $k = 1$, large k, small p, and large p in one go. Our methodology applies to all multiplicative $\mathcal{Q}_2$ access structures over such rings. To do so we utilize the language of Extended Span Programs, ESPs, introduced in [Feh98]. This allows us to consider not only replicated access structures, but also access structures coming from Galois Ring constructions. By considering such Galois Ring constructions as an ESP, we can maintain working over $\mathbb{Z}_{p^k}$ without the need to worry about complications arising from the Galois Ring.

We first show how one can create the necessary ESPs for a specific access structure, by constructing an associated MSP over the field $\mathbb{F}_p$ and then lifting it to $\mathbb{Z}_{p^k}$ in a trivial manner. This preserves the access structure, but it does not always preserve multiplicity (see [ACD$^+$20] for a relatively contrived counter example). For all "natural" MSPs one might encounter in practice (arising from Shamir or Replicated secret sharing) the lifting does preserve multiplicity. In any case if the resulting ESP over $\mathbb{Z}_{p^k}$ is not multiplicative, it can be extended to a multiplicative ESP in the standard manner[3].

We show that the error-detection properties of [SW19] apply in this more generalized context of finite rings. This allows us to reduce the communication cost in our protocols for ESPs. Note the error-detection properties exploited in [SW19] are the precise generalization to arbitrary $\mathcal{Q}_2$ MSPs of the classical check for correctness performed in threshold systems for $(n, t) = (3, 1)$ based on replicated sharing.

We also show that the trick of modulus extension from $\mathbb{Z}_{p^k}$ to $\mathbb{Z}_{p^{k+s}}$ also works in general, and we combine it with other tricks. For example, we use Schwartz-Zippel over Galois rings to allow greater batching, and modulus extension even in the case of checking over finite fields. Indeed, we show that one can also utilize modulus extension to avoid the problems with sacrificing when $k = 1$ and p is small. However, this comes at the expense of requiring to work modulo p^{k+s} and not working modulo p^k, which may be a problem in some instances (for example in the interesting case of $p^k = 2$). Thus, our multiplication checking procedures in Section 5.5 generalize a number of earlier results, and unify various approaches. Note, that depending on the underlying protocol choice such modulus extensions may not be needed.

We finally examine the smorgasbord of options for the offline, online and post-processing which we outlined above in this general context and examine the various benefits and tradeoffs which result. Our cost metrics in this matter are the total number of rounds of communication, as well as the total amount of data sent per multiplication[4]. We consider the case where the user is interested in minimizing the total cost (i.e. the combined cost of all three phases), as well as the case where the user is interested in minimizing the costs of the online and post-processing phases only (i.e. where the user assumes that the offline phase can be done overnight for example and is not an important consideration).

Chapter Outline:

In Section 5.2 we summarize some basic definitions and work from other papers which we will utilize. In Section 5.3 we explain how to utilize an MSP defined over a finite field $\mathbb{F}_p$ which computes a given access structure, as a way of doing the same operation over a finite ring $\mathbb{Z}_{p^k}$, for the same prime p. In Section 5.4 we generalize the results on $\mathcal{Q}_2$ MSPs over $\mathbb{F}_p$ of [SW19], to ESPs over $\mathbb{Z}_{p^k}$.

[3]This is a standard result for MSPs over fields, but we have seen no proof for ESPs over finite rings, so we present this construction in an Appendix.

[4]Note, as MPC protocols do not usually work *in practice* over arithmetic circuits this is only an approximation of the cost of the various options.

This enables us to open values to players, and ensure the opened values are "correct". Then in Section 5.5 we examine various methodologies for checking whether multiplication triples are correct or not. In this section we generalize a number of prior checking procedures to the full generality of working modulo $\mathbb{Z}_{p^k}$. Finally, in Section 5.6 (resp. Section 5.7) we examine the various offline (resp. complete) protocols and do a comparison.

5.2 Preliminaries

5.2.1 Notation

We let $\mathbb{F}$ denote a general finite field, and R denote a general finite commutative ring. We let $\mathbb{F}_p$ denote the specific finite field of p elements, and $\mathbb{Z}_{p^k}$ denote the ring of integers modulo p^k. For two sets X, Y we write $X \subset Y$ if X is a proper subset and $X \subseteq Y$ if X is not necessarily proper. For a set B, we denote by $a \leftarrow B$ the process of drawing a from B with a uniform distribution on the set B. For a probabilistic algorithm A, we denote by $a \leftarrow A$ the process of assigning a the output of algorithm A; with the underlying probability distribution being determined by the random coins of A.

For a vector $\mathbf{x}$ we let $\mathbf{x}^{(i)}$ denote it ith component, and for two vectors $\mathbf{x}$ and $\mathbf{y}$ of the same length we let $\langle \mathbf{x}, \mathbf{y} \rangle$ denote the dot-product, unless otherwise noted. We let $M_{n\times m}(K)$, where $K = \mathbb{F}$ or $K = R$, be the set of all matrices with n rows and m columns. For $M \in M_{n\times m}(K)$ denote the transpose by M^T. We let $\ker(M)$ to denote the subspace of K^m which maps to $\mathbf{0}$ under left multiplication by M, and we let $\mathrm{Im}(M)$ to be the subspace of K^n which is the image of all elements in K^n upon left multiplication by M. If V is a subspace of K^r for some r, we let $V^\perp = \{\mathbf{w} \in K^r \mid \forall \mathbf{v} \in V : \langle \mathbf{w}, \mathbf{v} \rangle = 0\}$ denote the orthogonal complement. Moreover, we let $\mathbf{0}$ and $\mathbf{1}$ be the all zero and all one vector of appropriate dimension (defined by the context unless explicitly specified) and let $\mathbf{e}_i$ be the ith canonical basis vector, that is $\mathbf{e}_i^{(j)} = \delta_{i,j}$ where δ is the Kronecker Delta.

5.2.2 ℓ-Good Rings and the Schwartz-Zippel Lemma

Following Fehr [Feh98], a ring R is said to be ℓ-good if there is a set S of ℓ units $\omega_i \in R^*$ such that

$$\omega_i - \omega_j \in R^* \text{ for all } \omega_i, \omega_j \in S \text{ such that } \omega_i \neq \omega_j.$$

It is known that a ring R is ℓ-good if and only if $\ell \leq |R/\mathfrak{m}_1| - 1$, where $\mathfrak{m}_1$ is the largest maximal ideal contained in R. If this is not the case then R can be extended to an ℓ-good Galois ring, [Feh98]. In particular if R is a ring $\mathbb{Z}_{p^k}$ then one can select a polynomial $F(X) \in \mathbb{F}_p[X]$ which is irreducible (over $\mathbb{F}_p[X]$) and of degree d_ℓ such that $\ell \leq p^{d_\ell} - 1$. Then one forms the Galois ring $\overline{R} = \mathbb{Z}_{p^k}[X]/F(X)$, which will be ℓ-good, with a set S being the embedding of the units of $\mathbb{F}_p[X]/F(X)$ into the ring $\overline{R}$.

This Galois ring extension $\overline{R}$ allows us to define a variant of the Schwartz-Zippel Lemma; we present here a univariate version as that is all we will need, a multivariate version follows by the standard argument.

Lemma 5.1: Schwartz-Zippel Lemma over Rings

Let $F \in R[X]$ denote a non-zero polynomial of degree d. Let $\overline{R}$ denote a Galois ring extension of R which is ℓ-good, with the set S of size ℓ as above. If one selects $r \in S$ uniformly at random then we have

$$\Pr[\ F(r) = 0\ ] \leq \frac{d}{\ell}.$$

Proof. We prove the result by showing that the polynomial $F(X)$ can have at most d roots in S. The proof follows by a simple induction on d. The case of $d = 0$, i.e. constant polynomials is trivial.

Now assume that all polynomials of degree $d - 1$ have at most $d - 1$ roots in $S \subset \overline{R}$. Consider a polynomial $F(X)$ of degree d and assume it has $d + 1$ distinct roots, $\omega_1, \ldots, \omega_{d+1} \in S$. We can then write $F(X) = (X - \omega_{d+1}) \cdot G(X)$, where $G(X)$ is of degree d. Now all ω_i with $i \neq d+1$ are roots of $F(X)$, but by assumption we have $\omega_i - \omega_{d+1}$ is a unit in $\overline{R}$. This means that ω_i with $i \neq d+1$ must be a root of $G(X)$, which contradicts the assumption that $G(X)$ has at most d roots. □

5.2.3 Monotone and Extended Span Programs

As is standard we can associate linear secret sharing schemes over fields with Monotone Span Programs. In [Feh98] these definitions are extended to linear secret sharing schemes over finite rings, such as $\mathbb{Z}_{p^k}$, with the associated structure being called an Extended Span Program. We recap on the relevant definitions here.

Access Structures:

The set of parties that the adversary can corrupt is drawn from an access structure (Γ, Δ). The set Γ is the set of all qualified sets, whilst Δ is the set of all unqualified sets. The access/adversary structure is assumed to be monotone, i.e. if $X \subset X'$ and $X \in \Gamma$, then $X' \in \Gamma$ and if $X \subset X'$ and $X' \in \Delta$ then $X \in \Delta$, and we assume $\Gamma \cap \Delta = \emptyset$. We are only interested in this chapter in access structures which are $\mathcal{Q}_2$:

Definition 5.1: $\mathcal{Q}_2$ Access Structure

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of parties, with access structure (Γ, Δ), then (Γ, Δ) is said to be a $\mathcal{Q}_2$ access structure if

$$P \neq A \cup B \text{ for all } A, B \in \Delta.$$

In other words: An access structure (Γ, Δ) is $\mathcal{Q}_2$, if for any two sets in Δ the union of those sets does not cover $\mathcal{P}$. An access structure is called complete if for any $Q \in \Gamma$ it holds that $\mathcal{P} \backslash Q \in \Delta$ and vice versa. In this chapter we will only consider complete access structures.

Monotone Span Programs over Fields:

Using this notation, the definition of a Monotone Span Program follows.

Definition 5.2: MSP

A Monotone Span Program (MSP), denoted $\mathcal{M}$, is a quadruple $(\mathbb{F}, M, \varepsilon, \varphi)$, where $\mathbb{F}$ is a field, $M \in M_{m \times d}(\mathbb{F})$ is a full-rank matrix for some m and $d \leq m$, $\varepsilon \in \mathbb{F}^d$ is an arbitrary non-zero vector called the target vector, and $\varphi : [m] \to \mathcal{P}$ is a surjective map of the rows of M to the parties in $\mathcal{P}$. The size of $\mathcal{M}$ is defined to be m, the number of rows of the matrix M.

Given a set of parties $\mathcal{S} \subseteq \mathcal{P}$, the submatrix $M_{\mathcal{S}}$ is the matrix whose rows are indexed by the set $\{i \in [m] : \varphi(i) \in \mathcal{S}\}$. Similarly, $\mathbf{s}_{\mathcal{S}}$ is the vector whose rows are indexed by the same set. We also define the supp-mapping, which maps the rows of a matrix M to a player in $\mathcal{P}$. Formally this is defined as $\mathsf{supp} : \mathbb{F}^d \to 2^{[d]}$ with $\mathbf{s} \mapsto \{i \in [d] : \mathbf{s}^{(i)} \neq 0\}$.

Definition 5.3: MSP computing (Γ, Δ)

An MSP $\mathcal{M}$ is then said to compute an access structure (Γ, Δ) if for every set $A \subset 2^{\mathcal{P}}$ it holds that

$$A \in \Gamma \Rightarrow \varepsilon \in \mathrm{Im}(M_A^T), \tag{5.1}$$

$$A \notin \Gamma \Rightarrow \varepsilon \notin \mathrm{Im}(M_A^T). \tag{5.2}$$

Note that this could be presented as a single if and only if condition, but to emphasize the difference with the generalization to rings, we present the condition in two equations. Also note that, an equivalent formulation for requirement (5.2) is the following:

$$A \notin \Gamma \Rightarrow \exists \mathbf{k} \in \ker(M_A) \text{ s.t. } \langle \varepsilon, \mathbf{k} \rangle \neq 0$$

Extended Span Programs over Rings:

In this chapter we are not only interested in the Monotone Span Programs, but also their extensions to finite rings, which are known as Extended Span Programs, [Feh98]. An Extended Span Program (ESP) over a ring R is a tuple $\mathcal{M} = (R, M, \varepsilon, \varphi)$ where $M \in M_{m \times d}(R)$ is a full-rank matrix for some m and $d \leq m$, $\varepsilon \in R^d$ is an arbitrary non-zero vector called the target vector, and $\varphi : [m] \to \mathcal{P}$ is a surjective map of the rows of M to the parties in $\mathcal{P}$. The analogue of Definition 5.3 is

Definition 5.4: ESP computing (Γ, Δ)

An ESP $\mathcal{M}$ is said to compute an access structure (Γ, Δ) if for every set $A \subset 2^{\mathcal{P}}$ it holds that

$$A \in \Gamma \Rightarrow \varepsilon \in \mathrm{Im}(M_A^T), \tag{5.3}$$

$$A \notin \Gamma \Rightarrow \exists \mathbf{v} \in \ker(M_A) \subset R^d : \langle \varepsilon, \mathbf{v} \rangle \in R^*. \tag{5.4}$$

Note that, (5.4) is a stronger assumption than the corresponding requirement for an MSP in the ESP case, namely (5.2). To see this note that if $\varepsilon = (\varepsilon_1, \varepsilon_2, \dots, \varepsilon_d)$ with $\varepsilon_i \notin R^*$ then there are situations in which $A \notin \Gamma$, however $\varepsilon \in \mathrm{Im}(M_A^T)$.

For the rest of this chapter we will only be considering MSPs over finite fields $\mathbb{F}_p$, or ESPs over the finite ring $\mathbb{Z}_{p^k}$. Let $\mathcal{P} = \{P_1, \dots, P_n\}$ be the set of parties involved in our protocols. To implement our MPC functionality over $\mathbb{Z}_{p^k}$ we will utilize an ESP $(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ given by a matrix $M \in \mathbb{Z}^{m\times d}$, such that $M = M \pmod{p}$ (i.e. the entries of M are in the range $[0, \dots, p)$), such that to share a value $x \in \mathbb{Z}_{p^k}$ one generates a vector $\mathbf{k} \in \mathbb{Z}_{p^k}^d$ such that $\langle \varepsilon, \mathbf{k}\rangle = x \pmod{p^k}$ and then compute the share values $\mathbf{s} = M \cdot \mathbf{k}$. The entries of $\mathbf{s}$ are passed to the players depending on the value of the function $\varphi : [m] \to \mathcal{P}$. i.e. player P_i gets $\mathbf{s}^{(j)}$ if $\varphi(j) = i$. Such a sharing $x \in \mathbb{Z}_{p^k}$ of a value will be denoted by $[x]_k$, note the subscript k which will be used to keep track of which ring we are considering at any given point.

5.2.4 Linear Secret Sharing Schemes Induced from MSPs and ESPs

When you have a Monotone/Extended Span Program it induces a Linear Secret Sharing Scheme (LSSS) using the method in Figure 5.1. Recombination works for qualified sets $A \in \Gamma$, since if A is qualified there exists a recombination vector λ_A such that $M_A^T \cdot \lambda_A = \varepsilon$, by requirement (5.3) of the MSP. Hence,

$$\langle \lambda_A, \mathbf{s}_A\rangle = \langle \lambda, \mathbf{s}\rangle = \langle \lambda, M \cdot \mathbf{x}\rangle = \langle M^T \cdot \lambda, \mathbf{x}\rangle = \langle \varepsilon, \mathbf{x}\rangle = s.$$

Conversely, if $A \notin \Gamma$ then A is unqualified, hence by requirement (5.2) of the MSP, or requirement (5.4) of the ESP, there is no λ that allows for reconstruction. That reconstruction vectors exist follows from the following two Lemmas, since $\mathbb{Z}$ is a Euclidean domain, and so Lemma 5.3 implies that we can solve linear equations in the quotient rings $\mathbb{Z}_{p^k}$.

Lemma 5.2

There exists an algorithm which solves linear equations, $\langle \mathbf{a}, \mathbf{x}\rangle = \mathbf{b}$, for any Euclidean domain D. Moreover, there exists an algorithm that solves linear equation systems $M \cdot \mathbf{x} = \mathbf{b}$ for any matrix $M \in M_{n\times m}(D)$.

Lemma 5.3

If linear equation systems can be solved in the ring R, then they can also be solved in the rings $R[X]$ and $R/\mathfrak{I}$ for all finitely generated ideals $\mathfrak{I}$ of R.

We note that the reconstruction step 2 can be relatively expensive for large MSPs, i.e. those with large m. Thus, it is common to only send “just enough” information to each player in order to allow reconstruction. How this is done in a manner which prevents active attacks is discussed in Section 5.4.

Induced LSSS from an MSP/ESP

Given a Monotone/Extended Span Program, $\mathcal{M} = \{\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\}$ and a secret s, distribution and reconstruction for the associated secret sharing scheme are as follows:

Distribution:

1. Sample $\mathbf{x} \leftarrow \mathbb{Z}_{p^k}^d$ under the condition that $\langle \mathbf{x}, \varepsilon \rangle = s$.
2. Compute $\mathbf{s} = M \cdot \mathbf{x}$, such that $\mathbf{s} = (s_1 s_2 \ldots s_n)$ and distribute each s_i to the party indicated by $\varphi(i)$, such that each party P_j has the vector
$$\mathbf{s}_{P_j} = \begin{cases} s_i & \varphi(i) = P_j \\ 0 & \text{otherwise} \end{cases}$$

Reconstruction: Let $A \in \Gamma$ be a qualified set of players:

1. Define λ_A such that $M_A^T \cdot \lambda_A = \varepsilon$.
2. Each player $P_i \in A$ sends their shares to all other $P_j \in A$ and computes $\mathbf{s}_A = \sum_{P_i \in A} s_{P_i}$.
3. Compute $s^* = \langle \mathbf{s}_Q, \lambda_Q \rangle$.
4. Return s^*.

Figure 5.1: Induced LSSS from a Monotone/Extended Span Program.

Multiplicative Linear Secret Sharing Scheme

A secret sharing scheme induced from a MSP/ESP is by definition linear, i.e. one can compute arbitrary linear functions of secret shared values without interaction. $\mathcal{Q}_2$ access structures are interesting as they allow us to also multiply secret shared values, but using interaction, if the underlying LSSS is multiplicative.

Recall a vector $\mathbf{s} = (s_i) = M \cdot \mathbf{k}$ is some sharing of a value s if we have that $\langle \varepsilon, \mathbf{k} \rangle = s$, with the shares distributed to party P_i being $\mathbf{s}_i = (s_j)_{\varphi(j)=i}$. We let the total number of shares held by party P_i be given by n_i. The local *Schur product* of two sharings $\mathbf{x}_i$ and $\mathbf{y}_i$ of values x and y for party P_i are the n_i^2 terms given by $\mathbf{x}_i \otimes \mathbf{y}_i$, i.e. the terms $p_{i,j} = \mathbf{x}_i^{(v)} \cdot \mathbf{y}_i^{(v')}$ for $j = 1, \ldots, n_i^2$ and v, v' range over all values for which $\varphi(v) = \varphi(v') = i$. An MSP is said to be *multiplicative* if there are constants $\mu_{i,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, n_i^2$ such that

$$x \cdot y = \sum_{i,j} \mu_{i,j} \cdot p_{i,j} \tag{5.5}$$

for all valid sharings of x and y. By abuse of notation we shall refer to the MSP/ESP being multiplicative, and not just the induced LSSS.

Many "natural" MSPs/ESPs computing $\mathcal{Q}_2$ access structures are multiplicative, i.e. those arising from Shamir secret sharing, or replicated sharing. It is well known, see [CDM00], that when you have a non-multiplicative MSP over a field that computes a $\mathcal{Q}_2$ access structure then it can be made multiplicative with only a small expansion of the dimensions of M. In Appendix 5.A we prove the following theorem, generalizing this result to ESPs over $\mathbb{Z}_{p^k}$,

Theorem 5.1

There exists an algorithm which, on input of a non-multiplicative ESP $\mathcal{M}$ over $\mathbb{Z}_{p^k}$ computing a $\mathcal{Q}_2$ access structure (Γ, Δ) outputs a multiplicative ESP $\mathcal{M}'$ computing Γ and of size at most $4 \cdot |\mathcal{M}|$. This algorithm is effective if $\ker(M^T)$ admits a basis.

5.2.5 Shamir over Rings, an Example:

To help solidify ideas we present here the standard construction of Shamir secret sharing over a small finite field (say $\mathbb{F}_2$), which is achieved via extension to a finite field $\mathbb{F}_{2^{d_n}}$, where $n \leq 2^{d_n} - 1$ We then show how this can be interpreted as an MSP over the finite field $\mathbb{F}_2$, where we only want to share elements in $\mathbb{F}_2$ and not $\mathbb{F}_{2^{d_n}}$. By extending scalars we then obtain an ESP over the ring $\mathbb{Z}_{2^k}$.

Consider first constructing an analogue of Shamir sharing for three players and threshold one[5] over the finite field $\mathbb{F}_2$, i.e. $(n,t) = (3,1)$. The problem with Shamir over $\mathbb{F}_2$, is that we do not have enough elements to interpolate via n evaluations. Thus, we need to extend the base field by a degree d_n extension so that it is n-good (see [Feh98]), Since $n = 3 = 2^2 - 1 = p^{d_3} - 1$ we simply need to take an extension of degree two. Thus, we set $K = \mathbb{F}_2[X]/(X^2 + X + 1)$ and we represent elements in K via $a_0 + a_1 \cdot \theta$ for a variable θ such that $\theta^2 + \theta + 1 = 0$. The set S being the set $\{1, \theta, \theta + 1\}$.

[5]The astute reader will be asking why bother? A simpler implementation in this case comes from replicated sharing. We give this example since, for large values of n and t, the construction via extensions fields/rings is more efficient than replicated sharing.

We now perform the standard Shamir sharing technique for $t = 1$. To share a secret $s = s_0 + s_1 \cdot \theta \in K$, we select a polynomial $f(X) = (s_0 + s_1 \cdot \theta) + (a_0 + a_1 \cdot \theta) \cdot X$, where $a_0, a_1 \in \mathbb{F}_2$ and generate the shares by evaluating $f(X)$ at the elements in S. We can then express this as an MSP over $\mathbb{F}_2$ by treating the coefficients of θ as separate shares. The MSP can be simplified a little, as we are only interested in sharings of elements in $\mathbb{F}_2$; thus we will always have $s_1 = 0$. Interpolation is then always possible via Lagrange interpolation as the denominators in the Lagrange coefficients, $\omega_i - \omega_j$, are always invertible via the choice of the set S.

Thus shares for player one, corresponding to the element $\omega_1 = 1 \in S$ are $\{ s_0 + a_0,\ a_1 \}$; the shares for player two, corresponding to the element $\omega_2 = \theta \in S$ are $\{ s_0 + a_1,\ a_0 + a_1 \}$; whilst the shares for player three, corresponding to the element $\omega_3 = \theta + 1 \in S$ are $\{ s_0 + a_0 + a_1,\ a_0 \}$. We can then write the secret sharing scheme down as an MSP over $\mathbb{F}_2$, as $\mathcal{M}_2 = (\mathbb{F}_2, M, \mathbf{e}_1, \varphi)$, where the matrix M is given by

$$M = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

and $\varphi(i) = \lceil i/2 \rceil$.

The self same construction, but starting with the ring $\mathbb{Z}_{p^k}$ will create the ESP $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \mathbf{e}_1, \varphi)$. We see that we can consider $\mathcal{M}_2$ as the reduction modulo 2 of the ESP $\mathcal{M}$, or we can consider $\mathcal{M}$ as the "lift" (by extension of scalars) of the MSP $\mathcal{M}_2$. Both compute the same access structure, and both are multiplicative.

This method of forming ESPs over rings $\mathbb{Z}_{p^k}$ for an access structure Γ, by first forming an MSP $\mathcal{M}_p$ over the field $\mathbb{F}_p$ for the same access structure Γ and then "lifting" the MSP to the ESP over the required ring $\mathbb{Z}_{p^k}$, will be our method for constructing ESPs in this work. We show later, Section 5.3, that this lifting always works in terms of the access structure, but we cannot show that the associated lift is always multiplicative (we conjecture that it is for all "interesting" in practice MSPs/ESPs).

Note, by generating the MSP via Galois ring extensions, but then restricting the shared value to the base ring, and also encoding all the ring extension arithmetic within the matrix M, means we can dispense with considering Galois rings as soon as the MSP is constructed. This avoids the complexity mentioned in [ACD+19][Section 3.4] of us never needing to worry about a reconstructed value is not in the base ring.

5.2.6 Basic Multi-Party Computation Protocols

The general MPC functionality that we aim to implement is given in Figure 5.2. We assume the secret sharing scheme defined by the ESP is multiplicative, and hence the underlying secret sharing scheme is $\mathcal{Q}_2$. For example purposes, the reader may want to consider three party replicated sharing for the threshold structure of $(n, t) = (3, 1)$. Here a value s is shared by $s = s_1 + s_2 + s_3$, with party P_i holding the two values $\{s_1, s_2, s_3\} \setminus \{s_i\}$, or our earlier Shamir based example for the same access structure.

Modular reduction is consistent with the opening procedure, in the sense that if $0 \leq k' \leq k$ then the operation of opening a sharing $[x]_k$ and taking the reduction modulo $p^{k'}$ commutes with the operation of reducing all the share values themselves modulo $p^{k'}$. We denote the latter operation by $[x]_{k'} \leftarrow [x]_k \pmod{p^{k'}}$. That the operation commutes follows from our definition of the ESP above.

There are some operations on secret shared values which we can immediately define. We summarize them here as they will be utilized throughout. Many protocols will make use of a cryptographic hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{|\mathcal{H}|}$ which we will model as a random oracle. The interface (API) for the hash function $\mathcal{H}$ will be the standard one provided by cryptographic hash functions in practice; as given in Figure 5.3

Commitment and Decommitment

We can implement the ideal functionality $\mathcal{F}_{\text{Commit}}$ given in Figure 5.4 using the protocol given in Figure 5.5, which utilizes the hash function/random oracle $\mathcal{H}$.

Agreeing on a random value:

In many instances we want to agree a random value from a domain D. This is easily done by each party P_i committing to a random bit string r_i, using the functionality $\mathcal{F}_{\text{Commit}}$, and then the committed values are opened. A seed is then produced via $r = r_1 \oplus \ldots \oplus r_n$, and finally the seed is used to generate random elements from D using a PRF with co-domain D. We shall denote this functionality by $\mathcal{F}_{\text{AgreeRandom}}(D)$ in Figure 5.6

Functionality $\mathcal{F}_{\text{Online}}$

The functionality runs with parties $P_1, \ldots, P_n$ and an ideal adversary. Let $\mathcal{A}$ be the set of corrupt parties. Given a set I of valid identifiers, all values are stored in the form (varid, x), where $\mathit{varid} \in I$.

Initialize: On input (Init, p, k) from all parties, with p a prime and k a positive integer, the functionality stores p^k. The adversary is assumed to have statically corrupted a subset $\mathcal{A}$ of the parties.

Input: This takes input $(\mathit{Input}, P_i, \mathit{varid}, x)$ from P_i, with $x \in \mathbb{Z}_{p^k}$, and $(\mathit{input}, P_i, \mathit{varid}, ?)$ from all other parties, with *varid* a fresh identifier. If the *varid*'s are the same the functionality stores (varid, x), otherwise it aborts.

Add: On command $(\mathit{Add}, \mathit{varid}_1, \mathit{varid}_2, \mathit{varid}_3)$ from all parties:

1. If $\mathit{varid}_1, \mathit{varid}_2$ are not present in memory or varid_3 is then the functionality aborts.
2. The functionality retrieves (varid_1, x), (varid_2, y) and stores $(\mathit{varid}_3, x + y)$.

Multiply: On input $(\mathit{Multiply}, \mathit{varid}_1, \mathit{varid}_2, \mathit{varid}_3)$ from all parties:

1. If $\mathit{varid}_1, \mathit{varid}_2$ are not present in memory or varid_3 is then the functionality aborts.
2. The functionality retrieves (varid_1, x), (varid_2, y) and stores $(\mathit{varid}_3, x \cdot y)$.

Output: On input $(\mathit{Output}, \mathit{varid}, i)$ from all parties (if *varid* is present in memory),

1. The functionality retrieves (varid, y).
2. If $i = 0$ and $\mathcal{A} \neq \emptyset$ then the functionality outputs y to the adversary, otherwise it outputs $\perp$ to the adversary.
3. The functionality waits for an input from the adversary.
4. If this input is Deliver then y is output to all players if $i = 0$, or y is output to player i if $i \neq 0$.
5. If the adversarial input is not equal to Deliver then abort.

Figure 5.2: The ideal functionality for MPC with Abort over $\mathbb{Z}_{p^k}$

Interface for a Cryptographic Hash Function $\mathcal{H}$

Let $\mathcal{H} : \{0,1\}^* \longrightarrow \{0,1\}^{|\mathcal{H}|}$ be a cryptographic hash function, then the $\mathcal{H}$ function object has three member functions associated with it:

- $H.\mathsf{Init}()$: Initializes the hash function.
- $H.\mathsf{Update}(\mathbf{s})$: Updates the hash function's internal state with the bit-vector $\mathbf{s}$.
- $H.\mathsf{Out}()$: Evaluates the hash function and outputs the result.

Figure 5.3: Interface for a Cryptographic Hash Function $\mathcal{H}$

The Ideal Functionality $\mathcal{F}_{\text{Commit}}$

Commit: On input $(\mathsf{Commit}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all players and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all players and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all players.

Figure 5.4: The Ideal Functionality for Commitments

Sharing a value:

If party P_i wants to share a value s it uniformly at random selects $\mathbf{k} \in \mathbb{Z}_{p^k}^k$ such that $\langle \varepsilon, \mathbf{k} \rangle = x \pmod{p^k}$, computes $\mathbf{s} = M \cdot \mathbf{k}$ and sends $\mathbf{s}^{(j)}$ to player i if $\varphi(j) = i$. We will denote this operation in our protocols by $[x]_k \leftarrow \mathsf{Share}(x, i, k)$.

Linear Operations:

Linear operations on secret shared values can be performed by applying the *same* linear operation to the shared values; where we interpret a constant value

The Protocol Π_{Commit}

Commit:

1. In order to commit to v, P_i sets $o \leftarrow v||r$, where r is chosen uniformly in a determined domain, and queries the Random Oracle $\mathcal{H}$ to get $c \leftarrow \mathcal{H}(o)$.
2. Player P_i then broadcasts (c, i, τ_v), where τ_v represents a handle for the commitment.

Open:

1. In order to open a commitment (c, i, τ_v), where $c = \mathcal{H}(v||r)$, player P_i broadcasts $(o = v||r, i, \tau_v)$.
2. All players call $\mathcal{H}$ on o and check whether $\mathcal{H}(o) = c$. Players accept if and only if this check passes.

Figure 5.5: The Protocol for Commitments.

Ideal Functionality $\mathcal{F}_{\text{AgreeRandom}}(D)$

On input $\mathsf{AgreeRandom}(\mathsf{cnt})$ from all parties, if the counter value is the same for all parties and has not been used before, the functionality samples a value $a \leftarrow D$, and sends a to all parties.

Figure 5.6: Ideal Functionality $\mathcal{F}_{\text{AgreeRandom}}(D)$

c as shared by the vector $c \cdot [1]_k = M \cdot \mathbf{k}_{\mathsf{one}}$ where $\mathbf{k}_{\mathsf{one}}$ is a fixed vector such that $\langle \varepsilon, \mathbf{k}_{\mathsf{one}} \rangle = 1$.

Sharing a random value:

If the number of maximally unqualified sets is small then this can be done, in a computationally secure non-interactive manner, by pre-distributing secret keys corresponding to the access structure and using a standard Pseudo-Random-Secret-Sharing (PRSS) construction to enable each party to obtain a random value [CDI05]. If the number of such sets if large then one can obtain an interactive, information theoretically secure manner, by each party P_i generating

$r_i \in \mathbb{Z}_{p^k}$, and then executing $[r_i]_k \leftarrow \mathsf{Share}(r_i, i, k)$. The resulting shared value being $\sum [r_i]_k$. We denote this functionality by $\mathcal{F}_{\text{PRSS}}$, which is given in Figure 5.7

Ideal Functionality $\mathcal{F}_{\text{PRSS}}(m)$

On input $\mathsf{PRSS}(\mathsf{cnt})$ from all parties, if the counter value is the same for all parties and has not been used before, the functionality samples a value $a \leftarrow \mathbb{Z}_{p^k}$, computes a sharing $[a]_k$ and sends the respected share values to the designated player.

Figure 5.7: Ideal Functionality $\mathcal{F}_{\text{PRSS}}(m)$

Passively Secure Multiplication:

We will utilize four forms of passively multiplication routine; all of which are actively secure up to an additive attack. The first is classic Beaver multiplication. This requires one round of interaction, requires the consumption of a multiplication triple, and requires two executions of $\mathsf{OpenToAll}$ (see later for how we define this protocol, which implements the $\mathsf{Output}([x]_k, 0)$ operation in the functionality in Figure 5.2). The protocol is passively secure if the underlying triple is only passively secure, and actively secure otherwise[6]. We refer to this protocol as $[z]_k \leftarrow \mathsf{BeaverMult}([x]_k, [y]_k)$.

The second is the classic passively secure multiply-and-reshare operation (which we call Maurer-multiplication as it seems to have been first given in full generality in [Mau06]). This requires one round of communication, no multiplication triples, and requires each player to execute Share on their local multiplication. This protocol is only passively secure. We refer to this protocol as $[z]_k \leftarrow \mathsf{MaurerMult}([x]_k, [y]_k)$.

The third technique is the method of [SW19, KRSW18] which is the generalization to arbitrary multiplicative LSSS of the classic multiplication algorithm for replicated $(n, t) = (3, 1)$ sharing. The paper [KRSW18] gives this for arbitrary replicated MSPs, whilst [SW19] generalizes this to an arbitrary $\mathcal{Q}_2$ multiplicative MSP. Again one round of interaction is required and no multiplication triples are consumed. The protocol requires access to a (modified) form of PRSS/PRZS protocol, and the amount of data sent depends highly on the specific secret sharing scheme being executed. This protocol is only passively secure. We refer to this protocol as $[z]_k \leftarrow \mathsf{KRSWMult}([x]_k, [y]_k)$.

[6] By which we mean that the triple is guaranteed to be correct

Our fourth and final technique is a generalization of [DN07] from honest-majority Shamir secret sharing to the setting of an arbitrary $\mathcal{Q}_2$ ESP. It relies on pairs of the form $([r]_k, \langle r \rangle)$, where $\langle r \rangle$ represents an additive sharing of r. Similarly to KRSWMult, the protocol requires access to a PRSS and PRZS protocol. Under certain assumptions, we can generate these pairs silently and perform a single multiplication with $n \cdot (n-1)$ ring elements of communication and a single round of communication. Contrary to the original design of [DN07], we do not use the king paradigm (which would make the communication be only linear in the number of players), as this doubles the number of rounds needed for a multiplication. We refer to this protocol as $[z]_k \leftarrow \mathsf{DNMult}([x]_k, [y]_k)$.

To perform an execution of our DNMult protocol for an arbitrary $\mathcal{Q}_2$ ESP, we proceed in two phases: first a pair $([r]_k, \langle r \rangle)$, is (silently) generated, afterwards it is used to transfer an additive sharing of the product (obtained via the usual local Schur multiplication of shares) back into a $\mathcal{Q}_2$ sharing, similar to the high-level approach of KRSWMult. To generate a pair, we assume the PRSS and PRZS protocols can be executed without communication. First, generate $[r]_k$ using the PRSS, such that each player P_i holds a vector of shares $\mathbf{r}_i$ and choose a fixed reconstruction vector λ such that

$$r = \sum_{i=1}^{n} \sum_{j=1}^{n_i} \lambda_{i,j} \cdot \mathbf{r}_i^{(j)}.$$

Also generate an additive sharing of zero $\langle t \rangle \leftarrow \mathsf{PRZS}$. Each player can then locally compute the additive share $r_i^a = \sum_{j=1}^{n_i} \lambda_{i,j} \cdot \mathbf{r}_i^{(j)} + t_i$. Note that since λ is a reconstruction vector, we have that the r_i^a form an additive sharing $\langle r \rangle$.

In the online phase, given the sharings $[x]_k$ and $[y]_k$, the players can locally compute an additive sharing $\langle x \cdot y \rangle$ thanks to the multiplicative property of the ESP. Then a pair $([r]_k, \langle r \rangle)$ is consumed to open the value $v \leftarrow \mathsf{OpenToAll}(\langle r \rangle - \langle x \cdot y \rangle)$. Note that we cannot rely on the properties of a $\mathcal{Q}_2$ ESP to optimize this as we are opening an additive sharing, so we simply have each player broadcast their own share. Given v, the players can then locally compute $[x \cdot y]_= [r]_k - v$.

If we refer to either MaurerMult, KRSWMult or DNMult (our three passively secure multiplication protocols which do not utilize pre-processed triples), without defining precisely which one, we will write $[z]_k \leftarrow \mathsf{PassMult}([x]_k, [y]_k)$.

5.3 Generating an ESP from an MSP

In the sections above we have described how an ESP can be defined, however it is in general not a simple task to define an ESP for a general access structure.

One of the reasons being that dealing with zero-divisors can be tricky and the initial choices of matrix, mapping, and target vector are not as natural as when one considers MSPs.

However, there is a natural construction to generate an ESP over $\mathbb{Z}_{p^k}$ for a given access structure (Γ, Δ). First generate an MSP $\mathcal{M}_p$ over $\mathbb{F}_p$ for the access structure (Γ, Δ), and then "lift" this to an ESP $\mathcal{M}$ over $\mathbb{Z}_{p^k}$. Indeed, one can simply think of $\mathcal{M}_p$ as defining $\mathcal{M}$ exactly. This is exactly what we did in our previous Shamir sharing example, and it was used in [ACD+20] in a similar context (but in the language of lifting the associated code and not the MSP). That this trivial methodology always works is guaranteed by the following theorem.

Theorem 5.2

Let $\mathcal{M}_p = (\mathbb{F}_p, M_p, \varepsilon_p, \varphi)$ be a Monotone Span Program computing the access structure (Γ_p, Δ_p) over $\mathbb{F}_p$. Let $\mathcal{M} = \left(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\right)$ be *any* Extended Span Program computing an access structure (Γ, Δ) over $\mathbb{Z}_{p^k}$ such that $M_p = M \pmod p$ and $\varepsilon_p = \varepsilon \pmod p$. Then $\Gamma = \Gamma_p$ (and hence also $\Delta = \Delta_p$).

Proof. To show that $\Gamma = \Gamma_p$ we show that the conditions on an ESP and MSP are equivalent on the qualified sets. Let Q be such a qualified set, then we show that

$$Q \in \Gamma \Leftrightarrow Q \in \Gamma_p,$$

$$Q \notin \Gamma \Leftrightarrow Q \notin \Gamma_p.$$

By contraposition, it is sufficient to show the implications from Γ to Γ_p.

$\underline{Q \in \Gamma \Rightarrow Q \in \Gamma_p}$: Let $Q \in \Gamma$, then by the first condition on the ESP it holds that $\varepsilon \in \operatorname{Im}(M_Q^T)$, so there exists a $\mathbf{c} \in \mathbb{Z}_{p^k}^r$ such that

$$\mathbf{c} \cdot M_Q^T = \sum_{i=1}^{r} c_i \cdot \mathbf{m}_i = \varepsilon,$$

where $\mathbf{m}_i$ is the i'th column vector of M_Q^T. Hence, by reduction modulo p:

$$\varepsilon_p \equiv \varepsilon \mod p \equiv \sum_{i=1}^{r} c_i \cdot \mathbf{m}_i \mod p \equiv \sum_{i=1}^{r} (c_i \cdot \mathbf{m}_i \mod p) \equiv \mathbf{c}_p \cdot (M_p)_Q^T,$$

where $\mathbf{c}_p^{(i)} = c_i \mod p$. Therefore, $\varepsilon_p \in \operatorname{Im}\left((M_p)_Q^T\right)$ and by the contrapositive of the second condition on the MSP this means that $Q \in \Gamma_p$.

$\underline{Q \notin \Gamma \Rightarrow Q \notin \Gamma_p}$: Reducing the second condition on an ESP modulo p does not change the condition as $\mathbf{a}$ contains at least one entry which is a unit, therefore $\langle \mathbf{a}, \varepsilon \rangle \in \mathbb{Z}_{p^k}^* \Rightarrow \langle \mathbf{a}_p, \varepsilon_p \rangle \in \mathbb{Z}_p^*$ and $\mathbf{a}_p \in \ker\left((M_p)_Q^T\right)$. By the fundamental theorem of linear algebra

$$\ker\left((M_p)_Q\right) = \operatorname{Im}\left((M_p)_Q^T\right)^{\perp},$$

so for all $\mathbf{b}_p \in \operatorname{Im}\left((M_p)_Q^T\right) : \langle \mathbf{a}_p, \mathbf{b}_p \rangle = 0$ and as $\langle \mathbf{a}_p, \varepsilon_p \rangle \neq 0$ we have that $\varepsilon_p \notin \operatorname{Im}\left((M_p)_Q^T\right)$. By the contrapositive to the first condition of the MSP this means $Q \notin \Gamma_p$. □

Using this result we can easily construct ESPs for any $\mathcal{Q}_2$ access structure; and we can transfer efficient constructions of MSPs from $\mathbb{F}_p$ to $\mathbb{Z}_{p^k}$ in a straight forward fashion. The question that arises as to whether the resulting ESP over $\mathbb{Z}_{p^k}$ is multiplicative if the original MSP over $\mathbb{F}_p$ was multiplicative.

Given an $\mathcal{M}_p = (\mathbb{F}_p, M_p, \varepsilon_p, \varphi)$, i.e. $M_p \in \mathbb{F}_p^{m \times d}$, $\varepsilon_p \in \mathbb{F}_p^d$ and $\varphi : [m] \to \mathcal{P}$. We define the "natural lift" of $\mathcal{M}_p$ to the ring $\mathbb{Z}_{p^k}$ to be the ESP $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ where $M = M_p$ and $\varepsilon = \varepsilon_p$ *over the integers*. Recall an ESP is multiplicative if one can find a solution to equation (5.5) modulo p^k. It is clear that if $\mathcal{M}$ is multiplicative then so is $\mathcal{M}_p$, the converse may not necessarily hold, although for all "natural" constructions which would seem to arise in practical applications this is indeed so. However, this is not true in general (see [ACD+20] for a counter example). If one is so-unlucky to construct an ESP which is not multiplicative one can always extend it to a multiplicative one using the method of Theorem 5.1.

To concretely see why $\mathcal{M}_p$ can be multiplicative but $\mathcal{M}$ may not be, we write $x = \langle \varepsilon, \mathbf{k}_x \rangle$ and $y = \langle \varepsilon, \mathbf{k}_y \rangle$ for some vectors $\mathbf{k}_x$ and $\mathbf{k}_y$, where we think of the values in $\mathbf{k}_x$ and $\mathbf{k}_y$ as $2 \cdot k$ variables over the integers. Again letting the number of shares held by P_i be n_i, then the local Schur product for player P_i is a sum of terms $p_{i,j} = \mathbf{s}_x^{(v)} \cdot \mathbf{s}_y^{(v')}$ for $j = 1, \ldots, n_i^2$ and v, v' range over all values for which $\varphi(v) = \varphi(v') = i$. The terms $p_{i,j}$ are (over the integers) equal to the product of two linear forms in the variables $\mathbf{k}_x$ and $\mathbf{k}_y$, as $\mathbf{s}_x^{(v)}$ is a linear form in $\mathbf{k}_x$ and $\mathbf{s}_y^{(v')}$ is a linear form in $\mathbf{k}_y$. These linear forms have coefficients are in the range $[0, \ldots, p)$, and so we can write

$$p_{i,j} = \sum_{v,v'=1}^{d} a_{i,j,v,v'} \cdot \mathbf{k}_x^{(v')} \cdot \mathbf{k}_y^{(v')}$$

where $a_{i,j,v,v'} \in [0,\ldots,p^2)$. The value $x \cdot y$ can be written in a similar way as

$$\sum_{v,v'=1}^{d} b_{v,v'} \cdot \mathbf{k}_x^{(v')} \cdot \mathbf{k}_y^{(v')}$$

with $b_{v,v'} \in [0,\ldots,p^2)$.

By equating coefficients of $\mathbf{k}_x^{(v')} \cdot \mathbf{k}_y^{(v')}$ on both sides, that an MSP is multiplicative modulo p^k is equivalent to there being a solution to the linear system of equations

$$A \cdot \underline{\mu} = \mathbf{b} \pmod{p^k} \tag{5.6}$$

for a matrix $A \in \mathbb{Z}^{s\times t}$ and a vector $\mathbf{b} \in \mathbb{Z}^s$, both of whose coefficients are in $[0,\ldots,p^2)$, where $s = d^2$ and $t = \sum n_i^2$. If we now compute the Smith Normal Form of A, that is we find two matrices $U \in \mathsf{GL}_s(\mathbb{Z})$ and $V \in \mathsf{GL}_t(\mathbb{Z})$, i.e. the matrices of determinant ± 1, such that

$$U \cdot A \cdot V = S = \begin{pmatrix} s_1 & 0 & \ldots & & \ldots & 0 \\ 0 & s_2 & & & & \vdots \\ \vdots & & \ddots & & & \vdots \\ \vdots & & & s_r & & \vdots \\ \vdots & & & & 0 & \vdots \\ \vdots & & & & \ddots & \vdots \\ 0 & \ldots & & \ldots & \ldots & 0 \end{pmatrix}$$

where r is the rank of A *over the integers* and $s_i | s_{i+1}$ for all i. We set $\underline{\nu} = V^{-1} \cdot \underline{\mu}$ and can write out equations as $S \cdot \underline{\nu} = U \cdot \mathbf{b}$. This will have a solution modulo p^k if and only if equation (5.6) has a solution modulo p^k.

If we write $U \cdot \mathbf{b} = (c_1,\ldots,c_s)^\mathsf{T}$ then we have a solution modulo p^k to this equation, if and only if

1. For $1 \le i \le r$ either $\mathrm{ord}_p(s_i) \le \mathrm{ord}_p(c_i)$ or $\min(\mathrm{ord}_p(s_i), \mathrm{ord}_p(c_i)) \ge k$
2. For $r < i \le s$ we have $\mathrm{ord}_p(c_i) \ge k$,

where $\mathrm{ord}_p(x)$ is the largest power of p dividing x. Thus, we can see that it appears possible that $\mathcal{M}_p$ is multiplicative modulo p, but using the same matrix for the MSP $\mathcal{M}$ may lead to a non-multiplicative MSP modulo p^k.

5.4 Opening Values to One Player and to All Players

The key cost in an LSSS-based MPC protocol, and essentially the only place where an active adversary can introduce errors, is when a secret shared value is opened to one player or to all players; i.e. in the realization of the Output command in Figure 5.2. Since we are utilizing $\mathcal{Q}_2$ access structures, we can make use of the properties of $\mathcal{Q}_2$ access structures to *detect* errors introduced by the adversary. In [SW19] it was shown that in the case of opening to one player one could use the parity check matrix of the underlying code associated to the secret sharing scheme to perform this check.

In the case of opening to all players, which is the main cost in protocols, one could apply the same technique. However, this is expensive as it relies on all players communicating their shares to all other players. This is very expensive, thus in [SW19, KRSW18] it is also shown that the method traditionally employed for replicated sharing for threshold $(n, t) = (3, 1)$ structures also generalizes to arbitrary $\mathcal{Q}_2$ access structures.

In this section we show that the methods of [SW19] which are proved in the context of MSPs over finite fields, also apply in our more general context of ESPs over the finite rings $\mathbb{Z}_{p^k}$. None of the results are deep, but are included here for completeness. The protocols are represented in Figure 5.8, but we explain them here at a high level here, and then go into more detail below (including the definition of various intermediate quantities).

Throughout our protocols, each player P_j maintains a running hash value $\mathcal{H}^j$ which is used to check consistency of openings, between the players. Each player should at any point hold the same value of $\mathcal{H}^j$. To ensure consistency we have a protocol HashCheck which ensures consistency of these values. Note, the communication in HashCheck can be done in the clear, and does not need to be protected against rushing adversaries, since each honest player will abort when it gets an incorrect hash value. This can either be an incorrect value sent maliciously by an adversarial player (in which case it should abort), or an incorrect value sent honestly by an honest player (which indicates something has gone wrong with the OpenToAll protocol).

Open to One:

To open a secret shared value $[x]_k$ to player P_i, an operation which we denote by $x \leftarrow \mathsf{OpenToOne}(i, [x]_k)$ in Figure 5.8. Each player sends their shares of $[x]_k$ to player P_i. Player P_i then verifies they are consistent using the parity check matrix for the ESP modulo p^k (see below). If they are not, he aborts, otherwise he stores the value of x. This gives an actively secure (with-abort) method to perform openings to a given player (see below for the proofs), since, if an adversary introduces an error in opening a value to an honest party, this is detected by the honest party.

Open to All:

To execute an opening to everyone, an operation which we denote by $x \leftarrow \mathsf{OpenToAll}([x]_k)$, one could execute $\mathsf{OpenToOne}(i, [x]_k)$ a total of n times. It is more efficient however to only send just enough data needed to perform the opening (as was done in [SW19, KRSW18]). For example in the case of replicated sharing with $(n,t) = (3,1)$, player P_i will send only the value $s_{(i+1) \pmod 3}$ to player $P_{(i+1) \pmod 3}$. To ensure correctness of the shares, each party uses the received share values to construct not only x, but also the sharing $\mathbf{x}$ of m values used to share x. They then update their hash function with $\mathcal{H}.\mathsf{Update}(\mathbf{x})$.

5.4.1 Open to One

In this section generalize the method of [SW19] from MSPs over fields to ESPs over finite rings, so as to show the method for OpenToOne in Figure 5.8 works. Note that to open a secret a party has to recombine the shares to reveal the underlying secret. Moreover, the opening party will have to be able to decide if the secret it is opening is a valid sharing. Given a multiplicative ESP, $\mathcal{M} = \left(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\right)$, and a valid encoding $\mathbf{s}$ of a secret s, augmented with a non-zero error $\mathbf{e}$, i.e. the player receives $\mathbf{c} = \mathbf{s} + \mathbf{e}$, it has to be possible for the opening party to detect that the secret has been corrupted. To achieve this we show that either $\mathbf{s} + \mathbf{e}$ is no longer a qualified vector or $\mathbf{e}$ encodes $\mathbf{0}$.

Lemma 5.4

Let $\mathcal{M} = \left(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\right)$ be an ESP computing a $\mathcal{Q}_2$ access structure Γ and $\mathbf{c} = \mathbf{s} + \mathbf{e}$ be the observed set of shares, given as a valid share vector $\mathbf{s}$ encoding s, with error $\mathbf{e}$. Then there exists a matrix N such that

$$\varphi(\mathsf{supp}(\mathbf{e})) \notin \Gamma \Rightarrow \begin{cases} \mathbf{e} \text{ encodes the error } e = 0 \\ N \cdot \mathbf{c} \neq \mathbf{0} \end{cases}$$

This generalizes [SW19][Lemma 2] and basically says that N can be viewed as a parity check matrix. The proof of this lemma rests on one main requirement, namely for M as given, $\ker(M^T)$ admits a basis. While this is generally true over fields this does not hold in general for modules over rings. To be able to show that $\ker M^T$ admits a basis we need to consider how M acts on $\mathbb{Z}_{p^k}$ as a module. Then the following proposition shows that $\ker(M^T)$ admits a basis.

Lemma 5.5

Let $\mathcal{M} = \left(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\right)$ be an ESP computing a $\mathcal{Q}_2$ access structure Γ, then $\ker(M^T)$ admits a basis.

Proof. Let $\mathcal{M}$ be as assumed, then $M \in M_{m\times d}(\mathbb{Z}_{p^k})$ is a full-rank matrix with $m \geq d$. Then M^T is also full rank and so $\dim(\operatorname{Im}(M^T)) = d = \operatorname{rank}(M)$, which means M^T is surjective. Hence, by the first isomorphism theorem:

$$\mathbb{Z}_{p^k}^m / \ker(M^T) \cong \operatorname{Im}(M^T)$$

$$\Rightarrow \ker(M^T) \cong \mathbb{Z}_{p^k}^m / \mathbb{Z}_{p^k}^d$$

Hence $\ker(M^T) \cong \mathbb{Z}_{p^k}^{m-d}$ and as $m \geq d$, $\ker(M^T)$ is a free module over $\mathbb{Z}_{p^k}$. From basic module theory it is known that a free module admits a basis, and so the lemma follows. $\square$

We also need to generalize [SW19][Lemma 1] to ESPs, which we do here

Lemma 5.6

For any ESP $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ computing a $\mathcal{Q}_2$ access structure Γ, for any vector $\mathbf{s} \in \mathbb{Z}_{p^k}$,

$$\varphi(\mathsf{supp}(\mathbf{s})) \notin \Gamma \Rightarrow \begin{cases} \mathbf{s} \notin \operatorname{Im}(M), \text{ or} \\ \mathbf{s} \in \operatorname{Im}(M), \text{ and } \mathbf{s} = M \cdot \mathbf{x} \text{ for some } x \in \mathbb{Z}_{p^k}^d \text{ where } \langle \mathbf{x}, \mathbf{e} \rangle = 0 \end{cases}$$

Proof. Consider the situation where $\varphi(\mathsf{supp}(\mathbf{s})) \notin \Gamma$. Then, by $\mathcal{Q}_2$-ness of the access structure, $\mathcal{P} \backslash \varphi(\mathsf{supp}(\mathbf{s})) \in \Gamma$ which means there exists a qualifying set $Q \subseteq \mathcal{P}$ that is contained in this set for which $\mathbf{s}^{(i)} = 0$ for all $i \in [m]$ for which $\varphi(i) \in Q$.

By Lemma 5.2 and 5.3 it holds that the recombination vectors exist, hence the qualified set Q can reconstruct the secret by computing $\langle \lambda, \mathbf{s} \rangle$ for the appropriate recombination vector λ, i.e. $\varphi(\mathsf{supp}(\lambda)) \subseteq Q$, however it is clear that, for this particular Q and λ, $\langle \lambda, \mathbf{s} \rangle = \langle \lambda_Q, \mathbf{s}_Q \rangle$, but $\mathbf{s}_Q = \mathbf{0}$ so $\langle \lambda_Q, \mathbf{s}_Q \rangle = 0$, so the secret is 0.

Now assume that $\mathbf{s} \in \mathrm{Im}(M)$, then for all $Q \in \Gamma$ is holds that $\langle \lambda_Q, \mathbf{s}_Q \rangle \equiv 0 \bmod p^k$, and so by definition of the ESP $\mathbf{s} = M \cdot \mathbf{x}$ and $\langle \mathbf{x}, \varepsilon \rangle = 0$. Else $\mathbf{s} \notin \mathrm{Im}(M)$, which proves the proposition. $\square$

We can now prove Lemma 5.4,

Proof. Let N be a matrix whose rows form a basis for $\ker(M^T)$, which exists by Lemma 5.6 and suppose $\mathbf{e} \in \mathbb{Z}_{p^k}^d$. Since by the Fundamental Lemma of Linear Algebra we have $\ker(M^T) = \mathrm{Im}(M)^\perp$, we also have $\mathbf{s} \in \mathrm{Im}(M)$ if and only if $N \cdot \mathbf{s} = 0$. By the predicate and Lemma 5.6 we have that either $\mathbf{e} \notin \mathrm{Im}(M)$ or $\mathbf{e} \in \mathrm{Im}(M)$ and $e = 0$. If the latter holds we are done. If $\mathbf{e} \notin \mathrm{Im}(M)$, then $N \cdot \mathbf{e} \neq 0$, and so $N \cdot \mathbf{c} = N \cdot (\mathbf{s} + \mathbf{e}) \neq 0$ as required. $\square$

5.4.2 Open to All

The OpenToAll procedure of Figure 5.8 also generalizes an idea set out in [SW19]. Each party, $P_i \in \mathcal{P}$, is assigned a set of shares it will receive for reconstruction. This is done via a map $\mathbf{q} : \mathcal{P} \to 2^{[m]}$, which is defined so that for each $P_i \in \mathcal{P}$, $\mathbf{q}(P_i)$ is a set $S_i \subseteq [m]$ under the conditions that

- $\ker(M_{S_i}) = \mathbf{0}$, i.e. the kernel of the submatrix of M with rows indexed by S_i is trivial.
- $\varphi^{-1}(P_i) \subseteq S_i$, i.e. each party includes all of their own shares in the set S_i.

Assume that such a function exists, then each P_i receives a set of shares, which we will denote $\mathbf{s}^i_{\mathbf{q}(P_i)}$ for a given secret s. Then, using this vector, P_i solves $\mathbf{s}^i_{\mathbf{q}(P_i)} = M_{S_i} \cdot \mathbf{x}^i_{P_i}$ for $\mathbf{x}^i_{P_i}$. This $\mathbf{x}^i_{P_i}$ is then used to completely reconstruct the share vector $\mathbf{s}^i = M \cdot \mathbf{x}^i$. Note that in [SW19] it is shown that such a function $\mathbf{q}$ exists and that there are semi-efficient ways to compute them for MSPs over fields. Now let $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \epsilon, \phi)$ be an ESP whose reduction modulo p is the MSP $\mathcal{M}_p = (\mathbb{F}_p, M_p, \epsilon_p, \phi)$, recall that both programs compute the access structure Γ. We first note that the same function $\mathbf{q}$ used for $\mathcal{M}_p$ can be used for $\mathcal{M}$, this means we can use the same $\mathbf{q}$ for both $\mathcal{M}_p$ and $\mathcal{M}$ that also means that an efficient $\mathbf{q}$ can be computed for $\mathcal{M}$.

Lemma 5.7

Let the MSP $\mathcal{M}_p$ be the reduction modulo p of the ESP $\mathcal{M}$ over $\mathbb{Z}_{p^k}$. Assume $\mathcal{M}_p$ is equipped with a function $\mathbf{q}$ as described above fulfilling the preconditions. Then $\mathbf{q}$ can be extended naturally to a function that fulfils the same preconditions for $\mathcal{M}$.

Proof. Assume that $M \equiv M_p \mod p$ and assume the given $\mathbf{q}$ exists for $\mathcal{M}_p$. Then, as $\mathcal{M}$ and $\mathcal{M}_p$ compute the same access structure, $\phi^{-1}(P_i) \subseteq S_i$ must also hold if we consider the natural extension of $\mathbf{q}$ to $\mathcal{M}$. By assumption $\ker(M_{p_{S_i}}) = \mathbf{0}$, and as $M \equiv M_p \mod p$ this means that if $\ker(M_{S_i}) \neq \mathbf{0}$ then

$$\mathbf{0} \subset \ker(M_{S_i}) \subseteq \{\mathbf{v} \in \mathbb{Z}_{p^k}^d \mid \mathbf{v}^{(i)} \in \{0, p, \ldots, p^{k-1}\}\}.$$

Hence, assume that this is the case and denote by $\mathbf{m}_i$ the ith row vector of M, such that

$$(M \cdot \mathbf{v})^{(i)} = \langle \mathbf{m}_i, \mathbf{v} \rangle = \sum_{j=1}^{m} \mathbf{m}_i^{(j)} \cdot \mathbf{v}^{(j)}.$$

It is clear that if this is to be $\mathbf{0} \mod p^k$, as required, then there exists a subset $Z \subseteq [d]$ such that $p \mid \sum_{z \in Z} m_z$ as $m_{i,j} \in [p]$ and $v_i \in \{0, p, \ldots, p^{k-1}\}$. But then *modulo* p there would exist a solution $\mathbf{v}_p$ such that $M \cdot \mathbf{v} = 0$ by setting $v_i = 1$ if $i \in Z$ and 0 otherwise, hence contradicting that $\ker(M_{p_{S_i}}) = \mathbf{0}$. So $\ker(M_{S_i}) = \mathbf{0}$ and therefore there exists a natural extension to $\mathbf{q}$ to $\mathbb{Z}_{p^k}$. □

Now consider the equation $\mathbf{s}^i = M \cdot vx^i$ and especially whether $\mathbf{x}^i$ exists. As $\ker M_{S_i} = \mathbf{0}$ we know that if $\mathbf{x}^i$ does not exist then the adversary must have introduced errors, because $\mathbf{s}^i_{\mathbf{q}(P_i)}$ is a subvector of some share vector $\mathbf{s}$. If this is the case the protocol in Figure 5.8 instructs P_i to send an abort message to all players. If such an $\mathbf{x}^i$ does exist the adversary may still have introduced errors. However, the hash values will differ between the players and so will cause an abort, when HashCheck is evaluated. This is formally described in the following lemma.

Lemma 5.8

Let $\mathbf{q} : \mathcal{P} \rightarrow 2^{[m]}$ be defined with the conditions given above (for a $\mathcal{Q}_2$ ESP) and let $\mathbf{s}^i_{\mathbf{q}(P_i)}$ denote the subvector of shares received by P_i for a given secret s. Suppose all parties $P_i \in \mathcal{P}$ are able to obtain a solution $\mathbf{x}^i$ to the equation $\mathbf{s}^i_{\mathbf{q}(P_i)} = M_{\mathbf{q}(P_i)} \cdot \mathbf{x}^i$, hence can compute $\mathbf{s}^i = M \cdot \mathbf{x}^i$. Then the adversary did not introduce any errors if the reconstructed values $\mathbf{s}^i$ and $\mathbf{s}^j$ are equal for all players P_i and P_j.

Proof. First note that the existence of $\mathbf{q}$ is not in question, as [*SW*19] shows that $\mathbf{q}$ exists modulo p and Lemma 5.7 has shown that the same $\mathbf{q}$ can be used.

As $\ker(M_{\mathbf{q}(P_i)}) = \mathbf{0}$, we see that that the map defined by $M_{\mathbf{q}(P_i)}$ is injective, hence there exists a unique $\mathbf{x}^i$ for every $P_i \in \mathcal{P}$ that is a solution to the equation

$$\mathbf{s}^i_{\mathbf{q}(P_i)} = M_{\mathbf{q}(P_i)} \cdot \mathbf{x}^i.$$

Each $\mathbf{x}^i$ will result in a unique vector $\mathbf{s}^i$, which will be the same for all parties if the players reconstruct the same vector $\mathbf{x}^i$. Recall that, by the $\mathcal{Q}_2$ assumption, the set of honest players is a qualified set. Thus, if all honest players agree on their values of $\mathbf{s}^i$, which they check via the hash checking, then they know that all the values they received from any dishonest parties are consistent with the valid sharing. $\square$

Protocol Π_{Opening}

For each $P_i \in \mathcal{P}$, the parties choose on some recombination vector λ^i such that $\mathsf{supp}(\lambda^i) \subseteq \mathbf{q}(P_i)$. Denote by $\mathcal{H}^i$ the hash function locally updated by player P_i, which has been pre-initialized with $\mathcal{H}^i.\mathsf{Init}()$. If at any point P_i receives the abort command it runs the subprotocol **Abort**.

$\mathsf{OpenToAll}([x]_k)$: Each $P_j \in \mathcal{P}$ executes:

1. Retrieve from memory the recombination vector λ^j.
2. For each $P_t \in \mathcal{P}$, for each $r \in \mathbf{q}(P_t)$, if $\varphi(r) = P_j$ then send $\mathbf{s}_r$ to P_t, see Section 5.4.2 for the definition of $\mathbf{q}(P_t)$.
3. For each $r \in \mathbf{q}(P_j)$, wait to receive $\mathbf{s}_r$ from player $\varphi(r)$.
4. Concatenate local and received shares into $\mathbf{s}^j_{\mathbf{q}(P_j)} \in \mathbb{Z}_{p^k}^{|\mathbf{q}(P_j)|}$.
5. Locally compute $s = \langle \lambda^j_{\mathbf{q}(P_j)}, \mathbf{s}^j_{\mathbf{q}(P_j)} \rangle$.
6. Solve $M_{\mathbf{q}(P_j)} \cdot \mathbf{x}^j = \mathbf{s}^j_{\mathbf{q}(P_j)}$ for $\mathbf{x}^j$. If there is no solution, run **Abort**.
7. Execute $\mathcal{H}.\mathsf{Update}^j(M \cdot \mathbf{x}^j)$.

$\mathsf{OpenToOne}(i, [x]_k)$: The secret has to be opened to P_i alone and so the parties P_j do the following:

1. Each $P_j \in \mathcal{P} \backslash \{P_i\}$ sends $\mathbf{s}_{\{P_j\}}$ to P_i, who concatenates local and received shares into a vector $\mathbf{s}$.
2. Party P_i computes $N \cdot \mathbf{s}$ as discussed in section 5.4.1. If $N\mathbf{s} = \mathbf{0}$, P_i outputs $\langle \lambda^i, \mathbf{s} \rangle = s$, and otherwise it runs **Abort**.

$\mathsf{HashCheck}()$: Each $P_i \in \mathcal{P}$ does the following:

1. Compute $h^i := \mathcal{H}^i.\mathsf{Out}()$.
2. Send h^i to all other parties P_j, for $i \neq j$ (this can be done in the clear).
3. Wait for h^j from all parties P_j, for $i \neq j$.
4. If $h^j \neq h^i$ for any j, run **Abort**

Abort : If a party calls this subroutine, it sends abort to all parties and aborts. If a party receives the message abort, it aborts.

Figure 5.8: Protocol Π_{Opening}

5.5 Multiplication Check

We present various protocols which allow one to verify that a set of passively secure multiplications are indeed correct. In the context of generating triples, we note that, we are unable to "lift" a valid triple modulo p^k to a valid triple modulo p^{k+v}. Thus, if one needs to perform a check modulo p^{k+s}, one needs to generate the passively secure multiplication triples modulo the larger modulus first, even if one is only interested in computation modulo p^k.

We assume that the desired security level is 2^κ, i.e. the probability that an adversary can pass off an incorrect passively secure multiplication as correct should be $2^{-\kappa}$. To ensure this we define four (integer) parameters (u, v, w, B) for our protocols defined by, where $B_z = 0$ unless $B \neq 1$ in which case we set $B_z = 1$.

$$u = \lceil (\kappa + B_z)/\log_2 p \rceil$$

$$v = u - 1,$$

$$1 \leq B \leq 1 + (p^w - 1)/2^{\kappa + B_z}.$$

The value u defines the size of the challenge space in our protocols, the value v defines how much bigger a modulus we need to work with, the value w defines the degree of any extension needed to allow the Schwartz-Zippel Lemma 5.1 to apply, using a set S of size $p^w - 1$, whilst B defines the bucket size of the check (equivalently the degree of the polynomial used in the Schwartz-Zippel Lemma).

Our methods here are a natural generalization of the methods given in [EKO+20, ADEN19] which are themselves based on ideas used in [CDE+18]. We note for the case of $k = 1$ and a small prime p the following protocols produce more efficient "sacrificing" steps than the "traditional" method of repeating the protocol $\kappa/\log_2 p$ times.

5.5.1 MultCheck$_1$

The first protocol, often called sacrifice, takes a set of N passively secure multiplication triples $([x_i]_{k+v}, [y_i]_{k+v}, [z_i]_{k+v})$, and checks whether indeed $z_i = x_i \cdot y_i \pmod{p^k}$, using another set of passively secure multiplication triples $([a_i]_{k+v}, [b_i]_{k+v}, [c_i]_{k+v})$. The "unchecked" triples $([a_i]_{k+v}, [b_i]_{k+v}, [c_i]_{k+v})$ need to be discarded at the end of the protocol (thus the term sacrificing). The output of the protocol is either an abort signal, or a set of N "actively" secure triple $([x_i]_k, [y_i]_k, [z_i]_k)$. The protocol is described in Figure 5.9 and is based internally on the Beaver multiplication protocol. For ease of exposition we assume B exactly divides N in the protocol, this can easily be removed.

The Protocol $\mathsf{MultCheck}_1$

```
Input: $([x_i]_{k+v}, [y_i]_{k+v}, [z_i]_{k+v})_{i=0}^{N-1}$ and $([a_i]_{k+v}, [b_i]_{k+v}, [c_i]_{k+v})_{i=0}^{N-1}$.
Output: abort or $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$.

  1. Let $R$ denote a degree $w$ Galois ring over $\mathbb{Z}_{p^{k+v}}$.
  2. Let $S$ denote the set from the Schwartz-Zippel Lemma of size $p^w - 1$.
  3. $t \leftarrow \mathcal{F}_{\mathrm{AgreeRandom}}(\mathbb{Z}_{p^u})$.
  4. For $j \in [0, \ldots, N/B)$ do
     (a) $r \leftarrow \mathcal{F}_{\mathrm{AgreeRandom}}(S)$.
     (b) For $i \in [0, \ldots, B)$ do
         i. $[\rho_i]_{k+v} \leftarrow t \cdot [a_{j \cdot B+i}]_{k+v} - [x_{j \cdot B+i}]_{k+v}$.
         ii. $[\sigma_i]_{k+v} \leftarrow [b_{j \cdot B+i}]_{k+v} - [y_{j \cdot B+i}]_{k+v}$.
     (c) $(\rho_i)_{i=0}^{B-1} \leftarrow (\mathsf{OpenToAll}([\rho_i]_{k+v}))_{i=0}^{B-1}$.
     (d) $(\sigma_i)_{i=0}^{B-1} \leftarrow (\mathsf{OpenToAll}([\sigma_i]_{k+v}))_{i=0}^{B-1}$
     (e) $[\tau]_{k+v} \leftarrow 0$.
     (f) For $i \in [0, \ldots, B)$ do
         i. $[d_i]_{k+v} \leftarrow t \cdot [c_{j \cdot B+i}]_{k+v} - [z_{j \cdot B+i}]_{k+v} - \sigma_i \cdot [x_{j \cdot B+i}]_{k+v} - \rho_i \cdot [y_{j \cdot B+i}]_{k+v} - \sigma_i \cdot \rho_i$.
         ii. $[\tau]_{k+v} \leftarrow [\tau]_{k+v} + r^i \cdot [d_i]_{k+v}$.
     (g) $\tau \leftarrow \mathsf{OpenToAll}([\tau]_{k+v})$
     (h) If $\tau \neq 0 \pmod{p^{k+v}}$ output abort and stop.
  5. For $i \in [0, \ldots, N)$ do
     (a) $[x_i]_k \leftarrow [x_i]_{k+v} \pmod{p^k}$, $[y_i]_k \leftarrow [y_i]_{k+v} \pmod{p^k}$, $[z_i]_k \leftarrow [z_i]_{k+v} \pmod{p^k}$.
  6. Output $([x]_k, [y]_k, [z]_k)_{i=1}^{N}$.
```

Figure 5.9: The Protocol $\mathsf{MultCheck}_1$

The number of calls to the procedure $\mathsf{OpenToAll}(\cdot)$, which is the main cost of the protocol is given by $2 \cdot N + N \cdot w/B$, and the number of rounds of communication (for the OpenToAll calls) is bounded by two (if one executes the main j-loop in parallel). This means the communication cost, per output triple, is equal to the communication of $2 + w/B$ executions of OpenToAll$(\cdot)$. In practice one would try to select w/B to be as small as possible. In such a situation we can treat the cost as two calls to OpenToAll$(\cdot)$.

In the case of $k = 1$ and a large prime p, the values $w = 1$, $u = 1$, $v = 0$ and $B = 1$ give rise to *exactly* the traditional sacrifice protocol from SPDZ. However, for such large p, we could choose $w = 2$ and allow B to be sufficiently big, without needing an overly large amount of triples to check at once. Thus, by utilizing our modified protocol one can achieve an improvement on the classical SPDZ sacrificing protocol. So for large p, for the classical SPDZ sacrifice, we have $w/B = 1$ and hence the cost is three calls to OpenToAll$(\cdot)$, but for our protocol we can achieve two calls to OpenToAll$(\cdot)$.

As long as we perform the calls to AgreeRandom only *after* the adversary had a chance to influence the triples, and the adversary is fully committed to any errors introduced in them, we can use the same random values for t and r over all instantiations. The practical advantage of this is that the data cost of these calls can then be amortized over all these executions, and we can consider it negligible. Due to the commit-reveal nature of the AgreeRandom sub-protocol, however, we still need to take a cost of two rounds of communication into account. All invocations of AgreeRandom that we need to generate the required t and r values can be executed in parallel, so the number of rounds we need does not grow as the number of times MultCheck$_1$ is executed grows.

We now prove the following theorem which is an adaption of similar results in [CDE+18] (especially Claim 6 in that paper) and the papers [EKO+20, ADEN19], but we have generalized the method to arbitrary p and also the case of potentially small k.

Lemma 5.9

In the presence of an active adversary, who can introduce arbitrary additive errors into the input triples, the protocol MultCheck$_1$ will output an invalid multiplication triple with probability $(B-1)/(p^w - 1) + p^{-u} \leq 2^{-\kappa}$.

Proof. We first note that since the OpenToAll sub-protocol ensures the opened shares are indeed consistent, the only error that can be introduced by the adversary is an error in the c_i or z_i. Without loss of generality we can assume these are additive errors known to the adversary, thus we have $c_i = a_i \cdot b_i + e_{c,i}$ and $z_i = x_i \cdot y_i + e_{z,i}$ for some $e_{c,i}, e_{z,i} \in \mathbb{Z}_{p^{k+v}}$.

The value τ represents a polynomial of degree $B-1$ over $\mathbb{Z}_{p^{k+v}}$ evaluated at a random point $r \in S$. Thus, by the Schwartz-Zippel Lemma 5.1 the probability that $\tau = 0 \pmod{p^{k+v}}$ when the polynomial is not identically zero is bounded by $(B-1)/(p^w-1)$. Thus, we can conclude that each coefficient is identically equal to zero, i.e. the errors satisfy for each i.

$$t \cdot e_{c,i} + e_{z,i} = 0 \pmod{p^{k+v}}.$$

Note, we are finally only interested in errors for which $e_{c,i} \neq 0 \pmod{p^k}$, as we only are going to output a sharing modulo p^k. So we write $p^{s_i} = \gcd(e_{c,i}, p^{k+v})$, and since $e_{c,i} \neq 0 \pmod{p^k}$ we have $s_i + 1 \leq k$.

We can write $e_{c,i} = p^{s_i} \cdot f_i$ and $e_{z,i} = p^{s_i} \cdot g_i$ for some $f_i, g_i \in \mathbb{Z}_{p^{k+v}}$. For the equation to pass we must then have that

$$t \cdot f_i + g_i = 0 \pmod{p^{k+v-s_i}}.$$

In particular this means that $t \equiv -g_i/f_i \pmod{p^{k+v-s_i}}$, as $\gcd(f_i, p) = 1$. In particular the value t which will make the protocol verify is determined (modulo p^{k+v-s_i}) completely by the error introduced by the adversary; and in effect it must be the same error introduced on each invalid pair of triples.

Note, that the adversary needed to commit to the values $e_{c,i}$ and $e_{z,i}$ before they see the t. Thus, t is pre-determined from a set of size $p^{k+v-s_i} \geq p^{s_i+1+v-s_i} = p^{v+1} = p^u$, since $k \geq s_i+1$. Therefore, the probability of an adversary passing off a set of invalid tuples as valid, when $\tau = 0$, is bounded by $p^{-u} < 2^{-(\kappa+B_z)}$. □

5.5.2 $\mathsf{MultCheck}'_1$

We will also use the $\mathsf{MultCheck}_1$ protocol in the case where we are already guaranteed that the auxiliary triples $([a_i]_k, [b_i]_k, [c_i]_k)_{i=0}^{N-1}$ are correct, and we have $v = 0$ and $u = k$, and we are simply checking whether the passively secure triples $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$ are correct. We refer to this special case as $\mathsf{MultCheck}'_1$ and it is presented in Figure 5.10. The round complexity is the same as that of $\mathsf{MultCheck}_1$, except for the output, although now we can operate modulo p^k only, without needing to extend to working modulo p^{k+s}. In this special case we obtain the following result,

The Protocol $\mathsf{MultCheck}'_1$

```
Input: $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$ and $([a_i]_k, [b_i]_k, [c_i]_k)_{i=0}^{N-1}$.
Output: abort or OK.

  1. Let $R$ denote a degree $w$ Galois ring over $\mathbb{Z}_{p^k}$.
  2. Let $S$ denote the set from the Schwartz-Zippel Lemma.
  3. For $j \in [0, \ldots, N/B)$ do
     (a) $r \leftarrow \mathcal{F}_{\text{AgreeRandom}}(S)$.
     (b) For $i \in [0, \ldots, B)$ do
         i. $[\rho_i]_k \leftarrow [a_{j \cdot B+i}]_k - [x_{j \cdot B+i}]_k$.
         ii. $[\sigma_i]_k \leftarrow [b_{j \cdot B+i}]_k - [y_{j \cdot B+i}]_k$.
     (c) $(\rho_i)_{i=0}^{B-1} \leftarrow (\mathsf{OpenToAll}([\rho_i]_k))_{i=0}^{B-1}$.
     (d) $(\sigma_i)_{i=0}^{B-1} \leftarrow (\mathsf{OpenToAll}([\sigma_i]_k))_{i=0}^{B-1}$
     (e) $[\tau]_k \leftarrow 0$.
     (f) For $i \in [0, \ldots, B)$ do
         i. $[d_i]_k \leftarrow [c_{j \cdot B+i}]_k - [z_{j \cdot B+i}]_k - \sigma_i \cdot [x_{j \cdot B+i}]_k - \rho_i \cdot [y_{j \cdot B+i}]_k - \sigma_i \cdot \rho_i$.
         ii. $[\tau]_k \leftarrow [\tau]_k + r^i \cdot [d_i]_k$.
     (g) $\tau \leftarrow \mathsf{OpenToAll}([\tau]_k)$
     (h) If $\tau \neq 0 \pmod{p^k}$ output abort and stop.
  4. Output OK.
```

Figure 5.10: The Protocol $\mathsf{MultCheck}'_1$

Lemma 5.10

In the presence of an active adversary, who can introduce arbitrary additive errors into the input triples $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$, but not the input triples $([a_i]_k, [b_i]_k, [c_i]_k)_{i=0}^{N-1}$, the protocol $\mathsf{MultCheck}'_1$ will output $\mathtt{OK}$ incorrectly with probability $(B-1)/(p^w-1) \leq 2^{-(\kappa+B_z)}$.

Proof. We first note that since the $\mathsf{OpenToAll}$ sub-protocol ensures the opened shares are indeed consistent the only error that can be introduced by the adversary is an error in the z_i, there can be no error in the c_i by assumption. Without loss of generality we can assume these are additive errors known to the adversary, thus we have $c_i = a_i \cdot b_i$ and $z_i = x_i \cdot y_i + e_{z,i}$ for some $e_{z,i} \in \mathbb{Z}_{p^k}$. The application of the Schwartz-Zippel lemma allows us to conclude, except with probability bounded by $(B-1)/(p^w-1)$, that we have, for each i, that $e_{z,i} = 0 \pmod{p^k}$. Hence, except with probability bounded by $(B-1)/(p^w-1)$, there can be no errors in the triples $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$. □

5.5.3 MultCheck$_2$

Our third protocol comes from a combination of ideas from [CDE+18] and [KOS16]. Instead of consuming previously produced multiplication triples (which themselves require a passively secure multiplication to produce) this second variant makes direct use of a passively secure multiplication protocol PassMult; which can be any of MaurerMult, KRSWMult or DNMult. The protocol, called MultCheck$_2$, is described in Figure 5.11. The argument for security is roughly the same as that for protocol MultCheck$_1$.

5.5.4 MacCheck

Our final protocol is the generalization of the MacCheck protocol from [DPSZ12] to our situation. The protocol checks, for an input of a single secret shared value $[\alpha]_{k+v}$ and a series of pairs of secret shared values $([x_i]_{k+v}, [y_i]_{k+v})_{i=0}^{N-1}$, whether we have $y_i = \alpha \cdot x_i \pmod{p^{k+v}}$, or whether y_i is invalid up to an additive error. Note, unlike the MacCheck protocol from [DPSZ12] we are not checking the MACs of opened values, but checking the consistency of pairs of unopened values with respect to the shared MAC key α, as such it is closer to the verification stage of the protocol in [CGH+18]. We note that with the instantiation given in Figure 5.12, this checking procedure "burns" the value $[\alpha]_{k+v}$, thus this does not allow for reactive computations. In [CGH+18] it is shown how to avoid this problem for *specific* secret sharing schemes. The protocol is given in Figure 5.12

Lemma 5.11

Protocol MacCheck in Figure 5.12 on input of an invalid set of pairs $([x_i]_{k+v}, [y_i]_{k+v})_{i=0}^{N-1}$ will return OK with probability less than $2^{-\kappa}$. Where a pair being invalid means that $y_i = \alpha \cdot x_i + e_i$, for an e_i known to the adversary with $e_i \neq 0 \pmod{p^k}$.

```
The Protocol MultCheck_2

Input: $([x_i]_{k+v}, [y_i]_{k+v}, [z_i]_{k+v})_{i=0}^{N-1}$.
Output: abort or $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$.

  1. Let $R$ denote a degree $w$ Galois ring over $\mathbb{Z}_{p^{k+v}}$.
  2. Let $S$ denote the set from the Schwartz-Zippel Lemma.
  3. For $i \in [0, \dots, N)$ do
     (a) $[a_i]_{k+v} \leftarrow \mathcal{F}_{\text{PRSS}}(k+v)$.
     (b) $[c_i]_{k+v} \leftarrow \mathsf{PassMult}([a_i]_{k+v}, [y_i]_{k+v})$.
  4. $t \leftarrow \mathcal{F}_{\text{AgreeRandom}}(\mathbb{Z}_{p^u})$.
  5. For $j \in [0, \dots, N/B)$ do
     (a) $r \leftarrow \mathcal{F}_{\text{AgreeRandom}}(S)$.
     (b) For $i \in [0, \dots, B)$ do
         i. $[\rho_i]_{k+s} \leftarrow t \cdot [x_{j \cdot B+i}]_{k+v} + [a_{j \cdot B+i}]_{k+v}$.
     (c) $(\rho_i)_{i=0}^{B-1} \leftarrow (\mathsf{OpenToAll}([\rho_i]_{k+v}))_{i=0}^{B-1}$.
     (d) $[\tau]_{k+v} \leftarrow 0$.
     (e) For $i \in [0, \dots, B)$ do
         i. $[\tau]_{k+v} \leftarrow [\tau]_{k+v} + r^i \cdot (t \cdot [z]_{k+v} + [c]_{k+v} - \rho \cdot [y]_{k+v})$.
     (f) $\tau \leftarrow \mathsf{OpenToAll}([\tau]_{k+s})$
     (g) If $\tau \neq 0 \pmod{p^{k+v}}$ output abort and stop.
  6. For $i \in [0, \dots, N)$ do
     (a) $[x_i]_k \leftarrow [x_i]_{k+v} \pmod{p^k}$, $[y_i]_k \leftarrow [y_i]_{k+v} \pmod{p^k}$, $[z_i]_k \leftarrow [z_i]_{k+v} \pmod{p^k}$.
  7. Output $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$.
```

Figure 5.11: The Protocol $\mathsf{MultCheck}_2$

Proof. Given the additive errors e_i we can define a global additive error e on the pair (u, v) with

$$t = v - \alpha \cdot u = \sum_{i=0}^{N-1} r_i \cdot y_i - r_i \cdot \alpha \cdot x_i$$

The Protocol MacCheck

Input: $[\alpha]_{k+v}$ and $([x_i]_{k+v}, [y_i]_{k+v})_{i=0}^{N-1}$.
Output: abort or OK.

```
1. For $i \in [0, N)$ do $r_i \leftarrow \mathcal{F}_{\text{AgreeRandom}}(\mathbb{Z}_{p^u})$.
2. $[u]_{\leftarrow} \sum_{i=0}^{N-1} r_i \cdot [x_i]_{k+v}$.
3. $[v]_{\leftarrow} \sum_{i=0}^{N-1} r_i \cdot [y_i]_{k+v}$.
4. $[c]_{k+v} \leftarrow \mathcal{F}_{\text{PRSS}}(k+v)$.
5. $\alpha \leftarrow \mathsf{OpenToAll}([\alpha]_{k+v})$.
6. $[t]_{k+v} \leftarrow [v]_{k+v} - \alpha \cdot [u]_{k+v}$.
7. $[s]_{k+v} \leftarrow \mathsf{PassMult}([t]_{k+v}, [c]_{k+v})$.
8. $s \leftarrow \mathsf{OpenToAll}([s]_{k+v})$.
9. If $s = 0$ then return OK, else return abort.
```

Figure 5.12: The Protocol MacCheck

$$= \sum_{i=0}^{N-1} r_i \cdot (y_i - \alpha \cdot x_i)$$

$$= \sum_{i=0}^{N-1} r_i \cdot e_i = e.$$

The passively secure multiplication can itself introduce an additive error d on s, i.e. $a = c \cdot t + d = c \cdot e + d$. Thus, the test will output OK incorrectly when $a = 0$, but there is an $e_i \neq 0 \pmod{p^k}$.

We are only interested in errors for which $e_i \neq 0 \pmod{p^k}$, i.e. $e \neq 0 \pmod{p^k}$. So we write $p^s = \gcd(e, p^{k+v})$, and since $e \neq 0 \pmod{p^k}$ we have $s + 1 \leq k$. We thus write $e = p^s \cdot f$ and $d = p^s \cdot g$ for some $f, g \in \mathbb{Z}_{p^{k+v}}$. For the equation to pass we then require that

$$c \cdot f + d = 0 \pmod{p^{k+v-s}}.$$

This in particular means that $c = -g/f \pmod{p^{k+v-s}}$, which means that c is completely determined by the errors e_i introduced by the adversary and the random values r_i. But these values must be committed to by the adversary before the value c is obtained. But c is chosen from a set of size $p^{k+v-s} \geq p^{v+1} > 2^{\kappa}$, and thus the probability that c will pass the test incorrectly is bounded by $2^{-\kappa}$. $\square$

5.5.5 Summary

We summarize the costs of various protocols in Table 5.2 for a general ESP over $\mathbb{Z}_{p^k}$. These are given in terms of the row m and column d dimensions of the matrix generating the underlying ESP, the number of parties n, and the parameters w and B used in the protocols above. We let $|\mathbf{s}_i|$ denote the share size of player P_i for the given ESP. The data column indicates the total amount of data sent for *all* players[7] as a multiple of the underlying secret shared data size (i.e. either $k \cdot \log_2 p$ or $(k+v) \cdot \log_2 p$); we ignore rounds/data to check the running hash values H as these are amortized over many sub-protocol executions. A $\star$ in the table indicates that the value depends highly on the specific ESP, and thus a formula is hard to present. The cost $\star_1$ of $\mathsf{OpenToAll}$ is generally $n \cdot d - m$ for an MSP with no redundancy, but it can be larger than this if the MSP has more redundancy than necessary.

We present three lines corresponding to $\mathsf{MultCheck}_2$ and $\mathsf{MacCheck}$ depending on whether the underlying passively secure multiplication is Maurer, KRSW or DN based. We assume $\mathcal{F}_{\text{PRSS}}$ is executed non-interactively in all cases, that any calls to $\mathcal{F}_{\text{AgreeRandom}}$ are amortized across many calls to $\mathsf{MultCheck}_i$, and that no king-paradigm is used in order to keep the number of rounds to a minimum. As mentioned in the discussion on the multiplication checks, we always consider w/B to be negligibly small.

To provide more concrete values we also give, in Table 5.3, the values for the three different instantiations of threshold sharings for $(n,t) \in \{(3,1),(5,2),(10,4)\}$. In the table we assume the parameters for our checking procedures are selected so that the term w/B can be ignored. The three different sharings have been selected as replicated (for general p^k), standard Shamir (for the case of $p > n$) and Shamir obtained via Galois rings (for the important case of $p = 2$). More specifically our three examples are; see Appendix 5.B for details

1. Threshold replicated sharing for threshold t. This has values $m = (n - t) \cdot {}^{n}C_t$ and $d = {}^{n}C_t$, where ${}^{n}C_t$ denotes the number of combinations of t objects selected from a pool of n. Each player holds $|\mathbf{s}_i| = m/n$ shares. The data cost of $\mathsf{OpenToAll}$ per player is $d - |\mathbf{s}_i|$, and thus the total cost,

[7]i.e. not the per-player amount

Protocol	General MSP			
	Rounds	Data	PRSS/PRZS	Triples
Share	1	$m - \lvert\mathbf{s}_i\rvert$	0	0
OpenToOne	1	$m - \lvert\mathbf{s}_i\rvert$	0	0
OpenToAll	1	$\star_1$	0	0
BeaverMult	1	$2 \cdot \star_1$	0	1
MaurerMult	1	$(n-1) \cdot m$	0	0
KRSWMult	1	$\star_2$	$\star_3$	0
DNMult	1	$n \cdot (n-1)$	2	0
MacCheck^M	4	$(n-1) \cdot m + 2 \cdot \star_1$	1	0
MacCheck^K	4	$\star_2 + 2 \cdot \star_1$	$1 + \star_3$	0
MacCheck^D	4	$n \cdot (n-1) + 2 \cdot \star_1$	3	0
MultCheck$_1$	4	$(2 + w/B) \cdot \star_1$	0	0
MultCheck$_2^M$	5	$(n-1) \cdot m + (1 + w/B) \cdot \star_1$	1	0
MultCheck$_2^K$	5	$\star_2 + (1 + w/B) \cdot \star_1$	$1 + \star_3$	0
MultCheck$_2^D$	5	$n \cdot (n-1) + (1 + w/B) \cdot \star_1$	3	0

Table 5.2: Costs of the Base Protocols for a General Access Structures

$\star_1$, over all players $n \cdot d - m$. Table 1 in [SW19] gives the cost $\star_2$ of KRSWMult as $n \cdot (n - t - 1)$.

2. Shamir sharing for threshold t when p is large. Here we have $m = n$ and $d = t + 1$, with each player holds $|\mathbf{s}_i| = 1$ share. The data cost of OpenToAll per player is again $d - |\mathbf{s}_i|$, and thus the total cost, $\star_1$, over all players $n \cdot (t + 1) - n = n \cdot t$. Table 1 in [SW19] gives the cost $\star_2$ of KRSWMult again as $n \cdot (n - t - 1)$.

3. Shamir sharing for threshold t when p is two. Here we need to define a degree d_n such that $n \leq 2^{d_n} - 1$, so we select $d_3 = 2$, $d_5 = 3$ and $d_{10} = 4$. We have $m = n \cdot d_n$ and $d = d_n \cdot t + 1$, and each player holds $|\mathbf{s}_i| = d_n$ elements in their sharing. Thus, the data cost of OpenToAll per player is again $d - |\mathbf{s}_i|$, and so the total cost, $\star_1$, over all players is $n \cdot d - m$. The cost of KRSWMult in this case depends on many factors and cannot be easily expressed in a closed formula. Therefore, we present the concrete values for our examples in Table 5.3[8]. The derivation of these costs can be found in Appendix 5.B.

In most cases we see that KRSW-based multiplication is the more efficient choice (in terms of bandwidth consumed as opposed to computational resources). Only

[8]In this table for threshold $(10, 4)$ we give the KRSWMult cost for $k = 128$.

for Shamir sharing over $\mathbb{Z}_{2^k}$, do we see that DNMult outperforms KRSWMult due to it having no dependency on the size of the ESP, as it's communication cost only depends (quadratically) on the number of players. Thus, we will assume the most efficient choice of passive multiplication is used for a given ESP for the rest of our analysis in this chapter. An interesting observation is that the KRSWMult cost of replicated sharing for a given access structure always is more efficient than the same cost using a dedicated Shamir-based sharing; although of course the other costs are more expensive when using replicated.

	Replicated (3, 1)				Replicated (5, 2)				Replicated (10, 4)			
Protocol	Rounds	Data	PRSS/PRZS	Triples	Rounds	Data	PRSS/PRZS	Triples	Rounds	Data	PRSS/PRZS	Triples
Share	1	4	0	0	1	24	0	0	1	1134	0	0
OpenToOne	1	4	0	0	1	24	0	0	1	1134	0	0
OpenToAll	1	3	0	0	1	20	0	0	1	840	0	0
BeaverMult	1	6	0	1	1	40	0	1	1	1680	0	1
MaurerMult	1	12	0	0	1	120	0	0	1	11340	0	0
KRSWMult	1	3	1	0	1	10	1	0	1	50	1	0
DNMult	1	6	2	0	1	20	2	0	1	90	2	0
MacCheck^M	4	18	1	0	4	160	1	0	4	13020	1	0
MacCheck^K	4	9	2	0	4	50	2	0	4	1730	2	0
MacCheck^D	4	12	3	0	4	60	3	0	4	1770	3	0
MultCheck$_1$	4	6	0	0	4	40	0	0	4	1680	0	0
MultCheck$_2^M$	5	15	1	0	5	140	1	0	5	12180	1	0
MultCheck$_2^K$	5	6	2	0	5	30	2	0	5	890	2	0
MultCheck$_2^D$	5	9	3	0	5	40	3	0	5	930	3	0

	Shamir (3, 1) for large prime				Shamir (5, 2) for large prime				Shamir (10, 4) for large prime			
Protocol	Rounds	Data	PRSS/PRZS	Triples	Rounds	Data	PRSS/PRZS	Triples	Rounds	Data	PRSS/PRZS	Triples
Share	1	2	0	0	1	4	0	0	1	9	0	0
OpenToOne	1	2	0	0	1	4	0	0	1	9	0	0
OpenToAll	1	3	0	0	1	10	0	0	1	40	0	0
BeaverMult	1	6	0	1	1	20	0	1	1	80	0	1
MaurerMult	1	6	0	0	1	20	0	0	1	90	0	0
KRSWMult	1	3	2	0	1	10	3	0	1	50	6	0
DNMult	1	6	2	0	1	20	2	0	1	90	2	0
MacCheck^M	4	12	1	0	4	40	1	0	4	170	1	0
MacCheck^K	4	9	3	0	4	30	4	0	4	130	7	0
MacCheck^D	4	12	3	0	4	40	3	0	4	170	3	0
MultCheck$_1$	4	6	0	0	4	20	0	0	4	80	0	0
MultCheck$_2^M$	5	9	1	0	5	30	1	0	5	130	1	0
MultCheck$_2^K$	5	6	3	0	5	20	4	0	5	90	7	0
MultCheck$_2^D$	5	9	3	0	5	30	3	0	5	130	3	0

	Shamir (3, 1) for $\mathbb{Z}_{2^k}$				Shamir (5, 2) for $\mathbb{Z}_{2^k}$				Shamir (10, 4) for $\mathbb{Z}_{2^k}$			
Protocol	Rounds	Data	PRSS/PRZS	Triples	Rounds	Data	PRSS/PRZS	Triples	Rounds	Data	PRSS/PRZS	Triples
Share	1	4	0	0	1	12	0	0	1	36	0	0
OpenToOne	1	4	0	0	1	12	0	0	1	36	0	0
OpenToAll	1	3	0	0	1	20	0	0	1	130	0	0
BeaverMult	1	6	0	1	1	40	0	1	1	260	0	1
MaurerMult	1	12	0	0	1	60	0	0	1	360	0	0
KRSWMult	1	9	5	0	1	52	5	0	1	280	32	0
DNMult	1	6	2	0	1	20	2	0	1	90	2	0
MacCheck^M	4	18	1	0	4	100	1	0	4	620	1	0
MacCheck^K	4	15	6	0	4	92	6	0	4	540	33	0
MacCheck^D	4	12	3	0	4	60	3	0	4	350	3	0
MultCheck$_1$	4	6	0	0	4	40	0	0	4	260	0	0
MultCheck$_2^M$	5	15	1	0	5	80	1	0	5	490	1	0
MultCheck$_2^K$	5	12	6	0	5	72	6	0	5	410	33	0
MultCheck$_2^D$	5	9	3	0	5	40	3	0	5	220	3	0

Table 5.3: Costs of the Base protocols for Various Access Structures

5.6 Offline Preprocessing Protocols

Given the previous components there are a large number of variations one can deploy to obtain an MPC protocol for a $\mathcal{Q}_2$ access structure which is actively secure with abort. In many cases, some form of preprocessing is used to generate multiplication triples. In this section, we aim to give an overview of different methods to generate passive and active multiplication triples, and evaluate the associated cost in terms of their round and data complexity. We give one passively secure offline protocol, and three actively secure variants. To generate actively secure multiplication triples, we generally first generate passively secure triples, and then we check for correctness (against potential additive attacks) in different ways.

Some of these offline protocols inherently require working (internally) with an extension of the modulus p^{k+v}, whilst all can produce triples modulo p^k or p^{k+v} depending on whether the output protocol requires triples modulo p^k or p^{k+v}. Whether the output is modulo p^k or p^{k+v} will depend into which main protocol we will embed the offline protocol. When we want to distinguish these various cases we will write $\mathsf{Offline}_X(p^{\mathsf{output}}, p^{\mathsf{internal}})$ for an offline protocol which outputs triples modulo p^{output}, whilst working internally modulo p^{internal}. Note, if $\mathsf{output} = k + v$ then we must have $\mathsf{internal} = k + v$ as well. In all cases we assume that all PRSS and PRZS operations are performed non-interactively, and all passive secure multiplications will be assumed to be performed using which ever is the best out of KRSW or DN for the specific parameter sets[9].

$\mathsf{Offline}_{\mathsf{Pass}}$:

When generating N passively secure multiplication triples, we take the approach of first generating $2 \cdot N$ random sharings by performing $2 \cdot N$ calls to PRSS. Following that, we perform a passively secure multiplication protocol N times in parallel to compute the product over pairs of those shares. Since we can perform the N required multiplications in parallel, for the multiplication we only need a single round of communication, with a total data cost of $N \cdot \mathsf{PassMult}_{\mathsf{data}}$, and a corresponding cost of $\mathsf{PassMult}_{\mathsf{data}}$ per triple produced. We simplify the presentation in Table 5.4 by writing $\mathsf{Offline}_{\mathsf{Pass}}(p^k)$ for $\mathsf{Offline}_{\mathsf{Pass}}(p^k, p^k)$ and $\mathsf{Offline}_{\mathsf{Pass}}(p^{k+v})$ for $\mathsf{Offline}_{\mathsf{Pass}}(p^{k+v}, p^{k+v})$.

[9]These are both cheaper than Maurer in terms of data transfer, although they require more PRSS and PRZS calls.

$\mathsf{Offline}_1$:

The first actively secure protocol, $\mathsf{Offline}_1$, will follow the ideas presented in [DPSZ12], in that to generate N actively secure multiplication triples it starts by executing $\mathsf{Offline}_{\mathsf{Pass}}$ to produce $2 \cdot N$ triples. Then half of the obtained triples are sacrificed, using $\mathsf{MultCheck}_1$, so as to check the remaining half for correctness.

The cost of $\mathsf{Offline}_{\mathsf{Pass}}$ is given above. For the verification stage we apply an application of $\mathsf{MultCheck}_1$ on two vectors of triples, each of length N. This requires two rounds of communication and $(2 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}$ in data transferred. This means, amortizing for the number of multiplications, that there are three rounds of communication in total and a data transfer, per triple, of

$$2 \cdot \mathsf{PassMult}_{\mathsf{data}} + (2 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}.$$

However, note that $\mathsf{MultCheck}_1$ requires $\mathsf{Offline}_1$ to work over the extended ring $\mathbb{Z}_{p^{k+v}}$ and therefore each multiplication requires $(k + v) \cdot \log_2(p)$ bits to be transferred, irrespective of the modulus of the output triples, leading to

$$(2 \cdot \mathsf{PassMult}_{\mathsf{data}} + (2 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}) \cdot (k + v) \cdot \log_2(p)$$

irrespective of whether $\mathsf{output} = k$ or $\mathsf{output} = k + v$ or not. Thus, the cost of $\mathsf{Offline}_1(p^k, p^{k+v})$ and $\mathsf{Offline}_1(p^{k+v}, p^{k+v})$ are identical. Thus, in our table below (Table 5.4) we simply write $\mathsf{Offline}_1(p^{k+v})$.

$\mathsf{Offline}_2$:

For the second active offline protocol, $\mathsf{Offline}_2$, we follow [EKO+20]. First N passively secure triples are generated using $\mathsf{Offline}_{\mathsf{Pass}}$. Then these triples are checked to be resistant to additive attacks by running $\mathsf{MultCheck}_2$ on the vector of N triples.

The communication in $\mathsf{MultCheck}_2$ requires three rounds of interaction and the data cost is $\mathsf{PassMult}_{\mathsf{data}} + (1 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}$. This renders the cost for the full protocol, amortizing for the number of multiplications, to be four rounds of communication and

$$2 \cdot \mathsf{PassMult}_{\mathsf{data}} + (1 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}$$

in data transferred. Much like $\mathsf{Offline}_1$ this protocol works over $\mathbb{Z}_{p^{k+v}}$ and therefore all data transferred is $(k + v) \cdot \log_2(p)$ bits, irrespective of the modulus for the output triples. This means that the total data transferred is

$$(k + v) \cdot \log_2(p) \cdot (2 \cdot \mathsf{PassMult}_{\mathsf{data}} + (1 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}).$$

Again, the cost of $\mathsf{Offline}_2(p^k, p^{k+v})$ and $\mathsf{Offline}_2(p^{k+v}, p^{k+v})$ are identical. Again, in our table below (Table 5.4) we simply write $\mathsf{Offline}_2(p^{k+v})$.

$\mathsf{Offline}_3$:

For our third variant of the Offline protocol, which we call $\mathsf{Offline}_3$, we use the cut-and-choose methodology of [ABF+17, Protocol 3.1]. This is parametrized by four integer parameters (Bk, C, X, L), and it generates $N = (X - C) \cdot L$ triples in each iteration, given input of $T = (N + C \cdot L) \cdot (\mathsf{Bk} - 1) + N$ passively secure triples. The value Bk represents a bucket size for the final checking procedure. The advantage of this version of the Offline protocol is that we achieve active security without needing to extend the ring, i.e. we can work modulo p^k and not work p^{k+v} if we require triples modulo p^k as output.

The T triples are initially generated using $\mathsf{Offline}_{\mathsf{Pass}}$ and to perform the check, the T triples are divided into Bk sets. The first D_1 of size N, whilst the rest $D_2, \ldots, D_{\mathsf{Bk}}$, of size $N + L \cdot C$. The sets $D_2, \ldots, D_{\mathsf{Bk}}$ are further subdivided into sets of size X, $D_{i,j}$ for $i = 2, \ldots, \mathsf{Bk}$ and $j = 1, \ldots, L$. The elements of set $D_{i,j}$ are then randomly permuted within each other, and then a random permutation is applied to the vector $(1, \ldots, L)$, so as to randomly permute the sets $D_2, \ldots, D_{\mathsf{Bk}}$. The permutation is done in this way to ensure cache efficiency. Finally, the first C triples in each subarray $D_{i,j}$, for $i = 2, \ldots, \mathsf{Bk}$ and $j = 1, \ldots, L$ are opened and verified to be correct. Thus, this set requires $3 \cdot C \cdot (\mathsf{Bk} - 1) \cdot L$ parallel calls to $\mathsf{OpenToAll}$, resulting in one round of communication and data transfer of

$$3 \cdot C \cdot (\mathsf{Bk} - 1) \cdot L \cdot \mathsf{OpenToAll}_{\mathsf{data}}.$$

The final step is to divide the remaining $\mathsf{Bk} \cdot N$ triples into N buckets of size Bk, with one triple in each bucket from D_1, and the rest from one of $D_2, \ldots, D_{\mathsf{Bk}}$. With very high probability we know the triples in $D_2, \ldots, D_{\mathsf{Bk}}$ are all correct. Thus, this final check can be performed using $(\mathsf{Bk} - 1)$ parallel calls to $\mathsf{MultCheck}'_1$, each containing N elements. This requires two rounds of communication and a total data transfer of $N \cdot (\mathsf{Bk} - 1) \cdot (2 + w/B) \cdot \mathsf{OpenToAll}_{\mathsf{data}}$.

Including the generation of triples, this requires a total of four rounds of communication and a total data transfer of

$$\begin{aligned}\frac{1}{N}\cdot\Big(T\cdot\mathsf{PassMult}_{\mathsf{data}}+3\cdot C\cdot(\mathsf{Bk}-1)\cdot L\cdot\mathsf{OpenToAll}_{\mathsf{data}}\\ +N\cdot(\mathsf{Bk}-1)\cdot(2+w/B)\cdot\mathsf{OpenToAll}_{\mathsf{data}}\Big)\\ =\Big((\mathsf{Bk}-1)\cdot(1+C\cdot L/N)+1\Big)\cdot\mathsf{PassMult}_{\mathsf{data}}\\ +\Big((3\cdot C\cdot L/N)+(2+w/B)\Big)\cdot(\mathsf{Bk}-1)\cdot\mathsf{OpenToAll}_{\mathsf{data}}.\end{aligned}$$

The last consideration to be had regarding the cost of this protocol is that using $\mathsf{MultCheck}'_1$ allows $\mathsf{Offline}_3$ to work, not in $\mathbb{Z}_{p^{k+v}}$, but in $\mathbb{Z}_{p^{\mathsf{output}}}$. Thus, if we have $\mathsf{output}=k$ then we do not need to work at modulo p^{k+v} when running this offline variant.

The statistical security offered by this approach is $1/N^{\mathsf{Bk}-1}$ when used as a standalone offline procedure, or $1/N^{\mathsf{Bk}}$ when used with a specific online procedure (see the third protocol of [ABF+17] for the details); note in the latter case one needs to select $C\geq 3$ and that this corresponds to our Protocol 4 below. In [ABF+17] the authors, for $p^k=2$, target a statistical security level of $\kappa=40$ bits. Thus, they can select $N=2^{20}$, $\mathsf{Bk}=2$, $L=512$ and $C=3$ to achieve an offline cost of 12 bits per triple when utilized in Protocol 4 below.

To provide a fair comparison between all protocols in this chapter we target a statistical security level of $\kappa=128$. Thus, when using $\mathsf{Offline}_3$ in Protocol 1 below we use the parameters $(N,\mathsf{Bk},L,C)=(2^{22},7,512,1)$ and when using $\mathsf{Offline}_3$ in Protocol 4 below we use the parameters $(N,\mathsf{Bk},L,C)=(2^{22},6,512,3)$. To simplify the presentation in Table 5.4 by writing $\mathsf{Offline}_3(p^k)$ for $\mathsf{Offline}_3(p^k,p^k)$ and $\mathsf{Offline}_3(p^{k+v})$ for $\mathsf{Offline}_3(p^{k+v},p^{k+v})$, and we give the costs for the parameters $(N,\mathsf{Bk},L,C)=(2^{22},6,512,3)$ in the table.

5.6.1 Comparing Actively Secure Offline Protocols

Having analysed the three actively secure offline protocols one could compare them theoretically, using the formulae. This is alas however not that illuminative, due to the complexity of the various parameters for $\mathsf{Offline}_3$. Comparing $\mathsf{Offline}_1$ vs $\mathsf{Offline}_2$, is simpler as $\mathsf{Offline}_1$ is better in terms of number of rounds of communication, whereas $\mathsf{Offline}_2$ is better in terms of the amount of data sent per multiplication. To allow a more direct comparison we present the precise values for our different access structures and ring/field sizes in Table 5.4. We assume a security parameter of $\kappa = 128$, and choices of (u, v, w, B) from Section 5.5 so that w/B can be ignored. In Table 5.4 we present the number of bits per triples that need to be transferred.

Access Structure	Ring	Scheme	Mult	$\mathsf{Offline}_{\mathsf{Pass}}(p^k)$	$\mathsf{Offline}_{\mathsf{Pass}}(p^{k+v})$	$\mathsf{Offline}_1(p^{k+v})$	$\mathsf{Offline}_2(p^{k+v})$	$\mathsf{Offline}_3(p^k)$	$\mathsf{Offline}_3(p^{k+v})$
(3, 1)	$\mathbb{F}_2$	Replicated	KRSW	3	387	1548	1161	57	7356
(3, 1)	$\mathbb{F}_2$	Shamir $\mathbb{Z}_{2^k}$	DN	6	774	2322	1935	78	10066
(3, 1)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	384	768	3072	2304	7299	14599
(3, 1)	$\mathbb{Z}_{2^{128}}$	Shamir $\mathbb{Z}_{2^k}$	DN	768	1536	4608	3840	9988	19976
(3, 1)	$\mathbb{F}_p$	Replicated	KRSW	384	768	3072	2304	7299	14599
(3, 1)	$\mathbb{F}_p$	Shamir	KRSW	384	768	3072	2304	7299	14599
(5, 2)	$\mathbb{F}_2$	Replicated	KRSW	10	1290	7740	5160	310	40010
(5, 2)	$\mathbb{F}_2$	Shamir $\mathbb{Z}_{2^k}$	DN	20	2580	10320	7740	380	49043
(5, 2)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	1280	2560	15360	10240	39700	79399
(5, 2)	$\mathbb{Z}_{2^{128}}$	Shamir $\mathbb{Z}_{2^k}$	DN	2560	5120	20480	15360	48662	97325
(5, 2)	$\mathbb{F}_p$	Replicated	KRSW	1280	2560	15360	10240	39700	79399
(5, 2)	$\mathbb{F}_p$	Shamir	KRSW	1280	2560	10240	7680	24331	48662
(10, 4)	$\mathbb{F}_2$	Replicated	KRSW	50	6450	229620	121260	10436	1346198
(10, 4)	$\mathbb{F}_2$	Shamir $\mathbb{Z}_{2^k}$	DN	90	11610	56760	39990	2191	282646
(10, 4)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	6400	12800	455680	240640	1335763	2671526
(10, 4)	$\mathbb{Z}_{2^{128}}$	Shamir $\mathbb{Z}_{2^k}$	DN	11520	23040	112640	79360	280455	560910
(10, 4)	$\mathbb{F}_p$	Replicated	KRSW	6400	12800	455680	240640	1335763	2671526
(10, 4)	$\mathbb{F}_p$	Shamir	KRSW	6400	12800	46080	35840	106288	212576

Table 5.4: Costs of the Offline Protocols in number of bits per multiplication, for various access structures; $\kappa = 128$, $p \approx 2^{128}$

5.7 Complete Protocols

We now examine the five (main) protocol variants we discussed in the introduction. For each of the following protocols, if an actively secure offline phase is required we can utilize the protocols $\mathsf{Offline}_x$, for x either 1, 2 or 3, given in Section 5.6. There are two basic metrics here that one could be interested in (assuming to a first order approximation we are processing arithmetic circuits over $\mathbb{Z}_{p^k}$), namely, the amount of data transferred per multiplication in the online phase only, or the amount of data transferred per multiplication in the combined online and offline phases. These two metrics capture the different potential use cases of whether pre-processing is considered a cost or not; which would depend on the precise implementation within a commercial environment. Note, here we consider the cost of any post-processing to be considered within the online phase costs. In all cases we assume we are processing an arithmetic circuit with N multiplication gates in a circuit of multiplicative depth d. In our calculations of costs below we ignore any round or data communication costs due to the input or output of the function; since these are usually negligible in comparison to the functions multiplicative complexity.

$\mathsf{Protocol}_1$:

This protocol executes an actively secure offline phase to produce N triples in $\mathbb{Z}_{p^k}$, i.e. we execute $\mathsf{Offline}_x(p^k, p^\star)$ for $\star$ being either k or $k+v$, depending on the precise protocol choice x. Note, this means we have three choices for $\mathsf{Protocol}_1$ depending on which offline protocol the main protocol is combined with. The online phase is executed, using these triples, using $\mathsf{BeaverMult}$ as the multiplication procedure. Since the Beaver multiplication is instantiated with actively secure triples the output will also be actively secure, and no post-processing check is necessary.

Recall that each instantiation of $\mathsf{BeaverMult}$ requires one round of communication and a total of $2 \cdot \mathsf{OpenToAll}_{\mathsf{data}}$ in data transferred. Thus, our we have the online phase requires d rounds of communication and

$$2 \cdot \log_2(p) \cdot k \cdot \mathsf{OpenToAll}_{\mathsf{data}}$$

data communication. The online cost does not depend on the choice of offline phase.

If we look at the combined cost of the online and offline phases then we will require $\mathsf{Offline}_{\mathsf{rounds}} + d$ rounds of communication and a data communication cost of

$$\mathsf{Offline}_{\mathsf{data}} + 2 \cdot \log_2(p) \cdot k \cdot \mathsf{OpenToAll}_{\mathsf{data}}$$

per multiplication, where $\mathsf{Offline}_{\mathsf{data}}$ is taken from the relevant columns of Table 5.4. In Table 5.5 we refer to the three combined costs per multiplication as Total_x, depending on which Offline phase we are utilizing.

$\mathsf{Protocol}_2$:

In this protocol we optimistically use a passively secure online multiplication protocol $\mathsf{PassMult}$ to execute the online phase, and a passively secure Offline protocol to generate N passively secure multiplication triples, all over $\mathbb{Z}_{p^{k+v}}$. These are then checked using a post-processing methodology, based on $\mathsf{MultCheck}_1$, to ensure active security. This approach of optimistic, passively secure online multiplication was first suggested in [EKO+20].

The number of rounds for the online phase is $d + 4$, as we require four rounds to execute $\mathsf{MultCheck}_1$, and the total communication for the online phase, per multiplication gate, is

$$\log_2(p) \cdot (k + v) \cdot (\mathsf{PassMult}_{\mathsf{data}} + \mathsf{MultCheck}_{1,\mathsf{data}}),$$

where $\mathsf{PassMult}_{\mathsf{data}}$ and $\mathsf{MultCheck}_{1,\mathsf{data}}$ are taken from Table 5.2.

If we look at the combined cost of the online and offline phases then we will require $\mathsf{Offline}_{\mathsf{rounds}} + d + 4$ rounds of communication and a data communication cost of

$$\mathsf{Offline}_{\mathsf{Pass}}(p^{k+v})_{\mathsf{data}} + \log_2(p) \cdot (k + v) \cdot (\mathsf{PassMult}_{\mathsf{data}} + \mathsf{MultCheck}_{1,\mathsf{data}})$$

per multiplication, where $\mathsf{Offline}_{\mathsf{Pass}}(p^{k+v})_{\mathsf{data}}$ is taken from Table 5.4.

$\mathsf{Protocol}_3$:

This proceeds very much as $\mathsf{Protocol}_2$ except instead of using an offline phase and the $\mathsf{MultCheck}_1$ procedure, one uses the $\mathsf{MultCheck}_2$ procedure. As there is no offline phase, online and post-processing costs are the total costs of the protocol. Again all operations needs to be performed over $\mathbb{Z}_{p^{k+v}}$.

The number of rounds for the online phase is now $d+5$, as we require five rounds to execute $\mathsf{MultCheck}_2$, and the total communication for the online phase, per multiplication gate, is

$$\log_2(p) \cdot (k + v) \cdot (\mathsf{PassMult}_{\mathsf{data}} + \mathsf{MultCheck}_{2,\mathsf{data}}),$$

where $\mathsf{PassMult}_{\mathsf{data}}$ and $\mathsf{MultCheck}_{2,\mathsf{data}}$ are taken from Table 5.2.

$\mathsf{Protocol}_4$:

This protocol variant follows the pattern from [ABF+17] and thus is particularly suited to small values of p^k. It can be applied using any of the actively secure offline protocols, but is better suited (for small p^k) to be used with $\mathsf{Offline}_3$.

In the offline phase we generate N actively secure multiplication triples in $\mathbb{Z}_{p^k}$. In the online phase a standard passively secure online phase is executed, using $\mathsf{PassMult}$. Then in the post-processing the triples produced in the offline phase are checked against the 'triples' resulting from the passively secure multiplications, using $\mathsf{MultCheck}'_1$. The entire procedure can be executed in $\mathbb{Z}_{p^k}$ without the need to extend to $\mathbb{Z}_{p^{k+v}}$.

The number of rounds for the online phase is $d+4$, as we require four rounds to execute $\mathsf{MultCheck}'_1$, and the total communication for the online phase, per multiplication gate, is

$$\log_2(p) \cdot k \cdot (\mathsf{PassMult}_{\mathsf{data}} + \mathsf{MultCheck}_{1,\mathsf{data}}),$$

where $\mathsf{PassMult}_{\mathsf{data}}$ and $\mathsf{MultCheck}_{1,\mathsf{data}}$ are taken from Table 5.2.

If we look at the combined cost of the online and offline phases then we will require $\mathsf{Offline}_{\mathsf{rounds}} + d + 4$ rounds of communication and a data communication cost of

$$\mathsf{Offline}_{\mathsf{data}} + \log_2(p) \cdot k \cdot (\mathsf{PassMult}_{\mathsf{data}} + \mathsf{MultCheck}_{1,\mathsf{data}})$$

per multiplication, where $\mathsf{Offline}_{\mathsf{data}}$ is taken from Table 5.4. Again in Table 5.5 we will refer to the three different combined costs per multiplication as Total_x.

$\mathsf{Protocol}_5$:

Our final approach is based upon the technique in [CGH+18]. At the start of the protocol, in a (very short) offline phase a sharing for an unknown, secret random value $[\alpha]_{k+v}$ is generated. This value is used as an information theoretic MAC key, similar to the SPDZ approach.

In the online phase each wire value x is held as two shared values $\{[x]_{k+v}, [\alpha \cdot x]_{k+v}\}$. To multiply two values x and y we execute a passively secure multiplication twice, once with $[x]_{k+v}$ and $[y]_{k+v}$ to obtain $[x \cdot y]_{k+v}$, and one with $[x]_{k+v}$ and $[\alpha \cdot y]_{k+v}$ to obtain $[\alpha \cdot x \cdot y]_{k+v}$. In a short post-processing phase the MAC values on *all multiplication gates* and *all input and output wires* are checked using the $\mathsf{MacCheck}$ procedure. To ensure the security of the $\mathsf{MacCheck}$ procedure all computation need to be performed in $\mathbb{Z}_{p^{k+v}}$.

The data cost for the preprocessing can be amortized away over all multiplications, as we only need a single $[\alpha]_{k+v}$, regardless of the total number of multiplications we need to perform. Thus, there is no cost essentially to the offline phase.

The number of rounds for the online phase is $d + 4$, as we require four rounds to execute MacCheck, and the total communication for the online phase, per multiplication gate, is

$$\log_2(p) \cdot (k + m) \cdot (2 \cdot \mathsf{PassMult}_{\mathsf{data}} + \mathsf{MacCheck}_{\mathsf{data}}),$$

where $\mathsf{PassMult}_{\mathsf{data}}$ and $\mathsf{MacCheck}_{\mathsf{data}}$ are taken from Table 5.2.

It has to be noted that the protocol also needs to perform a (passive) multiplication for every input wire, so as to obtain the initial authenticated shares, as described in [CGH+18]. These multiplications also need to be checked for consistency with the evaluation of MacCheck, but we focus on the cost per multiplication here, and thus do not account for this extra cost.

We can now present a summary (in Table 5.5) of all these options, by way of presenting their respective online and total communications costs (in number of bits communicated per multiplication), for a variety of different scenarios, access structures and base rings. Again, in all cases we utilize which ever of KRSW or DN, for the passive multiplication procedure, which results in the least amount of data transmitted for the specific LSSS under consideration. We also use again the choices of (u, v, w, B) from Section 5.5 so that w/B can be ignored, and a security parameter of $\kappa = 128$. In the table we mark in blue the online variant which is most efficient for a given access structure, ring, and ESP. This is almost always $\mathsf{Protocol}_1$. We also mark in gray the most efficient protocol option when one is interested in the total cost. For small rings this is always $\mathsf{Protocol}_4$ with $\mathsf{Offline}_3$ chosen as the pre-processing, for the others it is $\mathsf{Protocol}_5$. The data for this table was generated with publicly available code[10] that also provides a clean and extensible framework allowing for more online and offline protocols.

Note, that in the case of $\mathsf{Protocol}_4$ and $\mathsf{Offline}_3$ the paper [ABF+17] obtains a total cost of 21 bits per multiplication operation. As explained earlier this is because they target a statistical security level of $\kappa = 40$, instead of our security level of $\kappa = 128$.

Note that even when $\mathsf{Protocol}_1$ is not the most efficient choice, in practice one might still prefer using this protocol as our analysis assumes the only interaction occurs for multiplication. Most MPC protocols make use of OpenToAll

[10] `https://github.com/KULeuven-COSIC/MPCEstimator`

Access				Protocol$_1$				Protocol$_2$	
Structure	Ring	Scheme	Mult	Online	Total$_1$	Total$_2$	Total$_3$	Online	Total
(3, 1)	$\mathbb{F}_2$	Replicated	KRSW	6	1554	1167	63	1161	1548
(3, 1)	$\mathbb{F}_2$	Shamir $\mathbb{Z}_{2^k}$	DN	6	2328	1941	84	1548	2322
(3, 1)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	768	3840	3072	8067	2304	3072
(3, 1)	$\mathbb{Z}_{2^{128}}$	Shamir $\mathbb{Z}_{2^k}$	DN	768	5376	4608	10756	3072	4608
(3, 1)	$\mathbb{F}_p$	Replicated	KRSW	768	3840	3072	8067	2304	3072
(3, 1)	$\mathbb{F}_p$	Shamir	KRSW	768	3840	3072	8067	2304	3072
(5, 2)	$\mathbb{F}_2$	Replicated	KRSW	40	7780	5200	350	6450	7740
(5, 2)	$\mathbb{F}_2$	Shamir $\mathbb{Z}_{2^k}$	DN	40	10360	7780	420	7740	10320
(5, 2)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	5120	20480	15360	44820	12800	15360
(5, 2)	$\mathbb{Z}_{2^{128}}$	Shamir $\mathbb{Z}_{2^k}$	DN	5120	25600	20480	53782	15360	20480
(5, 2)	$\mathbb{F}_p$	Replicated	KRSW	5120	20480	15360	44820	12800	15360
(5, 2)	$\mathbb{F}_p$	Shamir	KRSW	2560	12800	10240	26891	7680	10240
(10, 4)	$\mathbb{F}_2$	Replicated	KRSW	1680	231300	122940	12116	223170	229620
(10, 4)	$\mathbb{F}_2$	Shamir $\mathbb{Z}_{2^k}$	DN	260	57020	40250	2451	45150	56760
(10, 4)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	215040	670720	455680	1550803	442880	455680
(10, 4)	$\mathbb{Z}_{2^{128}}$	Shamir $\mathbb{Z}_{2^k}$	DN	33280	145920	112640	313735	89600	112640
(10, 4)	$\mathbb{F}_p$	Replicated	KRSW	215040	670720	455680	1550803	442880	455680
(10, 4)	$\mathbb{F}_p$	Shamir	KRSW	10240	56320	46080	116528	33280	46080

Access		Protocol$_3$		Protocol$_4$				Protocol$_5$	
Structure	Ring	Online	Total	Online	Total$_1$	Total$_2$	Total$_3$	Online	Total
(3, 1)	$\mathbb{F}_2$	1161	1161	9	1557	1170	57	774	774
(3, 1)	$\mathbb{F}_2$	1935	1935	12	2334	1947	78	1548	1548
(3, 1)	$\mathbb{Z}_{2^{128}}$	2304	2304	1152	4224	3456	7299	1536	1536
(3, 1)	$\mathbb{Z}_{2^{128}}$	3840	3840	1536	6144	5376	9988	3072	3072
(3, 1)	$\mathbb{F}_p$	2304	2304	1152	4224	3456	7299	1536	1536
(3, 1)	$\mathbb{F}_p$	2304	2304	1152	4224	3456	7299	1536	1536
(5, 2)	$\mathbb{F}_2$	5160	5160	50	7790	5210	310	2580	2580
(5, 2)	$\mathbb{F}_2$	7740	7740	60	10380	7800	380	5160	5160
(5, 2)	$\mathbb{Z}_{2^{128}}$	10240	10240	6400	21760	16640	39696	5120	5120
(5, 2)	$\mathbb{Z}_{2^{128}}$	15360	15360	7680	28160	23040	48659	10240	10240
(5, 2)	$\mathbb{F}_p$	10240	10240	6400	21760	16640	39696	5120	5120
(5, 2)	$\mathbb{F}_p$	7680	7680	3840	14080	11520	24329	5120	5120
(10, 4)	$\mathbb{F}_2$	121260	121260	1730	231350	122990	10435	12900	12900
(10, 4)	$\mathbb{F}_2$	39990	39990	350	57110	40340	2191	23220	23220
(10, 4)	$\mathbb{Z}_{2^{128}}$	240640	240640	221440	677120	462080	1335642	25600	25600
(10, 4)	$\mathbb{Z}_{2^{128}}$	79360	79360	44800	157440	124160	280432	46080	46080
(10, 4)	$\mathbb{F}_p$	240640	240640	221440	677120	462080	1335642	25600	25600
(10, 4)	$\mathbb{F}_p$	35840	35840	16640	62720	52480	106280	25600	25600

Table 5.5: Costs of the Full Protocols in number of bits per multiplication, for various access structures; $\kappa = 128$, $p \approx 2^{128}$

executions to open masked data for use in various function specific optimizations. Using Protocol$_1$ enables these protocol-specific OpenToAll executions to be merged easily with the OpenToAll executions used in multiplication; thus reducing the total round count. For other online protocols this merging can be more complex.

Acknowledgements

We would like to thank Daniel Escudero and Tim Wood for conversations on aspects of this work whilst it was carried out.

This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) and

Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract FA8750-19-C-0502, by the FWO under an Odysseus project GOH9718N, and by CyberSecurity Research Flanders with reference number VR20192203.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the funders. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

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. IEEE Computer Society Press, May 2017.

[ACD+19] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Cham, December 2019.

[ACD+20] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, Matthieu Rambaud, Chaoping Xing, and Chen Yuan. Asymptotically good multiplicative LSSS over Galois rings and applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 151–180. Springer, Cham, December 2020.

[ADEN19] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. Cryptology ePrint Archive, Report 2019/1298, 2019.

[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*, volume 5283 of *LNCS*, pages 192–206. Springer, Berlin, Heidelberg, October 2008.

[CDE+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Cham, August 2018.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Berlin, Heidelberg, February 2005.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, Berlin, Heidelberg, May 2000.

[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, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Cham, August 2018.

[CRX19] Ronald Cramer, Matthieu Rambaud, and Chaoping Xing. Asymptotically-good arithmetic secret sharing over $Z/(p^\ell Z)$ with strong multiplication and its applications to efficient MPC. Cryptology ePrint Archive, Report 2019/832, 2019.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Berlin, Heidelberg, August 2007.

[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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.

[EKO$^+$20] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 5:1–5:24. Schloss Dagstuhl, June 2020.

[Feh98] Serge Fehr. Span programs over rings and how to share a secret from a module, 1998. MSc Thesis, ETH Zurich.

[Gá95] Anna Gál. Combinatorial methods in boolean function complexity, 1995. PhD Theses, University of Chicago.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.

[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*, pages 830–842. ACM Press, October 2016.

[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*, volume 11035 of *LNCS*, pages 181–199. Springer, Cham, September 2018.

[Mau06] Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[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, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 210–229. Springer, Cham, March 2019.

5.A Proof of Theorem 5.1

In this Appendix we discuss the proof of Theorem 5.1. To do this assume that $\mathcal{M} = \left(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\right)$ is an ESP computing a $\mathcal{Q}_2$ access structure Γ, such that $\mathcal{M}$ is not multiplicative. Then in two steps we can arrive at a multiplicative ESP. For this consider the ESPs $\mathcal{M}_0 = \left(\mathbb{Z}_{p^k}, M_0, \mathbf{e}_1, \varphi\right)$ and $\mathcal{M}_1 = \left(\mathbb{Z}_{p^k}, M_1, \mathbf{e}_1, \varphi\right)$ and let Γ_0 and Γ_1 be the access structures computed by $\mathcal{M}_0$ and $\mathcal{M}_1$ respectively.

Lemma 5.12

Suppose that $M_0^T \cdot M_1 = [\mathbf{e}_1, \mathbf{0}, \ldots, \mathbf{0}]$, then there exists a multiplicative ESP computing $\Gamma_0 \vee \Gamma_1$ of size at most $2(k + |\mathcal{P}|)$.

Proof. By [Feh98], we have that ESP's compute an additively homomorphic perfect LSSS, say LSSS_0 and LSSS_1 for $\mathcal{M}_0, \mathcal{M}_1$. Consider the LSSS generated by $\mathcal{M}_0$ and $\mathcal{M}_1$ simultaneously, so $\langle \mathbf{s}_0, \varepsilon \rangle = \langle \mathbf{s}_1, \varepsilon \rangle = s$. Let $\mathbf{s} = (\mathbf{s}_0, \mathbf{s}_1)$ be the share vector with the ith coordinates of $\mathbf{s}_0$ and $\mathbf{s}_1$ sent to $\mathcal{P}_{\varphi(i)}$. Clearly, if and only if A is a qualifying set for $\Gamma_0 \vee \Gamma_1$ this set can reconstruct s from their joint shares.

Now consider multiplication: Assume $s' \in \mathbb{Z}_{p^k}$ is a secret with secret vectors $(\mathbf{s}'_0, \mathbf{s}'_1)$. Let $\mathbf{s}_0 * \mathbf{s}'_1$ be the d-vector obtained by coordinate-wise multiplication. Then

$$\begin{aligned}\langle \mathbf{1}, \mathbf{s}_0 * \mathbf{s}'_1 \rangle &= \mathbf{s}_0^T \cdot \mathbf{s}'_1 = \mathbf{b}_0^T \cdot M_0^T \cdot M_1 \cdot \mathbf{b}_1 \\ &= \mathbf{b}_0^T \cdot E \cdot \mathbf{b}'_1 \qquad\qquad \text{as } M_0^T \cdot M_1 = E \\ &= s \cdot s'.\end{aligned}$$

Let M_i^j be the jth column vector of M_i and define the matrix M' such that

$$M' = \begin{pmatrix} M_0^1 & M_0^2 & \dots & M_0^{n+1} & \mathbf{0} & \dots & \mathbf{0} \\ M_1^1 & \mathbf{0} & \dots & \mathbf{0} & M_1^2 & \dots & M_1^{n+1} \end{pmatrix}$$

Then $\mathcal{M} = (\mathbb{Z}_{p^k}, M', \mathbf{e}_1, \varphi)$ corresponds to the constructed LSSS above, [CDM00]: The product of $(\mathbf{s}_0, \mathbf{s}_1)$ and $(\mathbf{s}'_0, \mathbf{s}'_1)$ contains among its entries $\mathbf{s}_0 * \mathbf{s}_1$, and a recombination vector λ exists, so $\mathcal{M}$ is multiplicative. □

This turns out to be useful as Fehr proved that given a mild inflation you can change the target vector of an ESP, [Feh98]:

Lemma 5.13

Let $\mathcal{M} = (R, M, \varepsilon, \varphi)$, with $M \in M_{n\times m}(R)$ be an ESP computing access structure Γ, then there exists an extended span program $\mathcal{M}' = (R, M', \mathbf{e}_1, \varphi)$, with $M' \in M_{n'\times m'}(R)$, computing Γ with $n' \leq n + |\mathcal{P}|$ and $m' \leq m + 1$.

This means we can prove the generalization of the theorem for Cramer et al., [CDM00], for Extended Span Programs, i.e. Theorem 5.1.

Proof. **Of Theorem 5.1:** Let Γ be an access structure, and define a boolean function $\gamma : \mathcal{P} \to \{0, 1\}$, such that $\gamma(A) = 0$ if $A \notin \Gamma$ and $\gamma(A) = 1$ if $A \in \Gamma$. Then we can define $\gamma^*(X) = \overline{\gamma(\overline{X})}$ where $\bar{\cdot}$ indicates a flip, that is $\gamma(A) = 0$ then $\overline{\gamma}(A) = 1$ and $\gamma(\overline{A}) = \gamma(\mathcal{P}\backslash A)$. By our assumption the access structure is complete so if a subset A is unqualified, then $\overline{A} = \mathcal{P}\backslash A$ is qualified, therefore $\gamma(A) = 0 \Leftrightarrow \gamma^*(A) = 0$, and therefore, tautologically, $\Gamma = \Gamma \vee \Gamma^*$.

Let $\mathcal{M} = \left(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi\right)$ be an ESP, such that $\varepsilon = \mathbf{e}_1$, computing Γ. Let $\Gamma_0 = \Gamma$, $\Gamma_1 = \Gamma^*$, and $\mathcal{M}_0 = \mathcal{M}$. By Lemma 5.13 all we need to show is that there exists a $\mathcal{M}_1$ such that $M_0^T \cdot M_1 = E$.

As described in [Gá95], there is a construction such that for a given MSP computing Γ you can define a "dual" MSP that computes Γ^* with the same target vector. It is simple to see that this proof can be extended to work over commutative unital Noetherian rings as the recombination vectors exist. In fact this dual MSP can be computed efficiently if $\ker(M^T)$ admits a basis. The only thing that poses an issue is that in [Gá95] the target vector is $\mathbf{1}$. However, this is easy to ensure as follows.

We define the d-dimensional matrix H as

$$H = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & \dots & 0 \\ 1 & 0 & 1 & \dots & 0 \\ & & \vdots & & \\ 1 & 0 & \dots & 0 & 1 \end{pmatrix}$$

inducing a map such that

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{pmatrix} \mapsto \mathbf{x} + \mathbf{x}_1 = \begin{pmatrix} x_1 \\ x_2 + x_1 \\ \dots \\ x_d + x_1 \end{pmatrix}$$

Clearly this is a module isomorphism and by letting $N = M \cdot H^T$ we define an MSP $\mathcal{N}$ exactly as $\mathcal{M}$ but with target vector $\mathbf{1}$. Let $\mathcal{N}^\star$ be its dual given by [Gá95], then

$$\mathcal{M}^* = \left(\mathbb{Z}_{p^k}*, M^* = N^* \cdot \left(H^{-1}\right)^T, \varepsilon, \varphi\right)$$

is the dual MSP computing Γ^* with target vector $\mathbf{e}_1$. Note that $M^T \cdot M^* = H^{-1} \cdot N^T \cdot N^* \cdot (H^{-1})^T = E$ and so the conditions for the lemma hold and the theorem follows. $\square$

5.B KRSW Multiplication Costs

In this Appendix we recap on the optimized variant of the passively secure multiplication method of [KRSW18] for replicated secret sharing (denoted the KRSW method in what follows), and its generalization to arbitrary MSPs of [SW19] (denoted the Smart–Wood method in what follows). The paper [SW19]

The Functionality $\mathcal{F}_{\text{PRZS}}$

This functionality outputs an additive sharing of zero to all players in a given set.

On input (cnt, S) from all parties in a set $S \subset \mathcal{P}$, if the counter value is the same for all parties and has not been used before, the functionality arbitrarily chooses some P_{i^*}, and then for each party P_i in $S \setminus \{P_{i^*}\}$ samples $t_i \leftarrow \mathbb{F}$ uniformly at random, fixes $t_{i^*} \leftarrow -\sum_{i \in S \setminus \{P_{i^*}\}} t_i$ and sends t_i to party P_i for each $i \in S \setminus \{P_{i^*}\}$ and t_{i^*} to P_{i^*}.

Figure 5.13: The Functionality $\mathcal{F}_{\text{PRZS}}$

contains a few of typographical errors in the description of the algorithms, so we correct those errors in our presentation below. We also give the calculations for our nine different examples, so as to illustrate the methods in different examples. We note that [SW19] is not necessarily more efficient than [KRSW18] for replicated secret sharings; we illustrate this with an example below.

Both protocol variants make use of pseudo-random zero-sharings (PRZSs), which are additive sharings of zero. These are used to mask shares before sending them without changing the underlying secret. The functionality is given in Figure 5.13. We do not provide the protocol here as it is given in [KRSW18], but we note that we may trivially extend the protocol there to allow the generation of PRZSs for any subset of parties if we assume pair-wise PRF keys have been created during a one-time setup phase (as in the protocol given) by each P_i computing

$$t_i \leftarrow \sum_{j \neq i, j \in S} F_{\kappa_{i,j}}(\mathsf{cnt}) - F_{\kappa_{j,i}}(\mathsf{cnt}).$$

The key part of the multiplication algorithm, and the only part which requires interaction, is the mechanism Π_{Convert}, (in Figure 5.15 for [KRSW18] and Figure 5.17 for [SW19]), to transfer an additive secret sharing $\langle x \rangle$ amongst all n parties, i.e. $x = x_1 + \cdots + x_n$ where P_i holds x_i, into a secret sharing under the desired MSP/ESP $\mathcal{M} = (R, M, \varepsilon, \varphi)$ where $M \in M_{m \times d}(R)$ is a matrix of rank d.

Π_{Convert} for KRSW:

This methodology makes use of a (minor) extension of the earlier functionality $\mathcal{F}_{\text{AgreeRandom}}(D)$, which we give in Figure 5.14. One can see $\mathcal{F}_{\text{AgreeRandom}}(D)$ as being the special case of $\mathcal{F}_{\text{AgreeRandom}'}(D, \mathcal{P})$. In practice one would execute this (per multiplication) non interactively by first agreeing a seed for all multiplications, and then expanding the seed as required for each multiplication via a PRF.

Ideal Functionality $\mathcal{F}_{\text{AgreeRandom}'}(D, \mathsf{S})$

On input $\mathsf{AgreeRandom}(\mathsf{cnt})$ from all parties, if the counter value is the same for all parties and has not been used before, the functionality samples a value $a \leftarrow D$, and sends a to all parties in $\mathsf{S} \subset \mathcal{P}$.

Figure 5.14: Ideal Functionality $\mathcal{F}_{\text{AgreeRandom}'}(D, \mathsf{S})$

It also uses a map $\chi : [n] \longrightarrow [d]$, which needs to be defined once and for all, which is injective and for which $\chi(i) = k$ implies that $\varphi(j) = i$ for a row j which contains the standard basis vector $\mathbf{e}_k$. In addition, we have a map $\psi : [d] \longrightarrow [n]$ which maps a column of M to a player P_i. The map ψ is defined so that $\psi(\chi(i)) = i$, and for all $k \notin \mathsf{im}(\chi)$ we have that $\psi(k) = i$ implies that there is a row j of M consisting of $\mathbf{e}_k$ such that $\varphi(j) = i$. The overall method for executing the protocol Π_{Convert} is given in Figure 5.15.

Π_{Convert} for Smart–Wood:

In this case, protocol Π_{Convert} utilizes a second ESP which is said to be "good". The main criteria for this second ESP is that it has a relatively large number of zero coefficients, and it is obtained from the original ESP via column operations. In addition, we need to compute a mapping $\chi : [n] \longrightarrow [d]$ of parties to columns. These indicate for each party how to map its additive sharing onto a column of the sharing of the under the ESP. To generate the new ESP and χ we use the algorithm in Figure 5.16.

We now illustrate the methods with our examples:

KRSW Protocol Π_{Convert}

At this point in the protocol, the parties have an additive sharing $\langle x \rangle$, where P_i holds x_i, and will convert it to a sharing under the ESP $(R, M, \varepsilon, \varphi)$ (which is assumed to be a replicated secret sharing scheme). It makes use of the maps $\chi : [n] \longrightarrow [d]$ and $\psi : [d] \longrightarrow [n]$ described in the text.

1. For $k \in [1, \ldots, d]$ let $\mathcal{J}_k$ denote the set of all rows consisting of the standard basis vector $\mathbf{e}_k$ and let $\mathcal{I}_k$ denote the set of parties $\{\varphi(j) : j \in \mathcal{J}_k\}$
2. The parties call $\mathcal{F}_{\text{PRZS}}$ with the command $(\mathsf{cnt}, \mathcal{P})$ to obtain a PRZS, denoted hereafter by $\langle t \rangle$.
3. For $k \notin \mathsf{im}(\chi)$, define $\mathbf{s}_j$ for $j \in \mathcal{J}_k$ by calling $\mathcal{F}_{\text{AgreeRandom}'}(R, \mathcal{I}_k)$.
4. For $k \in \mathsf{im}(\chi)$, define $i = \chi^{-1}(k)$ and player P_i compute, for $j \in \mathcal{J}_k$, the value $\mathbf{s}_j \leftarrow x_i + t_i - \sum \mathbf{s}_{j'}$, where the sum is over all j' such that $\varphi(j') = i$ and j' is a vector $\mathbf{e}_k$ with $\psi(k) = j'$. Party P_i sends $\mathbf{s}_j$ to party $\varphi(j)$ for $j \in \mathcal{J}_k$.

Figure 5.15: KRSW Protocol Π_{Convert} converting additive shares to shares in the LSSS

5.B.1 Replicated $(3, 1)$ Sharing

This secret sharing method for the threshold structure $(n, t) = (3, 1)$ works for any ring $\mathbb{Z}_{p^k}$, for any size p^k. Here our input ESP $(R, M, \varepsilon, \varphi)$ is given by

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

$$\varepsilon = (1, 1, 1),$$

$$\varphi(i) = \lceil i/2 \rceil.$$

Algorithm for computing a "good" ESP

The input is the multiplicative ESP $\mathcal{M} = (R, M, \varepsilon, \varphi)$. The output is a map $\chi : [n] \longrightarrow [d]$ of party indices to columns, and a new ESP $\mathcal{M}'$.

1. Perform column operations on M of $\mathcal{M}$ and the same on ε to obtain an ESP $\mathcal{M}'$ with the same φ and $\mathbb{F}$ but with matrix M' and target vector ε' such that all of the standard basis vectors in R^d, $\{\mathbf{e}_k\}_{i=1}^{d} \subset R^d$, appear as rows of M'.
2. Define the map $\chi : [n] \longrightarrow [d]$ in the following way, making choices so that $\mathsf{im}(\chi)$ is as large as possible:
   - If P_i owns a row which is a standard basis vector $\mathbf{e}_k$, and $\varepsilon_k \neq 0$, then set $\chi(i) \leftarrow k$;
   - If P_i does not own such a row, assign P_i any column k in which P_i owns a row j such that $M_j[k] \neq 0$ and $\varepsilon_k \neq 0$;
   - If no such column exists, find any row j (not necessarily owned by P_i), and any column k such that $M_j[k] \neq 0$ and $\varepsilon_k \neq 0$ and set $\chi(i) = k$.
3. Output χ and $\mathcal{M}'$.

Figure 5.16: Algorithm for computing a "good" ESP

KRSW Algorithm:

We now discuss the methodology for this example of KRSW, i.e. Figure 5.15. We define in this case $\chi(1) = 3$, $\chi(2) = 1$ and $\chi(3) = 2$. Protocol Π_{Convert} then consists of the following steps, on input of $x = x_1 + x_2 + x_3$, where x_i is held by player P_i.

1. We have $\mathcal{J}_1 = \{3, 5\}$, $\mathcal{J}_2 = \{1, 6\}$ and $\mathcal{J}_3 = \{2, 4\}$, and $\mathcal{I}_1 = \{2, 3\}$, $\mathcal{I}_2 = \{1, 3\}$ and $\mathcal{I}_3 = \{1, 2\}$.
2. Call $\mathcal{F}_{\text{PRZS}}$ to generate t^0 with $\sum_i t_i = 0$.
3. Player P_1 sends $\mathbf{s}_2 = \mathbf{s}_4 = x_1 + t_1$ to Player P_2.
4. Player P_2 sends $\mathbf{s}_3 = \mathbf{s}_5 = x_2 + t_2$ to Player P_3.
5. Player P_3 sends $\mathbf{s}_1 = \mathbf{s}_6 = x_3 + t_3$ to Player P_1.

Smart–Wood Protocol Π_{Convert}

At this point in the protocol, the parties have an additive sharing $\langle x\rangle$, where P_i holds x_i, and will convert it to a sharing under the ESP $(R, M', \varepsilon', \varphi)$ using the map χ (both output by the algorithm in Figure 5.16)

1. The parties call $\mathcal{F}_{\text{PRZS}}$ with the command $(\mathsf{cnt}, \mathcal{P})$ to obtain a PRZS, denoted hereafter by $\langle t^0\rangle$.
2. Each P_i splits $x_i + t_i^0$ as $x_i + t_i^0 = \sum_{k\in K_i\cap\mathsf{supp}(\varepsilon')} x_{i,k}$ where $K_i \leftarrow (\{(\chi(P_i)\} \cup ([d] \setminus \mathsf{im}(\chi)))$.
3. Each P_i sets $r_{i,k} \leftarrow x_{i,k}/\varepsilon'_k$ for each $k \in K_i \cap \mathsf{supp}(\varepsilon')$.
4. Each P_i sets $r_{i,k} \leftarrow R$ for each $k \in K_i \setminus \mathsf{supp}(\varepsilon')$.
5. For each row j which is *not* a standard basis vector, the parties do the following
   (a) The parties call $\mathcal{F}_{\text{PRZS}}$ with the command $(\mathsf{cnt}, \mathcal{P})$ to obtain a PRZS amongst them, denoted hereafter by $\langle t^j\rangle$.
   (b) Each P_i computes $a_i^j \leftarrow \left(\sum_{k\in K_i\cap\mathsf{supp}(\varepsilon')} M'_j[k]\cdot r_{i,k}\right) + t_i^j$, where $M_j[k]'$ denotes the kth element of row j of M'.
   (c) Party P_i sends a_i^j to party $\varphi(j)$.
   (d) Party $\varphi(j)$ computes $\mathbf{s}_j \leftarrow \sum_{i=1}^n a_i^j$.
6. Let $\mathcal{J}_k$ denote the rows which are the standard basis vector $\mathbf{e}_k$. For each k execute:
   (a) Let $X_k = \{P_i \in \mathcal{P} : k \in K_i\}$. If $|X_k| > 2$ then call $\mathcal{F}_{\text{PRZS}}$ with the command (cnt, X_k) to obtain a PRZS $\langle t^k\rangle$, otherwise set $t_i^k = 0$ for all i.
   (b) Each party $P_i \in X_k$ computes, for $j \in \mathcal{J}_k$,
   $$a_i^j \leftarrow M'_j[k]\cdot r_{i,k} + t_i^k = r_{i,k} + t_i^k,$$
   and then sends a_i^j to party $\varphi(j)$ (or retains it if $\varphi(j) = P_i$). (Note that we always have $M'_j[k] = 1$ in this case.)
   (c) For $j \in \mathcal{J}_k$, party $\varphi(j)$ sets $\mathbf{s}_j \leftarrow \sum_{i:P_i\in X_k} a_i^j$
7. This produces a sharing $\mathbf{s}$ under the ESP $(R, M', \varepsilon', \varphi)$ (and hence by definition $(R, M, \varepsilon, \varphi)$).

Figure 5.17: Smart–Wood Protocol Π_{Convert} converting additive shares to shares in the LSSS

It is easy to check in this case that this produces a sharing under the ESP $(R, M, \varepsilon, \varphi)$ of the value x. Thus, we require one execution of $\mathcal{F}_{\text{PRZS}}$ and we need to transfer three ring elements.

Smart-Wood Algorithm:

The algorithm in Figure 5.16 does not need to perform any column operations, however the mapping χ can be defined as $\chi(1) = 2$, $\chi(2) = 3$ and $\chi(3) = 1$. This gives us $\mathsf{im}(\chi) = \{1, 2, 3\}$.

Protocol Π_{Convert} then consists of the following steps, on input of $x = x_1 + x_2 + x_3$, where x_i is held by player P_i.

1. Call $\mathcal{F}_{\text{PRZS}}$ to generate t_i^0 with $\sum_i t_i^0 = 0$.
2. Define $K_1 = \{2\}$, $K_2 = \{3\}$, $K_3 = \{1\}$, we thus have $X_1 = \{P_3\}$, $X_2 = \{P_1\}$ and $X_3 = \{P_2\}$.
3. Set $r_{1,2} \leftarrow x_1 + t_1^0$, $r_{2,3} \leftarrow x_2 + t_2^0$, $r_{3,1} \leftarrow x_3 + t_3^0$, with all other $r_{i,j}$ set to zero.
4. We have $\mathbf{s}_1 = r_{3,1}$, $\mathbf{s}_2 = r_{1,2}$ and $\mathbf{s}_3 = r_{2,3}$, thus,
   (a) Player P_1 needs to send player P_3 the value $\mathbf{s}_2$,
   (b) Player P_2 needs to send player P_1 the value $\mathbf{s}_3$,
   (c) Player P_3 needs to send player P_1 the value $\mathbf{s}_1$.

It is easy to check in this case that this produces a sharing under the ESP $(R, M, \varepsilon, \varphi)$ of the value x. Thus, we require one execution of $\mathcal{F}_{\text{PRZS}}$ and we need to transfer three ring elements. Hence, in this case the two protocols have the same cost, and are basically identical.

5.B.2 Replicated $(5, 2)$ Sharing

This secret sharing method for the threshold structure $(n, t) = (5, 2)$ works for any ring $\mathbb{Z}_{p^k}$, for any size p^k. Here our input ESP $(R, M, \varepsilon, \varphi)$ is given by a matrix of dimension 30×10,

$$M = \begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{pmatrix},$$

$$\varepsilon = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1),$$

$$\varphi(i) = \lceil i/6 \rceil.$$

KRSW Algorithm:

We now discuss the methodology of KRSW, in Figure 5.15, for this example. We define $\chi(1) = 3$, $\chi(2) = 2$, $\chi(3) = 1$, $\chi(4) = 7$, and $\chi(5) = 4$. We also define $\psi(1) = 3$, $\psi(2) = 2$, $\psi(3) = 1$, $\psi(4) = 5$, $\psi(5) = 1$, $\psi(6) = 1$, $\psi(7) = 4$, $\psi(8) = 1$, $\psi(9) = 1$ and $\psi(10) = 1$. Protocol Π_{Convert} then consists of the following steps, on input of $x = x_1 + x_2 + x_3 + x_4 + x_5$, where x_i is held by player P_i.

1. Call $\mathcal{F}_{\text{PRZS}}$ to generate t_i with $\sum_i t_i = 0$.
2. For $k \in \{5, 6, 8, 9, 10\}$ the players in $\mathcal{I}_k$ generate locally the values $\mathbf{s}_j$ for $j \in \mathcal{J}_k$ by using a call to $\mathcal{F}_{\text{AgreeRandom}'}(R, \mathcal{I}_k)$.
3. Player P_1 computes $\mathbf{s}_1 \leftarrow x_1 + t_1 - \mathbf{s}_2 - \mathbf{s}_3 - \mathbf{s}_4 - \mathbf{s}_5 - \mathbf{s}_6$ and sends it to P_4 and P_5 (as $\mathbf{s}_{21}$ and $\mathbf{s}_{27}$).
4. Player P_2 computes $\mathbf{s}_7 \leftarrow x_2 + t_2$ and sends it to P_4 and P_5 (as $\mathbf{s}_{20}$ and $\mathbf{s}_{26}$).
5. Player P_3 computes $\mathbf{s}_{13} \leftarrow x_3 + t_3$ and sends it to P_4 and P_5 (as $\mathbf{s}_{19}$ and $\mathbf{s}_{25}$).
6. Player P_4 computes $\mathbf{s}_{22} \leftarrow x_4 + t_4$ and sends it to P_2 and P_3 (as $\mathbf{s}_{10}$ and $\mathbf{s}_{16}$).
7. Player P_5 computes $\mathbf{s}_{28} \leftarrow x_5 + t_5$ and sends it to P_2 and P_3 (as $\mathbf{s}_8$ and $\mathbf{s}_{14}$).

This requires one call to $\mathcal{F}_{\text{PRZS}}$, and five calls to $\mathcal{F}_{\text{AgreeRandom}'}(R, \mathcal{I}_k)$, and the transfer of ten elements.

Smart-Wood Algorithm:

We now discuss the methodology for this example of Smart and Wood, i.e. Figure 5.16 and Figure 5.17. The algorithm in Figure 5.16 does not need to perform any column operations, however the mapping χ can be defined as $\chi(1) = 3$, $\chi(2) = 2$, $\chi(3) = 1$, $\chi(4) = 7$ and $\chi(5) = 4$. This gives us $\mathsf{im}(\chi) = \{1, 2, 3, 4, 7\}$. This is our first interesting example as the ESP has more columns than the number of parties.

Protocol Π_{Convert} then consists of the following steps, on input of $x = x_1 + x_2 + x_3 + x_4 + x_5$, where x_i is held by player P_i.

1. Call $\mathcal{F}_{\text{PRZS}}$ to generate t_i^0 with $\sum_i t_i^0 = 0$.

2. Define $K_1 = \{3, 5, 6, 8, 9, 10\}$, $K_2 = \{2, 5, 6, 8, 9, 10\}$, $K_3 = \{1, 5, 6, 8, 9, 10\}$, $K_4 = \{5, 6, 7, 8, 9, 10\}$ and $K_5 = \{4, 5, 6, 8, 9, 10\}$ we then have $X_1 = \{P_3\}$, $X_2 = \{P_2\}$, $X_3 = \{P_1\}$, $X_4 = \{P_5\}$, $X_5 = X_6 = X_8 = X_9 = X_{10} = \mathcal{P}$, $X_7 = \{P_4\}$.

3. Player P_i generates $r_{i,k}$ for $k \in K_i$ such that $\sum_k r_{i,k} = x_i + t_i^0$, with all other $r_{i,k}$ set to zero, i.e. they generate $r_{i,k}$ such that

   (a) $r_{1,3} + r_{1,5} + r_{1,6} + r_{1,8} + r_{1,9} + r_{1,10} = x_1 + t_1^0$.
   (b) $r_{2,2} + r_{2,5} + r_{2,6} + r_{2,8} + r_{2,9} + r_{2,10} = x_2 + t_2^0$.
   (c) $r_{3,1} + r_{3,5} + r_{3,6} + r_{3,8} + r_{3,9} + r_{3,10} = x_3 + t_3^0$.
   (d) $r_{4,5} + r_{4,6} + r_{4,7} + r_{4,8} + r_{4,9} + r_{4,10} = x_4 + t_4^0$.
   (e) $r_{5,4} + r_{5,5} + r_{5,6} + r_{5,8} + r_{5,9} + r_{5,10} = x_5 + t_5^0$.

4. For $k \in \{1, 2, 3, 4, 7\}$ and $P_i \in X_k$, player P_i sends $r_{i,k}$ to player $\varphi(j)$ if the jth row is the basis vector $\mathbf{e}_k$, i.e.

   (a) Player P_1 needs to send players P_4 and P_5 the value $r_{1,3}$,
   (b) Player P_2 needs to send players P_4 and P_5 the value $r_{2,2}$,
   (c) Player P_3 needs to send players P_4 and P_5 the value $r_{3,1}$,
   (d) Player P_4 needs to send players P_2 and P_3 the value $r_{4,7}$,
   (e) Player P_5 needs to send players P_2 and P_3 the value $r_{5,4}$,

   The players set $\mathbf{s}_j = r_{\star,j}$ as appropriate. This step therefore requires sending 10 elements in total.

5. For $k \in \{5, 6, 8, 9, 10\}$ the players execute a PRZS on the set $\mathcal{P}$ to generate t_i^k for $i = 1, \ldots, 5$. The value $r_{i,k} + t_i^k$ is sent by player i to player $\varphi(j)$ if the jth row is the basis vector $\mathbf{e}_k$. For all $j \in \mathcal{J}_k$, player $\varphi(j)$ computes $\mathbf{s}_j$ as the sum of all values received. This step requires (in total) P_1 to send 10 elements, P_2 and P_3 a total of 12 elements, and P_4 and P_5 a total of 13 elements. This in total 60 elements.

It is easy to check in this case that this produces a sharing under the ESP $(R, M, \varepsilon, \varphi)$ of the value x. Thus, we require six executions of $\mathcal{F}_{\text{PRZS}}$ and we need to transfer 70 elements. Thus, in this case Smart-Wood is much less efficient than KRSW.

5.B.3 Replicated $(10, 4)$ Sharing

This one is a bit big to write out, but the basic methodology for replicated is the same (the underlying matrix has 252 columns and 1512 rows). The methodology of Smart-Wood will be highly inefficient in this case, however the optimized version of KRSW will produce a protocol Π_{Convert} which requires the transmission of $n \cdot (n - t - 1) = 50$ elements, and the execution of one $\mathcal{F}_{\text{PRZS}}$, plus $(252 - 10) = 242$ calls to $\mathcal{F}_{\text{AgreeRandom}'}(R, \mathcal{I}_k)$.

5.B.4 Shamir $(3, 1)$ for large p

This secret sharing method for the threshold structure $(n, t) = (3, 1)$ works for any ring $\mathbb{Z}_{p^k}$ for which $p > 4$. Here our input ESP $(R, M, \varepsilon, \varphi)$ is given by

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix},$$

$$\varepsilon = (1, 0),$$

$$\varphi(i) = i.$$

KRSW Algorithm:

This method cannot be applied as the underlying ESP does not correspond to replicated secret sharing.

Smart-Wood Algorithm:

After the column operations, from the algorithm in Figure 5.16, our new ESP $(R, M', \varepsilon', \varphi)$ becomes

$$M' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 2 \end{pmatrix},$$

$$\varepsilon' = (2, -1),$$

$$\varphi(i) = i,$$

and we assign $\chi(1) = 1$, $\chi(2) = 2$, $\chi(3) = 2$. Protocol Π_{Convert} then consists of the following steps, on input of $x = x_1 + x_2 + x_3$, where x_i is held by player P_i.

1. Call $\mathcal{F}_{\text{PRZS}}$ to generate t_i^0 with $\sum_i t_i^0 = 0$.
2. Define $K_1 = \{1\}$, $K_2 = \{2\}$, $K_3 = \{2\}$, and then we have $X_1 = \{P_1\}$ and $X_2 = \{P_2, P_3\}$.
3. Set $r_{1,1} \leftarrow (x_1 + t_1^0)/2$, $r_{2,2} \leftarrow -x_2 - t_2^0$, $r_{3,2} \leftarrow -x_3 - t_3^0$, with all other $r_{i,j}$ set to zero.
4. The parties call $\mathcal{F}_{\text{PRZS}}$ to generate t_i^3 with $\sum_i t_i^3 = 0$.

   (a) Player P_1 computes $a_1^3 = -1 \cdot r_{1,1} + t_1^3$ and sends it to player P_3.
   (b) Player P_2 computes $a_2^3 = 2 \cdot r_{2,2} + t_2^3$ and sends it to player P_3.
   (c) Player P_3 computes $a_3^3 = 2 \cdot r_{3,2} + t_3^3$ and retains it.

5. Party P_3 computes $\mathbf{s}_3 = a_1^3 + a_2^3 + a_3^3 = -r_{1,1} + 2 \cdot r_{2,2} + 2 \cdot r_{3,2}$.
6. Player one sets $\mathbf{s}_1 = a_1^1 = r_{1,1}$. Player P_2 sets $a_2^2 = r_{2,2}$, and player P_3 sets $a_3^2 = r_{3,2}$ and sends it to party P_2. Party P_2 sets $\mathbf{s}_2 = a_2^2 + a_3^2 = r_{2,2} + r_{3,2}$.

This requires two executions of $\mathcal{F}_{\text{PRZS}}$ and the transmission of three ring elements. To see that the sharing is correct under the original ESP $(R, M, \varepsilon, \varphi)$, we note that the shares of the three players are:

$$\mathbf{s}_1 = r_{1,1} = (x_1 + t_1^0)/2,$$

$$\mathbf{s}_2 = r_{2,2} + r_{3,1} = -x_2 - t_2^0 - x_3 - t_3^0,$$

$$\mathbf{s}_3 = -r_{1,1} + 2 \cdot r_{2,2} + 2 \cdot r_{3,2}.$$

The value shared is equal to

$$\begin{aligned} 2 \cdot \mathbf{s}_1 - \mathbf{s}_2 &= x_1 + t_1^0 + x_2 + t_2^0 + x_3 + t_3^0 \\ &= x_1 + x_2 + x_3 = x. \end{aligned}$$

The sharing is valid if $\mathbf{s}_3 - 2 \cdot \mathbf{s}_2 + \mathbf{s}_1 = 0$, thus we see it is valid as we have

$$\begin{aligned} \mathbf{s}_3 - 2 \cdot \mathbf{s}_2 + \mathbf{s}_1 &= (-r_{1,1} + 2 \cdot r_{2,2} + 2 \cdot r_{3,2}) - 2 \cdot (r_{2,2} + r_{3,2}) + r_{1,1} \\ &= 0. \end{aligned}$$

5.B.5 Shamir $(5,2)$ for large p

Shamir secret sharing for large p can be defined using the Vandermonde-matrix of the appropriate size. Note that this works for any ring $\mathbb{Z}_{p^k}$, where $p > 6$. The initial input for the ESP, $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \phi)$, is given by

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \end{pmatrix}$$

$$\varepsilon = (1,0,0)$$

$$\varphi(i) = i$$

KRSW Algorithm:

This method cannot be applied as the underlying ESP does not correspond to replicated secret sharing.

Smart-Wood Algorithm:

Doing the column operations as required we obtain the altered ESP, $M' = (\mathbb{Z}_{p^k}, M', \varepsilon', \varphi)$, as follows:

$$M' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -3 & 3 \\ 3 & -8 & 6 \end{pmatrix}$$

$$\varepsilon = (3,-3,1)$$

$$\varphi(i) = i$$

This allows us to define χ following Figure 5.16 as

$$\chi(1) = 1, \chi(2) = 2, \chi(3) = 3, \chi(4) = 1, \chi(5) = 3,$$

where $\chi(4)$ and $\chi(5)$ are freely chosen, so could be changed for any of the other columns. The protocol Π_{Convert} from Figure 5.17 then calls for the following steps upon input of $x = x_1 + x_2 + x_3 + x_4 + x_5$, where x_i is held by player P_i.

1. Call $\mathcal{F}_{\text{PRZS}}$ to generate t^0 with $\langle t^0\rangle_i = t_i^0$ such that $\sum_i t_i^0 = 0$.
2. Note that $\mathcal{J}_1 = \{1\}, \mathcal{J}_2 = \{2\}, \mathcal{J}_3 = \{3\}$ and define K_i and X_i for each player as follows:
$$K_1 = \{1\}, K_2 = \{2\}, K_3 = \{3\}, K_4 = \{1\}, K_5 = \{3\}$$
$$X_1 = \{P_1, P_4\}, X_2 = \{P_2\}, X_3 = \{P_3, P_5\}$$
3. Set $r_{1,1} = (x_1 + t_1^0)/3, r_{2,2} = (-x_2 - t_2^0)/3, r_{3,3} = x_3 + t_3^0, r_{4,1} = (x_4 + t_4^0)/3, r_{5,3} = x_5 + t_5^0$ and set all other $r_{i,k} = 0$.
4. Call $\mathcal{F}_{\text{PRZS}}$ to generate $\langle t^4\rangle$ and $\langle t^5\rangle$.
   (a) P_1 generates $a_1^4 = r_{1,1} + t_1^4$ and $a_1^5 = 3 \cdot r_{1,1} + t_1^5$. Then P_1 sends a_1^4 to P_4 and a_1^5 to P_5.
   (b) P_2 generates $a_2^4 = -3 \cdot r_{2,2} + t_2^4$ and $a_2^5 = -8 \cdot r_{2,2} + t_2^5$. Then P_2 sends a_2^4 to P_4 and a_2^5 to P_5.
   (c) P_3 generates $a_3^4 = 3 \cdot r_{3,3} + t_3^4$ and $a_3^5 = 6 \cdot r_{3,3} + t_3^5$. Then P_3 sends a_3^4 to P_4 and a_3^5 to P_5.
   (d) P_4 generates $a_4^4 = r_{4,1} + t_4^4$ and $a_4^5 = 3 \cdot r_{4,1} + t_4^5$. Then P_4 sends a_4^5 to P_5 and upon reception of all a_i^4's computes $\mathbf{s}_4$.
   (e) P_5 generates $a_5^4 = 3 \cdot r_{5,3} + t_5^4$ and $a_5^5 = 6 \cdot r_{5,3} + t_5^5$. Then P_5 sends a_5^4 to P_4 and upon reception of all a_i^5's computes $\mathbf{s}_5$.
5. Note that $|X_k| \leq 2$ so $t_i^k = 0$ for all $k \in \{1, 2, 3\}$ and all $i \in \{1, 2, 3, 4, 5\}$. Now generate the other a_i^j:
   (a) For X_1, P_1 generates $a_1^1 = r_{1,1}$ and P_4 generates $a_4^1 = r_{4,1}$. P_4 then sends a_4^1 to P_1 who computes $\mathbf{s}_1 = a_1^1 + a_4^1$.
   (b) For X_2, P_2 generates $a_2^2 = r_{2,2}$. P_2 retains a_2^2 and computes $\mathbf{s}_1 = a_2^2$.
   (c) For X_3, P_3 generates $a_3^3 = r_{3,3}$ and P_5 generates $a_5^3 = r_{5,3}$. P_5 then sends a_5^3 to P_3 who computes $\mathbf{s}_3 = a_3^3 + a_5^3$.

This requires a total of 3 $\mathcal{F}_{\text{PRZS}}$ executions and the transmission of ten ring elements. All that is left to do is to show that this is indeed a correct sharing under $\mathcal{M}'$. Note that the shares, in full, are

$$\begin{aligned}
\mathbf{s}_1 =& a_1^1 + a_4^1 = r_{1,1} + r_{4,1},\\
\mathbf{s}_2 =& a_2^2 = r_{2,2},\\
\mathbf{s}_3 =& a_3^3 + a_5^3 = r_{3,3} + r_{5,3},\\
\mathbf{s}_4 =& \sum_i a_i^4 = r_{1,1} - 3 \cdot r_{2,2} + 3 \cdot r_{3,3} + r_{4,1} + 3 \cdot r_{5,3},\\
\mathbf{s}_5 =& \sum_i a_i^5 = 3 \cdot r_{1,1} - 8 \cdot r_{2,2} + 6 \cdot r_{3,3} + 3 \cdot r_{4,1} + 6 \cdot r_{5,3}.
\end{aligned}$$

The shared value is given by

$$\begin{aligned}
3 \cdot \mathbf{s}_1 - 3 \cdot \mathbf{s}_2 + \mathbf{s}_3 &= 3 \cdot (r_{1,1} + r_{4,1}) - 3 \cdot r_{2,2} + r_{3,3} + r_{5,3}\\
&= x_1 + t_1^0 + x_4 + t_4^0 + x_2 + t_1^2 + x_3 + t_3^0 + x_5 + t_5^0 = \sum_i x_i = x
\end{aligned}$$

and is hence correct. To verify that the sharing is valid we only have to show that $\mathbf{s}_4 - \mathbf{s}_1 + 3 \cdot \mathbf{s}_2 - 3 \cdot \mathbf{s}_3 = 0$ and that $\mathbf{s}_5 - 3 \cdot \mathbf{s}_1 + 8 \cdot \mathbf{s}_2 - 6 \cdot \mathbf{s}_3 = 0$. This is immediate as

$$\begin{aligned}
\mathbf{s}_4 - \mathbf{s}_1 + 3 \cdot \mathbf{s}_2 - 3 \cdot \mathbf{s}_3 &= (r_{1,1} - 3 \cdot r_{2,2} + 3 \cdot r_{3,3} + r_{4,1} + 3 \cdot r_{5,3})\\
&\quad - (r_{1,1} + r_{4,1}) + 3 \cdot r_{2,2} - 3 \cdot (r_{3,3} + r_{5,3}) = 0,\\
\mathbf{s}_5 - 3 \cdot \mathbf{s}_1 + 8 \cdot \mathbf{s}_2 - 6 \cdot \mathbf{s}_3 &= (3 \cdot r_{1,1} - 8 \cdot r_{2,2} + 6 \cdot r_{3,3} + 3 \cdot r_{4,1} + 6 \cdot r_{5,3})\\
&\quad - 3 \cdot (r_{1,1} + r_{4,1}) + 8 \cdot r_{2,2} - 6 \cdot (r_{3,3} + r_{5,3}) = 0.
\end{aligned}$$

5.B.6 Shamir $(10, 4)$ for large p

As before, for Shamir secret sharing with a large enough p, we obtain an ESP with a Vandermonde matrix. This particular ESP requires $p > 11$, and is defined by the following values $(\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$:

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \\ 1 & 6 & 36 & 216 & 1296 \\ 1 & 7 & 49 & 343 & 2401 \\ 1 & 8 & 64 & 512 & 4096 \\ 1 & 9 & 81 & 729 & 6561 \\ 1 & 10 & 100 & 1000 & 10000 \end{pmatrix}$$

$$\varepsilon = (1, 0, 0, 0, 0)$$

$$\varphi(i) = i$$

KRSW Algorithm:

This method can not be applied as the underlying ESP does not correspond to replicated secret sharing.

Smart-Wood Algorithm:

After performing column operations, we obtain the ESP over $\mathbb{Z}_{p^k}$ defined by

$$M' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & -5 & 10 & -10 & 5 \\ 5 & -24 & 45 & -40 & 15 \\ 15 & -70 & 126 & -105 & 35 \\ 35 & -160 & 280 & -224 & 70 \\ 70 & -315 & 540 & -420 & 126 \end{pmatrix}$$

$$\varepsilon' = (5, -10, 10, -5, 1)$$

$$\varphi(i) = i$$

As $\varepsilon'_k \neq 0$ for all k, the mapping χ can be arbitrarily chosen for players 6 through 10. In this example, we will choose $\chi(i) = i, 1 \leq i \leq 5$ and $\chi(i) = 5, 6 \leq i \leq 10$.

We now trace through the steps taken in Figure 5.17 to determine the communication cost of the conversion protocol from a full threshold additive sharing onto our ESP. The parties hold $x = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10}$.

1. All parties obtain a PRZS $\langle t^0 \rangle$
2. The values K_i are defined as $K_i = \{\chi(i)\}$, as $\operatorname{Im}\chi = [d]$, so every player splits $x_i + t_i^0 = x_{i,\chi(i)}$
3. From these $x_{i,\chi(i)}$, we obtain the values $r_{i,\chi(i)}$, namely:
   - $r_{1,1} = (x_1 + t_1^0)/5$
   - $r_{2,2} = (-x_2 - t_2^0)/10$
   - $r_{3,3} = (x_3 + t_3^0)/10$
   - $r_{4,4} = (-x_4 - t_4^0)/5$
   - $r_{5,5} = x_5 + t_5^0$
   - $r_{6,5} = x_6 + t_6^0$
   - $r_{7,5} = x_7 + t_7^0$
   - $r_{8,5} = x_8 + t_8^0$

- $r_{9,5} = x_9 + t_9^0$
- $r_{10,5} = x_{10} + t_{10}^0$

4. Since ε' has no zero entries, nothing happens in this step
5. (a) The parties generate the PRZSs $\langle t^j \rangle, j = 6 \dots 10$

   (b) party P_6 receives:
   - from P_1 the value $(x_1 + t_1^0)/5 + t_1^6$
   - from P_2 the value $(x_2 + t_2^0)/2 + t_2^6$
   - from P_3 the value $x_3 + t_3^0 + t_3^6$
   - from P_4 the value $2 \cdot (x_4 + t_4^0) + t_4^6$
   - from P_i for $i = 5, 7, 8, 9, 10$ the value $5 \cdot (x_i + t_i^0) + t_i^6$

   and sums it to obtain $\mathbf{s}_6$

   (c) Similarly, players $P_j, 7 \leq j \leq 10$ each also receive 9 ring elements, of the form $\alpha^j_{\chi(i)} \cdot (x_i + t_i^0) + t_i^j$, with $\alpha_i^j = M_j[i]/\varepsilon_i'$:
   - $\alpha^7 = (1, 12/5, 9/2, 8, 15)$
   - $\alpha^8 = (3, 7, 63/5, 21, 35)$
   - $\alpha^9 = (7, 16, 28, 224/5, 70)$
   - $\alpha^{10} = (14, 63/2, 54, 84, 126)$

   which they also sum to obtain $\mathbf{s}_j$
6. Note the sets $\mathcal{J}_k = \{k\}, 1 \leq k \leq 5$, $X_k = \{P_k\}, 1 \leq k \leq 4$ and $X_5 = \{P_i \mid 5 \leq i \leq 10\}$.

   (a) Parties $P_5, \dots, P_{10}$ obtain the PRZS $\langle t^5 \rangle$.

   (b) The same parties P_i then compute $a_i^5 = x_i + t_i^0 + t_i^5$ and send it to P_5. Hence, this costs 5 ring elements of communication.

   Further communication is not needed.

We see that execution of this conversion protocol costs a total of 7 executions of $\mathcal{F}_{\mathsf{PRZS}}$, which could be brought down to 6 by assigning $\chi(i) = i - 5$ for $6 \leq i \leq 10$ as that removes the need for a PRZS in step 6. A total of 50 ring elements need to be communicated.

At the end of this process, the players P_i hold the following shares $\mathbf{s}_i$:

$$\mathbf{s}_1 = (x_1 + t_1^0)/5$$

$$\mathbf{s}_2 = (-x2 - t_2^0)/10$$

$$\mathbf{s}_3 = (x_3 + t_3^0)/10$$

$$\mathbf{s}_4 = (-x_4 - t_4^0)/5$$

$$\mathbf{s}_5 = \sum_{5 \le i \le 10} x_i + t_i^0 + t_i^5$$

$$\mathbf{s}_6 = (x_1 + t_1^0)/5 + (x_2 + t_2^0)/2 + x_3 + t_3^0 + 2 \cdot (x_4 + t_4^0) + \sum_{5 \le i \le 10} 5 \cdot (x_i + t_i^0)$$

$$\mathbf{s}_{j=7,\dots,10} = \alpha_1^j \cdot (x_1 + t_1^0) + \alpha_2^j \cdot (x_2 + t_2^0) + \alpha_3^j \cdot (x_3 + t_3^0) + \alpha_4^j \cdot (x_4 + t_4^0) + \sum_{5 \le i \le 10} \alpha_5^j \cdot (x_i + t_i^0)$$

It can then be verified that $\langle \varepsilon', (\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3, \mathbf{s}_4, \mathbf{s}_5) \rangle$ is equal to the shared secret x, and the underlying parity check matrix is satisfied by the resulting share vector.

5.B.7 Shamir $(3,1)$ for $\mathbb{Z}_{2^k}$

For the case of Shamir over $\mathbb{Z}_{2^k}$ we refer back to the example in Section 5.2.5, where it can be seen how the matrix M is derived. The rest of the ESP, $\mathcal{M} = (\mathbb{Z}_{2^k}, M, \epsilon, \varphi)$ is defined as follows:

$$M = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\varepsilon = (1, 0, 0)$$

$$\varphi(i) = \lceil i/2 \rceil$$

KRSW Algorithm:

This method can not be applied as the underlying ESP does not correspond to replicated secret sharing.

Smart-Wood Algorithm:

The first step is to do column operations to obtain the new ESP $\mathcal{M}' = (\mathbb{Z}_{2^k}, M', \varepsilon', \varphi)$:

$$M' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & -1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\varepsilon' = (1, -1, 0)$$

and φ as previously defined. It is easily checked that the access structure stays the same and this will be left to the reader. The map χ is then defined as follows $\chi(1) = 1, \chi(2) = 2, \chi(3) = 2$. Note that this means that $\text{Im}(\varphi) = \{1, 2\}$ and therefore not surjective. Now Π_{Convert} comes into action once more on input of $x = x_1 + x_2 + x_3$, where x_i is held by P_i.

1. Call $\mathcal{F}_{\text{PRZS}}$ to generate t_i^0 with $\sum_i t_i^0 = 0$.
2. Define the K_i, X_i and $\mathcal{J}_i$ as follows:
   (a) $K_1 = \{1,3\}, K_2 = \{2,3\}, K_3 = \{2,3\}$.
   (b) $X_1 = \{P_1\}, X_2 = \{P_2, P_3\}, X_3 = \{P_1, P_2, P_3\}$
   (c) $\mathcal{J}_1 = 1, \mathcal{J}_2 = 6, \mathcal{J}_3 = 2$
3. Define $r_{1,1} = x_{1_1} + t_1^0, r_{2,2} = -x_{2,2} - t_2^0, r_{3,2} = -x_{3,2} - t_3^0$ and $r_{i,3} = \mathsf{Rand}(R)$ for all i. Let all other $r_{i,j} = 0$.
4. For $j \in \{3,4,5\}$ run $\mathcal{F}_{\text{PRZS}}$ to generate $\langle t^j \rangle$ with shares denoted t_i^j and $\sum_i t_i^j$. Then
   (a) P_1 computes $a_1^3 = r_{1,1} + t_1^3$, $a_1^4 = t_1^4$, $a_1^5 = r_{1,1} + t_1^5$ and sends a_1^3, a_1^4 to P_2 and a_1^5 to P_3.
   (b) P_2 computes $a_2^3 = -r_{2,2} + t_2^3$, $a_2^4 = r_{2,2} + t_2^4$, and $a_2^5 = t_1^5$ and sends a_2^5 to P_3, then P_2 computes $\mathbf{s}_3 = \sum_i a_i^3$ and $\mathbf{s}_4 = \sum_i a_i^4$.
   (c) P_3 computes $a_3^3 = -r_{3,2} + t_3^3$, $a_3^4 = r_{3,2} + t_3^4$, and $a_3^5 = t_3^5$, and sends a_3^3 and a_3^4 to P_2, then P_3 computes $\mathbf{s}_5 = \sum_i a_i^5$.
5. Note that $|X_3| > 2$, hence run $\mathcal{F}_{\text{PRZS}}$ to generate $\langle t^2 \rangle$ with corresponding shares t_i^2 and set $t^1 = t^6 = 0$. Then
   (a) For X_1, P_1 computes $a_1^1 = r_{1,1}$ and retains a_1^1. Then P_1 sets $\mathbf{s}_1 = a_1^1$.
   (b) For X_2, P_2 computes $a_2^6 = r_{2,2}$ and P_3 computes $a_3^6 = r_{3,2}$ and P_2 sends a_2^6 to P_3. Then P_3 sets $\mathbf{s}_6 = a_2^6 + a_3^6$.
   (c) For X_3, P_1 computes $a_1^2 = r_{1,3} + t_1^2$, P_2 computes $a_2^2 = r_{2,3} + t_2^2$, and P_3 computes $a_3^2 = r_{3,3} + t_3^2$ Then P_1 and P_3 send a_1^2 and a_3^2 to P_2 respectively. Then P_2 sets $\mathbf{s}_2 = a_1^2 + a_2^2 + a_3^2$.

Which concludes the Π_{Convert} protocol. Summarizing we can see that there are two calls to $\mathcal{F}_{\text{PRZS}}$, while six ring elements are communicated in step 4 and three ring elements are communicated in step 5. This leads to a total of nine sent elements.

This produces a sharing with shares

$$\begin{aligned}
\mathbf{s}_1 =& a_1^1 = r_{1,1} = x_1 + t_1^0, \\
\mathbf{s}_2 =& a_1^2 + a_2^2 + a_3^2 = r_{1,3} + r_{2,3} + r_{3,3} = \mathsf{Rand}_1(R) + \mathsf{Rand}_2(R) + \mathsf{Rand}_3(R), \\
\mathbf{s}_3 =& a_1^3 + a_2^3 + a_3^3 = r_{1,1} - r_{2,2} - r_{3,2} = x_1 + x_2 + x_3, \\
\mathbf{s}_4 =& a_1^4 + a_2^4 + a_3^4 = r_{2,2} + r_{3,2} = -x_2 - t_2^0 - x_3 - t_3^0, \\
\mathbf{s}_5 =& a_1^5 + a_2^5 + a_3^5 = r_{1,1} = x_1 + t_1^0 - x_3 - t_3^0, \\
\mathbf{s}_6 =& a_2^6 + a_3^6 = r_{2,2} + r_{3,2} = -x_2 - t_0^2 - x_3 - t_3^0.
\end{aligned}$$

Then to check that this is a correct we first check the shared value, which is correct:

$$\mathbf{s}_1 - \mathbf{s}_6 = x_1 + t_1^0 + x_2 + t_0^2 + x_3 + t_3^0 = \sum_i x_i = x$$

Then we show that the sharings are correct, by verifying that the parity check matrix (of the original ESP $\mathcal{M}$) when applied to this share vector results in the zero vector,

$$\mathbf{s}_3 - \mathbf{s}_1 - \mathbf{s}_6 = r_{1,1} - r_{2,2} - r_{3,2} - r_{1,1} + r_{2,2} + r_{3,3} = 0$$

$$\mathbf{s}_4 - \mathbf{s}_6 = 0$$

$$\mathbf{s}_5 - \mathbf{s}1 = 0$$

5.B.8 Shamir $(5, 2)$ for $\mathbb{Z}_{2^k}$

We construct the original matrix M similarly to the example in section 5.2.5, where we now need to work over an extension of degree 3. The ESP $(\mathbb{Z}_{2^k}, M, \varepsilon, \varphi)$ then becomes

$$M = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

$$\varepsilon = (1, 0, 0, 0, 0, 0, 0)$$

$$\varphi(i) = \lceil i/3 \rceil$$

KRSW Algorithm:

This method can not be applied as the underlying ESP does not correspond to replicated secret sharing.

Smart-Wood Algorithm:

As must be familiar by now, we start with the column reduction of M to M' with corresponding ε'. Note that the sharing of x is given by $x = x_1 + x_2 + x_3 + x_4 + x_5$

$$M' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 3 & 4 & -2 & -1 & -2 & 0 \\ 0 & 2 & 3 & -2 & -2 & -1 & 2 \\ 1 & 0 & -1 & 2 & 1 & 1 & -2 \\ -1 & 1 & 1 & -1 & -1 & 0 & 2 \\ -1 & -1 & -1 & 0 & 1 & 1 & 1 \\ 1 & 0 & -1 & 1 & 1 & 1 & -1 \\ -1 & 0 & 1 & -1 & -1 & 0 & 2 \\ -1 & -1 & -2 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$\varepsilon = (1, 0, 0, 1, 0, 0, -1)$$

Having obtained this we can define $\chi : [n] \mapsto [d]$ by $\chi(1) = 1$, $\chi(2) = 4$, $\chi(3) = 7$, $\chi(4) = 1$, $\chi(5) = 4$. From here we call Π_{Convert} doing the steps as follows:

1. Call $\mathcal{F}_{\text{PRZS}}$ to obtain sharing $\langle t^0 \rangle$ such that $\sum_i t_i^0 = 0$.
2. Define X_i and $\mathcal{J}_i$ as follows:
   (a) $K_1 = K_4 = \{1, 2, 3, 5, 6\}, K_2 = K_5 = \{2, 3, 4, 5, 6\}, K_3 = \{2, 3, 5, 6, 7\}$
   (b) $X_1 = \{P_1, P_4\}, X_4 = \{P_2, P_5\}, X_7 = \{P_3\}, X_2 = X_3 = X_5 = X_6 = \{\mathcal{P}\}$
   (c) $\mathcal{J}_i = i$ for $i \in \{1, \ldots, 7\}$.
3. Define $r_{1,1} = x_{1,1} + t_1^0$, $r_{2,4} = x_{2,4} + t_2^0$, $r_{3,7} = -x_{3,7} - t_3^0$, $r_{4,1} = x_{4,1} + t_4^0, r_{5,4} = x_{5,4} + t_5^0$. Let $r_{i,2}, r_{i,3}, r_{i,5}, r_{i,6} \leftarrow R$ for all $i \in [n]$ and let all other $r_{i,j} = 0$.
4. For $j \in \{8, \ldots, 15\}$ run $\mathcal{F}_{\text{PRZS}}$ to generate $\langle t^j \rangle$ with shares denoted t_i^j and $\sum_i t_i^j = 0$. Then

(a) P_1 computes $a_1^8 = 2 \cdot r_{1,1} + t_1^8$,$a_1^9 = t_1^9$, $a_1^{10} = r_{1,1} + t_1^{10}$, $a_1^{11} = -r_{1,1} + t_1^{11}$, $a_1^{12} = -r_{1,1} + t_1^{12}$, $a_1^{13} = r_{1,1} + t_1^{13}$, $a_1^{14} = -r_{1,1} + t_1^{14}$, $a_1^{15} = -r_{1,1} + t_1^{15}$. P_1 then sends a_1^8 and a_1^9 to P_3, a_1^{10}, a_1^{11}, and a_1^{12} to P_4, and $a_1^{13}, a_1^{14}, a_1^{15}$ to P_5.

(b) P_2 computes $a_2^8 = -2 \cdot r_{2,4} + t_2^8$, $a_2^9 = -2 \cdot r_{2,4} + t_2^9$, $a_2^{10} = 2 \cdot r_{2,4} + t_2^{10}$, $a_2^{11} = -r_{2,4} + t_2^{11}$, $a_2^{12} = t_2^{12}$ $a_2^{13} = r_{2,4} + t_2^{13}$, $a_2^{14} = -r_{2,4} + t_2^14$, $a_2^{15} = t_2^{15}$. P_2 then sends a_2^8 and a_2^9 to P_3, a_2^{10}, a_2^{11}, and a_2^{12} to P_4, and $a_2^{13}, a_2^{14}, a_2^{15}$ to P_5.

(c) P_3 computes $a_3^8 = t_3^8$, $a_3^9 = 2 \cdot r_{3,7} + t_3^9$, $a_3^{10} = -2 \cdot r_{3,7} + t_3^{10}$, $a_3^{11} = 2 \cdot r_{3,7} + t_3^11$, $a_3^{12} = r_{3,7} + t_3^12$, $a_3^{13} = -r_{3,7} + t_3^{13}$, $a_3^{14} = 2 \cdot r_{3,7} + t_3^{14}$, and $a_3^{15} = r_{3,7} + t_3^{14}$. P_3 then sends a_1^{10}, a_1^{11}, and a_1^{12} to P_4, and $a_1^{13}, a_1^{14}, a_1^{15}$ to P_5. Upon receipt of all a_i^8, a_i^9, P_3 computes $\mathbf{s}_8$ and $\mathbf{s}_9$.

(d) P_4 computes $a_4^8 = 2 \cdot r_{4,1} + t_4^8$, $a_4^9 = t_4^9$, $a_4^{10} = r_{4,1} + t_4^{10}$, $a_4^{11} = -r_{4,1} + t_4^{11}$, $a_4^{12} = -r_{4,1} + t_4^{12}$, $a_4^{13} = r_{4,1} + t_4^{13}$, $a_4^{14} = -r_{4,1} + t_4^{14}$, $a_4^{15} = -r_{4,1} + t_4^{15}$. P_4 then sends a_4^8 and a_4^9 to P_3 and $a_4^{13}, a_4^{14}, a_4^{15}$ to P_5. Upon receipt of all a_i^{10}, a_i^{11}, and a_i^{12}, P_4 computes $\mathbf{s}_{10}, \mathbf{s}_{11}$, and $\mathbf{s}_{12}$.

(e) P_5 computes $a_5^8 = -2 \cdot r_{5,4} + t_5^8$, $a_5^9 = -2 \cdot r_{5,4} + t_5^9$, $a_5^{10} = 2 \cdot r_{5,4} + t_5^{10}$, $a_5^{11} = -r_{5,4} + t_5^{11}$, $a_5^{12} = t_5^{12}$ $a_5^{13} = r_{5,4} + t_5^{13}$, $a_5^{14} = -r_{5,4} + t_5^14$, $a_5^{15} = t_5^{15}$. P_5 then sends a_5^8, a_5^9 to P_3 and a_5^{10}, a_5^{11}, and a_5^{12} to P_4. Upon receipt of all $a_i^{13}, a_i^{14}, a_i^{15}$, P_5 computes $\mathbf{s}_{13}$, $\mathbf{s}_{14}$, and $\mathbf{s}_{15}$.

5. Note that $|X_2| = |X_3| = |X_5| = |X_6| = |\mathcal{P}| > 2$ hence we run $\mathcal{F}_{\text{PRZS}}$ to obtain four sharings $\langle t^2 \rangle, \langle t^3 \rangle, \langle t^5 \rangle, \langle t^6 \rangle$. While $t_i^j = 0$ for $j \in \{1, 4, 7\}$ Then

   (a) For X_1, P_1 computes $a_1^1 = r_{1,1}$, and P_4 computes $a_4^1 = r_{4,1}$. P_1 then retains a_1^1 and receives a_4^1 from P_4. Upon receipt of all a_1^4 P_1 computes $\mathbf{s}_1 = a_1^1 + a_1^4$.

   (b) For X_2, Each P_i computes $a_i^2 = r_{i,2} + t_i^2$ and sends a_i^2 to P_1 which then computes $\mathbf{s}_2 = \sum_i a_i^2$.

   (c) For X_3, Each P_i computes $a_i^3 = r_{i,3} + t_i^3$ and sends a_i^3 to P_1 which then computes $\mathbf{s}_3 = \sum_i a_i^3$.

   (d) For X_4, P_2 computes $a_2^4 = r_{2,4}$ and P_5 computes $a_5^4 = r_{5,4}$. P_2 retains a_2^4 and upon receipt of a_5^4 from P_5 computes $\mathbf{s}_4 = a_2^4 + a_5^4$.

   (e) For X_5, Each P_i computes $a_i^5 = r_{i,5} + t_i^5$ and sends a_i^5 to P_2 which then computes $\mathbf{s}_5 = \sum_i a_i^5$.

   (f) For X_6, Each P_i computes $a_i^6 = r_{i,6} + t_i^6$ and sends a_i^6 to P_2 which then computes $\mathbf{s}_6 = \sum_i a_i^3$.

(g) For X_7, P_3 computes $a_3^7 = r_{3,7}$ and sets $\mathbf{s}_3 = r_{3,7}$.

This means that we, in total, obtain 5 calls to $\mathcal{F}_{\text{PRZS}}$ and 52 ring elements communicated. All that remains is that we show that the sharings are valid.

We approach this by first computing the shares

$$\mathbf{s}_1 = a_1^1 + a_1^4 = r_{1,1} + r_{4,1} = x_{1,1} + t_1^0 + x_{4,1} + t_4^0$$

$$\mathbf{s}_2 = \sum_i a_i^2 = \sum_i r_{i,2} \in R$$

$$\mathbf{s}_3 = \sum_i a_i^3 = \sum_i r_{i,3} \in R$$

$$\mathbf{s}_4 = a_2^4 + a_5^4 = r_{2,4} + r_{5,4} = x_{2,4} + t_2^0 + x_{5,4} + t_5^0$$

$$\mathbf{s}_5 = \sum_i a_i^5 = \sum_i r_{i,5} \in R$$

$$\mathbf{s}_6 = \sum_i a_i^6 = \sum_i r_{i,6} \in R$$

$$\mathbf{s}_7 = a_3^7 = r_{3,7} = -x_{3,7} - t_3^0$$

$$\mathbf{s}_8 = \sum_i a_i^8 = 2 \cdot r_{1,1} - 2 \cdot r_{2,4} + 2 \cdot r_{4,1} - 2 \cdot r_{5,4}$$

$$\mathbf{s}_9 = \sum_i a_i^9 = -2 \cdot r_{2,4} + 2 \cdot r_{3,7} - 2 \cdot r_{5,4}$$

$$\mathbf{s}_{10} = \sum_i a_i^{10} = r_{1,1} + 2 \cdot r_{2,4} - 2 \cdot r_{3,7} + r_{4,1} + 2 \cdot r_{5,4}$$

$$\mathbf{s}_{11} = \sum_i a_i^{11} = -r_{1,1} - r2,4 + 2 \cdot r_{3,7} - r_{4,1} - r5,4$$

$$\mathbf{s}_{12} = \sum_i a_i^{12} = -r_{1,1} + r_{3,7} - r_{4,1}$$

$$\mathbf{s}_{13} = \sum_i a_i^{13} = r_{1,1} + r_{2,4} - r_{3,7} + r_{4,1} + r_{5,4}$$

$$\mathbf{s}_{14} = \sum_i a_i^{14} = -r_{1,1} - r_{2,4} + 2 \cdot r_{3,7} - r_{4,1} - r_{5,4}$$

$$\mathbf{s}_{15} = \sum_i a_i^{15} = -r_{1,1} + r_{3,7} - r_{4,1}$$

Note that the recombination is correct if $x = \mathbf{s}_1 + \mathbf{s}_4 - \mathbf{s}_7$. Clearly this is the case.

$$\mathbf{s}_1 + \mathbf{s}_4 - \mathbf{s}_7 = x_{1,1} + x_{4,1} + x_{2,4} + x_{5,4} + x_{3,7} = x_1 + x_2 + x_3 + x_4 + x_5 = x$$

To verify the sharings are correct, we can simply verify the following equations (which arise from the parity check matrix of the ESP $\mathcal{M}$) all evaluate to zero (a task which we leave to the reader)

$$\begin{aligned}
\mathbf{s}_8 - 2\cdot\mathbf{s}_1 + 2\cdot\mathbf{s}_4 &= 2\cdot r_{1,1} - 2\cdot r_{2,4} + 2\cdot r_{4,1} - 2\cdot r_{5,4} - 2\cdot(r_{1,1} + r_{4,1}) \\
&\quad + 2\cdot(r_{2,4} + r_{5,4}) \\
\mathbf{s}_9 + 2\cdot\mathbf{s}_4 - 2\mathbf{s}_7 &= -2\cdot r_{2,4} + 2\cdot r_{3,7} - 2\cdot r_{5,4} + 2\cdot(r_{2,4} + r_{5,4}) - 2\cdot r_{3,7} \\
\mathbf{s}_{10} - \mathbf{s}_1 - 2\cdot\mathbf{s}_4 + 2\cdot\mathbf{s}_7 &= r_{1,1} + 2\cdot r_{2,4} - 2\cdot r_{3,7} + r_{4,1} + 2\cdot r_{5,4} - (r_{1,1} + r_{4,1}) \\
&\quad - 2\cdot(r_{2,4} + r_{5,4}) + 2\cdot r_{3,7} \\
\mathbf{s}_{11} + \mathbf{s}_1 + \mathbf{s}_4 - 2\mathbf{s}_7 &= -r_{1,1} - r_{2,4} + 2\cdot r_{3,7} - r_{4,1} - r_{5,4} + r_{1,1} + r_{4,1} \\
&\quad + r_{2,4} + r_{5,4} - 2\cdot(r_{3,7}) \\
\mathbf{s}_{12} + \mathbf{s}_1 - \mathbf{s}_7 &= -r_{1,1} + r_{3,7} - r_{4,1} + r_{1,1} + r_{4,1} - r_{3,7} \\
\mathbf{s}_{13} - \mathbf{s}_1 - \mathbf{s}_4 + \mathbf{s}_7 &= r_{1,1} + r_{2,4} - r_{3,7} + r_{4,1} + r_{5,4} - r_{1,1} + r_{4,1} - r_{2,4} \\
&\quad + r_{5,4} + r_{3,7} \\
\mathbf{s}_{14} + \mathbf{s}_1 + \mathbf{s}_4 - 2\cdot\mathbf{s}_7 &= -r_{1,1} - r_{2,4} + 2\cdot r_{3,7} - r_{4,1} - r_{5,4} + r_{1,1} + r_{4,1} + r_{2,4} \\
&\quad + r_{5,4} - 2\cdot(r_{3,7}) \\
\mathbf{s}_{15} + \mathbf{s}_1 - \mathbf{s}_7 &= -r_{1,1} + r_{3,7} - r_{4,1} + r_{1,1} + r_{4,1} - r_{3,7}
\end{aligned}$$

5.B.9 Shamir $(10, 4)$ for $\mathbb{Z}_{2^k}$

For this final example we use a degree $d_4 = 4$ extension to generate the matrix that is defined in the ESP $\mathcal{M} = (\mathbb{Z}_{2^k}, M, \varepsilon, \varphi)$. The complete ESP then becomes

$$M = \begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1
\end{pmatrix}$$

$$\varepsilon = (1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$$

$$\varphi(i) = \lceil i/4 \rceil$$

KRSW Algorithm:

This method can not be applied as the underlying ESP does not correspond to replicated secret sharing.

Smart-Wood Algorithm:

For the Smart-Wood reduction we start by doing the column reduction, however in this process a problem arises for the general case. In that the precise column operations performed depend on the precise choice of k. In all cases however we obtain a matrix M' such that

$$M' = \begin{pmatrix} \mathbf{I}_{17} \\ N \end{pmatrix}$$

for some dense matrix $N \in M_{23\times17}(\mathbb{Z}_{2^k})$. The costs then depend on the precise choice of χ, which is affected by the number of zero entries in new target vector ε', and hence no k.

We examine two sub-cases, which are relevant for our main tables, either $k = 128$ or $k = 1$. In our description below we refer to a specific column reduction of M, obviously different column reductions will produce different outcomes.

$k = 128$**:** When $k = 128$ the new target vector ε' from our specific column reduction is non-zero except in position 16. This means that we can explicitly define the function χ as follows: $\chi(1) = 1$, $\chi(2) = 5$, $\chi(3) = 9$, $\chi(4) = 13$, $\chi(5) = 17$, $\chi(6) = 2$, $\chi(7) = 3$, $\chi(8) = 4$, $\chi(9) = 6$, $\chi(10) = 7$, which gives us such that $\mathsf{im}(\chi) = \{1, 2, 3, 4, 5, 6, 7, 9, 13, 17\}$. From this we obtain

$$\begin{aligned}
K_1 =& \{1, 8, 10, 11, 12, 14, 15, 16\},\\
K_2 =& \{5, 8, 10, 11, 12, 14, 15, 16\},\\
K_3 =& \{9, 8, 10, 11, 12, 14, 15, 16\},\\
K_4 =& \{13, 8, 10, 11, 12, 14, 15, 16\},\\
K_5 =& \{17, 8, 10, 11, 12, 14, 15, 16\},\\
K_6 =& \{2, 8, 10, 11, 12, 14, 15, 16\},\\
K_7 =& \{3, 8, 10, 11, 12, 14, 15, 16\},\\
K_8 =& \{4, 8, 10, 11, 12, 14, 15, 16\},\\
K_9 =& \{6, 8, 10, 11, 12, 14, 15, 16\},\\
K_{10} =& \{7, 8, 10, 11, 12, 14, 15, 16\},
\end{aligned}$$

and $X_1 = \{P_1\}$, $X_2 = \{P_6\}$, $X_3 = \{P_7\}$, $X_4 = \{P_8\}$, $X_5 = \{P_2\}$, $X_6 = \{P_9\}$, $X_7 = \{P_{10}\}$, $X_9 = \{P_3\}$, $X_{13} = \{P_4\}$, $X_{17} = \{P_5\}$, and $X_i = \mathcal{P}$ for all $i \in \{8, 10, 11, 12, 14, 15, 16\}$. From this we can present our analysis of the algorithm in Figure 5.17.

- In step one we call $\mathcal{F}_{\text{PRZS}}$ once to generate $\langle t^0 \rangle$
- In steps two to four no communication happens
- In step five we call $\mathcal{F}_{\text{PRZS}}$ a total of 23 times to generate $\langle t^i \rangle$ for $i \in \{18, \ldots, 40\}$, and we communicate $207 = (40 - 18) \cdot (10 - 1)$ ring elements.
- In step six, for the sets X_i of size one we do nothing, however for each of the seven larger X_k we call $\mathcal{F}_{\text{PRZS}}$ once to generate $\langle t^k \rangle$. Each party in these larger sets X_k then has to communicate its value a_i^j to the party $\varphi(j)$, i.e. to 9 other parties. Thus, we need to communicate $63 = 7 \cdot 9$ elements in total.

This leads to a total cost of $31 = 1 + 23 + 7$ calls to the $\mathcal{F}_{\text{PRZS}}$ functionality and $270 = 207 + 63$ ring elements.

$k = 1$: When $k = 1$ our target vector will obviously have more zero components, in particular for our column reductions we obtain

$$\varepsilon' = (1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1).$$

This allows us to make the choice of χ (which itself depends on the placing of the zero entries in N for our choice of column reduction) of $\chi(1) = 1, \chi(2) = 4, \chi(3) = 11, \chi(4) = 13, \chi(5) = 17, \chi(6) = 3, \chi(7) = 7, \chi(8) = 8, \chi(9) = 1, \chi(10) = 12$, such that $\mathrm{Im}(\chi) = \{1, 3, 4, 7, 8, 11, 12, 13, 17\}$. From this we obtain

$$\begin{aligned}
K_1 =& \{1, 2, 5, 6, 9, 10, 14, 15, 16\},\\
K_2 =& \{2, 4, 5, 6, 9, 10, 14, 15, 16\},\\
K_3 =& \{2, 5, 6, 9, 10, 11, 14, 15, 16\},\\
K_4 =& \{2, 5, 6, 9, 10, 13, 14, 15, 16\},\\
K_5 =& \{2, 5, 6, 9, 10, 14, 15, 16, 17\},\\
K_6 =& \{2, 3, 5, 6, 9, 10, 14, 15, 16\},\\
K_7 =& \{2, 5, 6, 7, 9, 10, 14, 15, 16\},\\
K_8 =& \{2, 5, 6, 8, 9, 10, 14, 15, 16\},\\
K_9 =& \{1, 2, 5, 6, 9, 10, 14, 15, 16\},\\
K_{10} =& \{2, 5, 6, 9, 10, 12, 14, 15, 16\},
\end{aligned}$$

and hence, $X_1 = \{P_1, P_9\}$, $X_3 = \{P_6\}$, $X_4 = \{P_2\}$, $X_7 = \{P_7\}$, $X_8 = \{P_8\}$, $X_{11} = \{P_3\}$, $X_{12} = \{P_{10}\}$, $X_{13} = \{P_4\}$, $X_{17} = \{P_5\}$, and $X_i = \mathcal{P}$ for all $i \in \{2, 5, 6, 9, 10, 14, 15, 16\}$.

As before we can now analyse the algorithm in Figure 5.17.

- In step one we call $\mathcal{F}_{\mathrm{PRZS}}$ once to generate $\langle t^0 \rangle$
- In steps two to four no communication happens
- In step five we again call $\mathcal{F}_{\mathrm{PRZS}}$ a total of 23 times to generate $\langle t^i \rangle$ for $i \in \{18, \ldots, 40\}$, and we communicate $207 = (40 - 18) \cdot (10 - 1)$ ring elements.
- In step six we now do something slightly different: For the sets X_i of size one we again do nothing. For the set X_1 of size two we need to communicate one element. The eight sets X_i equal to $\mathcal{P}$ result in eights calls to $\mathcal{F}_{\mathrm{PRZS}}$, with each one resulting in nine elements being communicated, i.e. $8 \cdot 9 = 72$ in total.

This leads to a total cost of $32 = 1 + 23 + 8$ calls to the $\mathcal{F}_{\mathrm{PRZS}}$ functionality and a total transmission of $270 = 207 + 1 + 72 = 280$ ring elements.

CHAPTER 6

Feta: Efficient Threshold Designated-Verifier Zero-Knowledge Proofs

Carsten Baum[1], Robin Jadoul[2], Emmanuela Orsini[2], Peter Scholl[1], and Nigel P. Smart[2]

[1]Dept. Computer Science, Aarhus University, Aarhus, Denmark.
[2]imec-COSIC, KU Leuven, Leuven, Belgium.

[BJO+22] Carsten Baum, Robin Jadoul, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. Feta: Efficient threshold designated-verifier zero-knowledge proofs. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 293–306. ACM Press, November 2022.

Abstract: Zero-Knowledge protocols have increasingly become both popular and practical in recent years due to their applicability in many areas such as blockchain systems. Unfortunately, public verifiability and small proof sizes of zero-knowledge protocols currently come at the price of strong assumptions, large prover time, or both, when considering statements with millions of gates. In this regime, the most prover-efficient protocols are in the designated verifier setting, where proofs are only valid to a single party that must keep a secret state.

In this work, we bridge this gap between designated-verifier proofs and public verifiability by *distributing the verifier* efficiently. Here, a set of verifiers can then verify a proof and, if a given threshold t of the n verifiers is honest and trusted, can act as guarantors for the validity of a statement. We achieve this while keeping the concrete efficiency of current designated-verifier proofs, and present constructions that have small concrete computation and communication cost. We present practical protocols in the setting of threshold verifiers with $t < n/4$ and $t < n/3$, for which we give performance figures, showcasing the efficiency of our approach.

My contributions: Main author
I was a main collaborator on the design of the protocols and responsible for the full proof of concept implementation and its associated experiments.

6.1 Introduction

A zero-knowledge proof of knowledge (ZKPoK) is an interactive protocol which allows a prover to convince a verifier, given a statement x, that the prover knows a witness w such that the pair (x, w) lies in some NP language $\mathcal{L}$. This is done in such a way that the verifier learns nothing but the validity of the statement, i.e. they learn nothing about the witness w, only that the prover knows it. ZKPoKs have a wide range of applications, especially in the burgeoning area of blockchain [HBHW16], but also as building blocks of highly efficient signature schemes [CDG+17] or to increase the security level of existing cryptographic protocols from passive to active security in a black-box manner [GMW87].

There are various parameters that influence which ZKPoK scheme is suitable for a certain application. For example, when using ZKPoKs for blockchains one needs proofs that are *publicly verifiable* and *non-interactive*; namely the proof is sent in a single message from the prover such that any verifier can verify it. Another common requirement is that they are *succinct*, namely that the proof has size and verification time that is sublinear in the size of the statement.

Therefore, most ZKPoKs such as SNARKs [BCG+13] and STARKs [BBHR19] that are considered for practical applications within blockchains for instance, are mainly optimized for small proof size and verification time (and are also publicly verifiable and non-interactive). Their drawback is that prover running time can be prohibitive for large statements, i.e. statements expressed by arithmetic circuits with billions of gates. This is because the prover runtime for all current practical succinct schemes has an inherent $\mathsf{polylog}(|x|)$ overhead over the optimal $O(|x|)$ proof time and because prover memory access is not local[1], which leads to inherent slowdowns for increasing $|x|$.

Modern MPC-in-the-Head ZKPoKs such as KKW [KKW18] or Limbo [DOT21] have a proof size that is at least linear in $|x|$, with the unique exception of Ligero [AHIV17] which achieves sub-linear proof for large enough statements. In addition, they usually use a "light" inner proof (which is a passively-secure MPC scheme) that requires $O(|x|)$ computation, but must be repeated $s/\log(s)$ times to achieve negligible soundness error where s is the security parameter.

Alternative ZKPoKs for large statements, which also have a practically efficient prover due to small concrete constants, are either based on garbled circuits ([JKO13] and follow-ups) or vOLE-commitments [WYKW21, YSWW21, BMRS21]). All of these prover-efficient schemes have the disadvantage that they require the verifier to keep a secret state, i.e. they are designated-verifier

[1]There are theoretical works that achieve linear prover time such as e.g. [LSTW21], but to the best of our knowledge they are not concretely efficient.

ZKPoKs. This means that the proof can only be verified by a single party, who must be identified before the proof is produced. This makes the application in blockchains, where a proof may need to be verified by a set of validator nodes, impossible.

One can mitigate the problem of a designated verifier by distributing the verification among a larger set of parties. Here, each such verifier comes from a pre-defined, possibly large set, leading to a form of distributed designated verifier proof system. Now, if a majority of these verifiers is trusted, the statement of the prover can be accepted as validated by a majority of third parties.

Distributing Verification. This distribution of verification has an impact on the question of what a proof actually is, and also changes how protocols for such a setting can be designed.

- If the verifier is distributed, an adversary may corrupt multiple verifiers, in addition to the prover, in order to convince honest verifiers of the validity of a false statement. This means that soundness must be redefined to take this into consideration.
- When a proof is rejected, this might happen either if a prover does not have a proof or if it is honest, but verifiers may prevent successful verification of a proof. Hence, honest verifiers may want to distinguish these cases in order to not blame an honest prover or verifier as corrupt. So in the case of dishonest behaviour a security definition may require that honest verifiers do not just abort, but they also identify one (or more) of the cheating parties. This enables a form of cheater elimination.
- The distributed nature of the verifier may allow to obtain more efficient protocols: while in standard zero-knowledge the verifier must always be considered as fully corrupted, we may now be ok with only maintaining zero-knowledge if a strict subset of the verifiers does not collude.

6.1.1 Related Work

Thresholdizing in zero-knowledge proofs has a long history. The earliest works are those of [BD91] and [Bea91], both from 1991. In the work of [BD91] the verifiers do not need to agree on the validity of the proof, and in addition do not communicate directly. Our work is closer in spirit to that of [Bea91], although with a modern security definition and practical, concrete efficiency. In particular our security notion is UC-based, and captures issues related to a dishonest prover and dishonest verifiers colluding, as well as (by definition) providing a proof-of-knowledge. We also require that cheaters are identified which we feel is important in applications (and is missing in all prior work). The construction in [Bea91] is based, unsurprisingly for it's time, on Verifiable Secret Sharing (VSS), thus the protocol is highly inefficient compared to our more modern approach. Whilst VSS enables identifiable abort, it is unclear in [Bea91] how (or even if) this can be used to identify if a verifier and prover collaborate to cheat.

In the 2000's interests continued in this problem, but focused on proofs related to languages based on discrete logarithms (for example proving that certain discrete logarithm-based commitments satisfied some given properties). Work in this vein included [ACF02], which focused on statements related to relations between discrete logarithms. Their application are statements tailored for systems using VSS in MPC. We essentially lift the definitions of [ACF02] to a more general UC setting for arbitrary adversary structures, as well as extend the definitions to general languages (and not just those related to discrete logarithms).

Conceptually, our setting bears resemblance to the one considered in the MPC-in-the-head paradigm [IKOS07] where the proof is verified by a set of simulated verifiers. Compared to [IKOS07] we require that an adversarial prover can only cooperate with a small set of corrupt verifiers, as we assume a majority of verifiers to be honest.

There are other related works, which are similar but distinct from our own work. For example, one related notion is the concept of distributed zero-knowledge from [BBC$^+$19], which looks at the case where the *statement* x is unknown to any given verifier, and is instead secret shared. The protocols in that work only support a limited class of languages, and do not consider identifiable abort, and so are vulnerable to denial-of-service attacks from a malicious verifier. Our notion can be seen as orthogonal to Multi-Prover Interactive Proofs [BGKW88], where multiple provers act independently to convince a verifier. Our notion is also complementary to the setting considered in [WZC$^+$18] where the witness w is shared among a set of provers. Instead, we only have one prover and w is shared among the verifiers.

A relatively recent paper [BKZZ20] focuses on reducing the total amount of entropy needed by a set of verifiers, if all verifiers are to verify the proof. This is orthogonal to our work, as we require a joint/distributed verification where some verifiers can be dishonest. However, the idea of reducing the entropy requirement would be an interesting aspect to consider in the future. As would extending the ideas of [BKZZ20] to more general problem statements, since [BKZZ20] focuses on languages based on discrete logarithms.

Another interesting orthogonal direction to our work is that of "Fair-Zero Knowledge" introduced in [LMs05]. In this work a distributed-verifier notion is presented, where a prover might leak the secret to a dishonest verifier via a subliminal channel. Nevertheless, since our "online" proof stage only requires broadcast interaction from the prover to the verifiers, fairness as in [LMs05] for our type of proof systems might be interesting line for future work.

The renewed interest in the distributed verifier setting is shown by two recent papers by Yang et al. [YW22] and Applebaum et al. [AKP22]. Both works consider the case where a majority of verifiers are honest. Applebaum et al. focus more on the theoretical side and study the minimal assumptions needed to achieve round-optimal distributed verifier protocols; the work of Yang et al. is similar to our approach and oriented to real-world efficiency, however does not present an implementation and does not consider cheater identification, thus only achieving security with selective abort.

6.1.2 Our Contribution

In this work, we formalize the notion of Distributed Verifier ZKPoKs (DV-ZKPoKs) in the UC framework. We provide multiple constructions of such protocols, all with cheater identification, that are secure against different thresholds of corrupted verifiers[2].

New definitions. We first present a formal definition of what it means for a DV-ZKPoK to be secure in the UC framework. Let us first redefine the three standard properties of ZKPoKs to be applicable to the threshold setting:

Distributed Correctness: If the prover has a witness, then the honest parties either accept the proof or identify the same corrupted verifiers that interfered with the proof.

Distributed Soundness: If the prover does not have a witness then honest verifiers only accept with negligible probability, given not too many other verifiers are corrupted. In addition, the honest verifiers either agree that the prover does not have a witness, or will identify a set of corrupted verifiers.

Distributed Zero-Knowledge: The corrupted verifiers learn no new information beyond the fact that the statement is true.

[2]In our construction, the single (cheesy) verifier of the Mac-and-Cheese protocol [BMRS21] has been crumbled into a large set of smaller verifiers. Thus, our protocol name *Feta*.

Our definition will allow different adversarial structures for all of these properties. This means that our definition also encapsulates protocols where e.g. soundness breaks down if just one verifier is corrupted, but which are zero-knowledge even if all verifiers are corrupted.

There are a number of "naive" protocols which enable such distributed verifier zero-knowledge proofs using existing techniques. We will describe some of these protocols, showing the applicability of our framework.

New protocols. We then present two efficient DV-ZKPoK protocols together with necessary preprocessing protocols. These protocols are optimized for $t < n/4$ and $t < n/3$ corruptions, respectively, where n is the number of verifiers and t is the number of corrupted verifiers. Our protocols are plausibly post-quantum secure, and require as setup assumptions a PKI as well as a broadcast channel. The latter can easily be implemented if $t < n/3$ information theoretically.

Implementations. We have implemented our protocols in C++, showing concretely efficiency both in terms of prover and verifier time. For example, for the case of $t < n/4$, the combined pre-processing and prover time for proving knowledge of the pre-image of a single SHA-256 evaluation with $n = 5$ verifiers is about 10 milliseconds, with a proof time of around 7 milliseconds. The verification time is under 15 milliseconds. A circuit with a million AND gates requires a total proof time of 96 milliseconds pre-processing and 30 milliseconds for the proof generation. The verification time is 90 milliseconds. For $n = 100$ verifiers and $t = 20$ the million AND gate circuit times become 431 milliseconds for pre-processing, 176 milliseconds to generate a proof and 219 milliseconds for the 100 verifiers to verify it. This is with a single threaded implementation of our protocols.

As remarked above the prior works on distributed verifier zero-knowledge have all been for discrete logarithm based languages, as opposed to the general languages considered in this chapter. In addition, they have considered different and often less general security requirements, as we outlined above. Thus, to compare our implementation we are left, with the admittedly unsatisfactory situation of, comparison against either publicly verifiable or designated verifier proof systems.

Our run times are all significantly smaller than the single instance publicly-verifiable proofs of similar SHA-256 pre-images, using a system such as Ligero [AHIV17]. Using machines less powerful than the ones we used in our experiments, [AHIV17] give prover and verification times for a single pre-image of a SHA-256 evaluation of over 100 milliseconds. Our proof size, excluding pre-processing, is also significantly smaller (8 KBytes vs 100's of KBytes for Ligero). Note, Ligero provides a publicly verifiable proof as opposed to our distributed designated verifier proofs.

The Limbo system [DOT21], which again provides publicly verifiable proofs, reports single threaded prover and verifier times for the same circuit of 50 milliseconds, using machines comparable to the ones in our experiments, with their proof sizes being 42 KBytes.

The Mac-n-Cheese [WYKW21] and Quicksilver protocols [BMRS21], which provide designated verifier proofs using a single threaded implementation can achieve around 7 million AND gates per second in terms of prover/verification time. Translating this to the 22.573 AND gate SHA-256 circuit would equate to a prover/verification time of 3 milliseconds.

Thus, we see our prover/verification time of 6.5/10 milliseconds, for the SHA-256 circuit in the distributed verifier case, provides a compromise between slower publicly verifiable proofs and faster designated verifier proofs.

The protocol for the case of $t < n/3$ is slightly less efficient, but still provides a highly efficient methodology for performing distributed verifier zero-knowledge proofs. Also in this case both prover and verification time are significantly smaller than in publicly verifiable schemes like Ligero and Limbo.

Hence, we see that our notion of distributed designated verifier proofs can enable more efficient practical zero-knowledge proofs when compared to publicly verifiable proofs.

6.1.3 Applications

Protocols with distributed verification have a number of applications, mainly in blockchains.

- For *permissioned blockchains*, which are popular for use in companies, the validators (usually) authenticate the next block via majority voting. Such validators could act as the distributed verifiers for a proof. In such a situation the total number of validators is a handful, and thus the techniques of this chapter could be used to validate a proof *before* the next block is authenticated by the chosen validator.

- In *permissionless blockchains* with oracles (i.e. groups of parties that vouch for certain external facts), the oracle parties could serve as verifiers for our proofs. The oracles are e.g. trusted by a smart contract, and our distributed verifier means that this trust can be minimized in the case of proof verification. Oracles are sometimes also used in Layer-2 protocols on the blockchain. For example, in commit-chains like NOCUST [KZF+18], there is an operator responsible (i.e. an oracle) for committing the latest state of user account balances to the main blockchain every epoch. In the case of optimistic rollups (as in Arbitrum [KGC+18] and Optimism[3]), the verification and state-progression are done off-chain by the validator (i.e. an oracle) as well, while the final states (assertions) are published on the blockchain. Our distributed verifier proofs can act as a balance between optimistic rollups and full ZK-rollups. In all cases, the number of such oracles is relatively small and so the techniques of this chapter could be applied.

More generally, zero-knowledge with distributed verification can be used in all zero-knowledge applications where the verifiers are known ahead of time.

6.1.4 Techniques

On a high level, our protocols can be described using the following four-step paradigm:

1. The verifiers create consistent commitments to random values r_i such that only the prover can open these later. Here, if t or less verifiers are corrupted, then they cannot reconstruct the committed values themselves.

2. The verifiers and the prover check together that the commitments to the random values are indeed consistent among all verifiers, and that the prover knows the openings. If not, then cheaters are identified. If they are consistent, then the preprocessing of the DV-ZKPoK is considered as finished.

3. In the online phase, the prover uses the r_i to commit to w as well as auxiliary information necessary to show that $(x, w) \in R$. This commitment can ideally be done by sending one message via a broadcast channel.

4. Upon the prover having finished committing, the verifiers perform a proof verification step. Here we aim for a “cheap” proof verification that only requires the verifiers to communicate in $O(1)$ rounds, with a message complexity that is sublinear in $|x|$ or $|w|$ as well.

[3] https://community.optimism.io

To achieve this, our "preprocessing" phase lets the verifiers create many random Shamir secret sharings as commitments, where the prover only learns the secret being shared. Given the linearity of this secret sharing, consistency can easily be established using a linear test. This test only requires communication that scales in the number of parties but not $|x|$ or $|w|$. Moreover, we show that cheater identification can be achieved by additionally signing certain messages in the preprocessing protocol.

In our online phase, our protocols let the prover commit both to w as well as the intermediate wire values for a circuit C that evaluates to 0 iff w is a valid witness for the statement x. The verifiers re-evaluate C based on the committed w using the homomorphic properties of the commitment/secret sharing and check if the intermediate wire values are consistent with w and that the output of $C(w)$ is 0. This only requires a depth-1 circuit to be evaluated by the verifiers.

In the first protocol (for $t < n/4$) we make use of error-detecting properties of a Reed-Solomon code/Shamir sharing. The linear gates are free to evaluate as the Shamir sharing is linearly homomorphic, while the multiplication is performed by each verifier multiplying the input shares of a multiplication gate locally. The bound of $t < n/4$ comes from having to perform error detection on product codes (coming from degree $2 \cdot t$ polynomials stemming from the share multiplication), which is necessary to detect cheating during the multiplication protocol by a verifier.

Our second protocol (for $t < n/3$) is slightly more complex and avoids the verifiers having to multiply shares altogether. Instead, we let the prover commit to slightly more data and use a checking procedure for multiplications that is based on the Schwartz-Zippel Lemma, similarly to [BBC+19]. This means that multiplication checks only require linear operations.

6.2 Preliminaries

6.2.1 Shamir Sharing

Our protocols are built on top of Shamir's secret-sharing scheme [Sha79]. We briefly recap on it here in order to fix the notation we will use in the rest of the chapter.

A secret s, in a finite field $\mathbb{F}$, is shared amongst n parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ by the sharing party defining a random degree t polynomial $f_s(X)$ whose constant term is the value s. Assuming $n > |\mathbb{F}|$ and that the integers $\{1, \ldots, n\}$ are mapped to distinct non-zero values $\alpha_1, \ldots, \alpha_n$ in $\mathbb{F}$, each party P_i is given the share $s^{(i)} = f_s(\alpha_i) \in \mathbb{F}$. We denote such a sharing by $\langle s \rangle_t$.

Note that this secret sharing scheme is linear, namely given $\beta, \delta, \gamma \in \mathbb{F}$ and two sharings $\langle x \rangle_t$ and $\langle y \rangle_t$, both of degree t, parties can locally produce the sharing $\langle z \rangle_t$, where $z = \beta \cdot x + \delta \cdot y + \gamma$, by computing

$$z^{(i)} = \beta \cdot x^{(i)} + \delta \cdot y^{(i)} + \gamma.$$

Also note that one can linearly combine sharings of different degrees to produce a sharing of the maximal degree, i.e. given $\langle x \rangle_{t_1}$ and $\langle y \rangle_{t_2}$ then one can locally produce $\langle x + y \rangle_t$, where $t = \max(t_1, t_2)$, which we shall write as $\langle x \rangle_{t_1} + \langle y \rangle_{t_2}$.

Reconstruction of a secret s, shared via $\langle s \rangle_t$, requires $t + 1$ correct share values from different parties. It is well known that Shamir's secret sharing scheme defined as above is equivalent to a Reed-Solomon code $[n, t + 1, n - t]$ over $\mathbb{F}$, where the shares $(f_s(\alpha_1), \ldots, f_s(\alpha_n))$ are viewed as a codeword. In particular, when the number of dishonest parties is bounded by d and $n > t + 2 \cdot d$, the parties can robustly reconstruct a shared value $\langle s \rangle_t$, so that any party who lies about their sharings will be detected. In one of our protocols we will use the fact that, if $n > 4 \cdot t$ and $d < t$ we can robustly reconstruct a value for a sharing of degree $2 \cdot t$.

Assuming $n > t+2\cdot d$, we denote by $\mathsf{RobustReconstruct}(\langle s \rangle_t, d)$ the reconstruction algorithm associated with Shamir's scheme which outputs a pair (s, flag), where either $\mathsf{flag} = (\mathsf{correct}, \emptyset)$, indicating that all the shares are consistent with a degree t sharing, or $\mathsf{flag} = (\mathsf{incorrect}, \mathcal{D})$ where $\mathcal{D}$ indicates the parties who input an inconsistent share.

6.2.2 Digital Signatures

Our basic protocols will make use of digital signatures, for which we use the following two standard definitions.

Definition 6.1: Digital signature scheme

A digital signature scheme for message space $\mathcal{M}$ is given by the polynomial time algorithms (KeyGen, Sign, Verify).

- $\mathsf{KeyGen}(1^\lambda)$: On input a security parameter λ this randomized algorithm outputs a public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$.
- $\mathsf{Sign}(\mathfrak{sk}, m)$: On input of private key $\mathfrak{sk}$ and a message $m \in \mathcal{M}$, this (potentially) randomized algorithm outputs a digital signature

σ.

- $\mathsf{Verify}(\mathfrak{pk}, \sigma, m)$: On input of a public key $\mathfrak{pk}$, a message m and a purported signature σ, this algorithm outputs either true (meaning accept the signature) or false (meaning reject the signature).

A digital signature scheme is said to be correct if for each $m \leftarrow \mathcal{M}$ and $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, $\mathsf{Verify}(\mathfrak{pk}, \mathsf{Sign}(\mathfrak{sk}, m), m) = \mathsf{true}$.

A digital signature scheme is said to be UF-CMA secure if the probability of any adversary $\mathcal{A}$ winning the following game is negligible in λ

1. $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
2. $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathfrak{sk}, \cdot)}(\mathfrak{pk})$.
3. Output 'win' if and only if $\mathsf{Verify}(\mathfrak{pk}, \sigma^*, m^*) = \mathsf{true}$ and m^* was not queried to $\mathcal{A}$'s signing oracle.

6.2.3 Zero-knowledge Proofs

A standard zero-knowledge proof takes a statement x and a witness w from some NP relation $\mathcal{R}$. The prover $\mathcal{P}$ holds the pair $(x, w) \in \mathcal{R}$, whilst the verifier only has x. The goal of a zero-knowledge proof (of knowledge) is to convince the verifier that x is in the language $\mathcal{L}_\mathcal{R}$ of statements that have a witness in $\mathcal{R}$. This is done by asserting that the prover holds w such that $(x, w) \in \mathcal{R}$, while no information about w (bar the fact that the prover knows it) is revealed to the verifier. Informally, a zero-knowledge proof has three security properties:

Correctness: If $(x, w) \in \mathcal{R}$ then $\mathcal{V}$ always accepts.

Soundness: If $\mathcal{P}$ does not have w then $\mathcal{V}$ only accepts with negligible probability.

Zero-Knowledge: There exists a simulator $\mathcal{S}$ that on input x can create transcripts of protocol instances between $\mathcal{P}$ and $\mathcal{V}$ that make $\mathcal{V}$ accept.

In the designated verifier setting, the soundness only holds for a verifier that has a secret state.

6.2.4 Schwartz-Zippel Lemma

One of our protocols will make use of the Schwartz-Zippel lemma for univariate polynomials, which we state here.

Lemma 6.1: Schwartz-Zippel Lemma

Let $F \in \mathbb{F}[X]$ denote a non-zero polynomial of degree d over a field $\mathbb{F}$. Let S denote a finite subset of elements of $\mathbb{F}$. If one selects $r \in S$ uniformly at random then

$$\Pr[\ F(r) = 0\ ] \leq \frac{d}{|S|}.$$

6.2.5 Coin Flipping

We will utilize at various points the ideal functionality $\mathcal{F}_{\text{Rand}}(\mathcal{P}, M, \mathbb{F})$, described in Figure 6.1. This functionality allows a set of parties $\mathcal{P}$ to sample M uniformly random values from a finite field $\mathbb{F}$ such that each party learns these. It does this in a manner which has identifiable abort, in the case that the adversary aborts the execution of the protocol. The implementation of this functionality is standard: The parties agree on a shared single seed using a non-interactive commitment via broadcast, then open via broadcast, and then the seed is expanded into the desired number of random values from $\mathbb{F}$ using a PRG.

The Ideal $\mathcal{F}_{\text{Rand}}(\mathcal{P}, M, \mathbb{F})$ Functionality

On input $(\mathsf{Rand}, \mathsf{cnt})$ from all parties in $\mathcal{P}$, if the counter value is the same for all parties and has not been used before:

1. Sample $r_i \leftarrow \mathbb{F}$ for $i \in [M]$.
2. The values r_i are sent to the adversary, and the functionality waits for its input.
3. If the input is $\mathsf{Deliver}$ then the values r_i are sent to all parties. Otherwise, the adversary will return a non-trivial subset C_A of the dishonest parties. The value (Abort, C_A) is returned to all parties.

Figure 6.1: Functionality $\mathcal{F}_{\text{Rand}}(\mathcal{P}, M)$

6.3 Distributed Verifier Zero-Knowledge Proofs

Our definition of *Distributed Verifier Zero-Knowledge Proofs* (DV-ZKPoKs) aims to generalize the notion of a *Designated Verifier Zero-Knowledge Proof* to the threshold setting. Namely, we will have a set of designated verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ who jointly verify the correctness of the proof using an interactive protocol.

6.3.1 Zero-Knowledge in the Threshold Setting

As mentioned in Section 6.1 in a distributed verifier setting there might exist multiple verifiers $\mathcal{V}_i$, some of whom may collaborate with a potentially corrupt prover $\mathcal{P}$. For a DV-ZKPoK we therefore get the following intuitive properties.

Distributed Correctness: If $(x, w) \in \mathcal{R}$ then either all honest verifiers $\mathcal{V}$ always accept or all honest verifiers agree on a set of cheating verifiers C_A.

Distributed Soundness: If $\mathcal{P}$ does not have w then honest verifiers only accept with negligible probability.

Distributed Zero-Knowledge: There exists a simulator $\mathcal{S}$ that on input x can create transcripts of protocol instances between $\mathcal{P}$ and verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ that make verifiers accept.

Let $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ denote the set of verifiers. An *access structure* Γ on $\mathcal{V}$ is a monotonically increasing subset of $2^{\mathcal{V}}$, i.e., if $S \in \Gamma$ then we have $T \in \Gamma$ for all T such that $S \subseteq T \subseteq \mathcal{V}$. The *adversary structure* Δ associated with Γ is the set of all sets $\mathcal{V} \setminus S$ for $S \in \Gamma$.

When dealing with a potentially dishonest prover and a subset of potentially dishonest verifiers, we can consider three different access structures related to the three different properties of ZK proofs. We let the relevant access structures, for the potentially dishonest verifiers, be denoted by Γ_C (for Correctness), Γ_S (for Soundness) and Γ_Z (for Zero-Knowledge). With their different associated adversary structures being Δ_C, Δ_S and Δ_Z. We allow different access structures to provide better flexibility in applications, as well as more flexibility in designing protocols. To aid the reader one could initially think of the threshold case of $\Gamma_C = \Gamma_S = \Gamma_Z$ being all subsets of size greater than $n - t$, and $\Delta_C = \Delta_S = \Delta_Z$ being all subsets of the verifiers of size less than or equal to t.

We let $\mathcal{V}_\mathcal{D}$ denote the precise set of dishonest verifiers in a given protocol instance. We desire that at the end of the protocol, the verifiers either output Abort, Success or Fail. Here, Success or Fail imply that the proof was correct or not, respectively, while Abort means that some verifiers or the prover may have aborted. In all cases each honest party P will obtain a non-empty list of parties who aborted.

Distributed Correctness.

We first discuss correctness; as usual this assumes an honest prover. In the case of $\mathcal{V}_\mathcal{D} \notin \Delta_C$ then the adversary has enough power to break correctness. In this case some honest verifiers will abort, some will accept and some will fail - no common guarantees can be made. Note in the case when $\mathcal{V}_\mathcal{D} \notin \Delta_C$, the set C that each honest verifier identifies as corrupt parties in the case of abort, can be different for each of them, and they may even identify honest parties as corrupted. In the case of failure or success the honest verifiers may in addition identify cheating verifiers. This is captured by the procedure Breakdown() in our ideal functionality $\mathcal{F}_{\mathrm{DV-ZK}}$, which can be found in Figure 6.2.

However, when $\mathcal{V}_\mathcal{D} \in \Delta_C$ then the parties obtain consensus of output: either all honest verifiers output Success or they all output Abort. In the latter case, the verifiers identify a set $C_A \neq \emptyset$ of dishonest verifiers which is the same for each honest verifier. Consensus of output when $\mathcal{V}_\mathcal{D} \in \Delta_C$ is needed to avoid denial-of-service attacks where a single dishonest verifier can make the honest verifiers reject a valid proof. This is captured by the procedure CompleteWithAbort() in our ideal functionality $\mathcal{F}_{\mathrm{DV-ZK}}$.

Note that cheater identification is not necessary in the case of honest majority access structures Γ_C. This is because a simple majority vote will result in the honest verifiers accepting the proof (assuming consensus on accept). In the case of dishonest majority the ability for the honest parties to identify a single dishonest party (with consensus) will act as a deterrent to verifiers to act dishonestly. Thus even in the case of acceptance we allow the identification of dishonest verifiers so as to allow our functionality to capture the dishonest majority case.

Distributed Soundness.

Soundness considers the case of a dishonest prover. We require that if $\mathcal{V}_\mathcal{D} \notin \Delta_S$ then the adversary can get the honest verifiers to output anything it wants. Which is again captured by the procedure Breakdown() in Figure 6.2.

Protocol	Assumptions	Γ_C	Γ_S	Γ_Z
Protocol 0	Broadcast Channel	$\emptyset$	$\emptyset$	$\emptyset$
Protocol 0	no Broadcast Channel	$\mathcal{Q}_3$	$\mathcal{Q}_3$	$\emptyset$
Protocol 1	-	$\mathcal{Q}_3$	$\mathcal{Q}_3$	$\emptyset$
Protocol 2	Robust/identifiable abort MPC Protocol for Γ	Γ	Γ	Γ
Protocol 3	Threshold structures	$t_c < (n+1)/3$	$t_s < (n+1)/3 - 1$	$t_z < (n+1)/3$
$\Pi_{4\text{t}}$	Digital Signatures	$t < n/4$	$t < n/4$	$t < n/4$
$\Pi_{3\text{t}}$	Digital Signatures	$t < n/3$	$t < n/3$	$t < n/3$

Table 6.1: Comparison of Protocols

As we require the prover to input a witness w, if $\mathcal{V}_\mathcal{D} \in \Delta_S$ and if $(x, w) \in \mathcal{R}$ then the *worst* $\mathcal{P}$ can do is get some honest verifiers to abort and identify a cheating party. This is again captured by the procedure CompleteWithAbort() in Figure 6.2. On the other hand, if $(x, w) \notin \mathcal{R}$ then the *best* $\mathcal{P}$ can achieve is to get some honest verifiers to abort and identify a cheating party (which could include the prover). Again, this is captured by the procedure FailWithAbort() in Figure 6.2.

Distributed Zero-Knowledge.

Finally, in the case of an honest prover, if $\mathcal{V}_\mathcal{D} \notin \Delta_Z$ then the adversary has enough power to break the zero-knowledge property and potentially learn information about w. But if $\mathcal{V}_\mathcal{D} \in \Delta_Z$ then the adversary cannot learn w.

It is straightforward to change $\mathcal{F}_{\mathrm{DV-ZK}}$ so that it only has unanimous abort. Another interesting strengthening is to not permit identifiable aborts if $\mathcal{V}_\mathcal{D} \in \Delta_C$. Since this setting seems to be not achievable if a majority of verifiers is corrupted for any interesting protocol[4], we have opted for a definition that is achievable in both the honest and dishonest-majority setting.

6.3.2 Examples

We now explain the ideas behind our definition by presenting some naïve protocols that $\mathcal{F}_{\mathrm{DV-ZK}}$ captures, with different access structures Γ_C, Γ_S, and Γ_Z. In Table 6.1, we present a comparison of four "naïve" protocols, alongside our two more elaborate constructions, $\Pi_{4\text{t}}$ and $\Pi_{3\text{t}}$.

[4]It is achievable if the prover broadcasts a publicly verifiable proof to all verifiers. If the verifiers need to use a secret-shared state to validate the proof, then dishonest-majority completeness implies that $< n/2$ verifiers are sufficient to perform this validation and possibly reconstruct the secret state. But then, this implies that $< n/2$ corrupted verifiers can use their knowledge to aid a dishonest prover to break soundness.

Functionality $\mathcal{F}_{\mathrm{DV-ZK}}$

This functionality communicates with $n+1$ parties $\mathcal{P}, \mathcal{V}_1, \ldots, \mathcal{V}_n$ as well as the ideal adversary $\mathcal{S}$. We call $\mathcal{P}$ the prover and $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ the verifiers. For simplicity, we write $\mathcal{W} = \mathcal{V} \cup \{\mathcal{P}\}$. The functionality is instantiated with descriptions of three access structures $\Gamma_C, \Gamma_S, \Gamma_Z \subseteq 2^{\mathcal{V}}$, and their associated adversary structures Δ_C, Δ_S and Δ_Z. The adversary structures denote which parties $\mathcal{S}$ can corrupt without leading to a loss of correctness, soundness or zero-knowledge. Let `init` be a flag that is initially $\perp$.

Corrupt: Before any other command, $\mathcal{S}$ sends $(\mathsf{Corrupt}, \mathcal{D})$ where $\mathcal{D} \subseteq \mathcal{W}$. Let $\mathcal{H} = \mathcal{W} \setminus \mathcal{D}$. If $\mathcal{P} \in \mathcal{D}$ then we call the prover "corrupted", otherwise "honest". We call $\mathcal{V}_{\mathcal{D}} = \mathcal{V} \cap \mathcal{D}$ the corrupted verifiers and $\mathcal{V}_{\mathcal{H}} = \mathcal{V} \setminus \mathcal{V}_{\mathcal{D}}$ the honest verifiers.

Init: On input (Init) by all parties in $\mathcal{H}$:

1. Send $(\mathsf{Init?})$ to $\mathcal{S}$. If $\mathcal{S}$ responds with $(\mathtt{OK})$ then send (InitOK) to all parties in $\mathcal{H}$ and set `init` $\leftarrow \top$. Otherwise, send (Abort) to all parties in $\mathcal{H}$.

ProveHonest: On input (Prove, x, w) by $\mathcal{P} \in \mathcal{H}$ as well as (Prove, x) by all parties in $\mathcal{V}_{\mathcal{H}}$, if `init` $= \top$ and if $(x, w) \in R_L$:

1. If $\mathcal{V}_{\mathcal{D}} \notin \Delta_Z$ then send $(\mathsf{Prove?}, x, w)$ to $\mathcal{S}$, otherwise send $(\mathsf{Prove?}, x)$.
   - If $\mathcal{V}_{\mathcal{D}} \notin \Delta_C$ then run $\mathsf{Breakdown}()$.
   - If $\mathcal{V}_{\mathcal{D}} \in \Delta_C$ then run $\mathsf{CompleteWithAbort}()$.

ProveDishonest: On input (Prove, x, w) by $\mathcal{S}$ if $\mathcal{P} \in \mathcal{D}$ as well as (Prove, x) by all parties in $\mathcal{V}_{\mathcal{H}}$ and if `init` $= \top$:

- If $\mathcal{V}_{\mathcal{D}} \notin \Delta_S$ or $\mathcal{V}_{\mathcal{D}} \notin \Delta_C$ then run $\mathsf{Breakdown}()$.
- If $\mathcal{V}_{\mathcal{D}} \in \Delta_S$, $\mathcal{V}_{\mathcal{D}} \in \Delta_C$ and $(x, w) \in R_L$ then run $\mathsf{CompleteWithAbort}()$.
- If $\mathcal{V}_{\mathcal{D}} \in \Delta_S$, $\mathcal{V}_{\mathcal{D}} \in \Delta_C$ and $(x, w) \notin R_L$ then run $\mathsf{FailWithAbort}()$.

Figure 6.2: Functionality $\mathcal{F}_{\mathrm{DV-ZK}}$ for Distributed-Verifier ZK

Functionality $\mathcal{F}_{\mathrm{DV-ZK}}$ (cont.)

Method Breakdown()**:**

1. Wait for a message $(\mathsf{Abort}, A, F, S, C)$ from $\mathcal{S}$ where A, F, S are disjunct sets, $A \cup F \cup S = \mathcal{H}$, $C_A : \mathcal{H} \to 2^{\mathcal{W}}$.
2. Send $(\mathsf{Abort}, x, C_A(P))$ to each $P \in A$, $(\mathsf{Fail}, x, C_A(P))$ to each $P \in F$ and $(\mathsf{Success}, x, C_A(P))$ to each $P \in S$.

Method CompleteWithAbort()**:**

1. Wait for a message (Abort, b, C_A) from $\mathcal{S}$ where $C_A \subseteq \mathcal{V}_{\mathcal{D}}$, $b \in \{0, 1\}$ and $C_A \neq \emptyset$ if $b = 0$.
2. If $b = 0$ then send (Abort, x, C_A) to each $P \in \mathcal{H}$, otherwise send $(\mathsf{Success}, x, C_A)$ to each $P \in \mathcal{H}$.

Method FailWithAbort()**:**

1. Wait for a message (Abort, b, C_A) from $\mathcal{S}$ where $C_A \subseteq \mathcal{V}_{\mathcal{D}}$, $b \in \{0, 1\}$ and $C_A \neq \emptyset$ if $b = 0$.
2. If $b = 0$ then send (Abort, x, C_A) to each $P \in \mathcal{H}$, otherwise send (Fail, x, C_A) to each $P \in \mathcal{H}$.

Figure 6.3: Functionality $\mathcal{F}_{\mathrm{DV-ZK}}$ for Distributed-Verifier ZK, continued

P0: Send a NIZK Assuming the existence of a functionality $\mathcal{F}_{\mathrm{NIZK}}$, as well as a broadcast channel, we can easily realize $\mathcal{F}_{\mathrm{DV-ZK}}$. There is no preprocessing (bar what is needed to set up the functionality $\mathcal{F}_{\mathrm{NIZK}}$) and the prover simply broadcasts the non-interactive proof. The verifiers then verify it using $\mathcal{F}_{\mathrm{NIZK}}$ and then come to consensus on the output. In the case of acceptance, any party who does not concur is determined to be an identified adversary. In that case $\Gamma_C = \Gamma_S = \Gamma_Z = \emptyset$, i.e. we can tolerate any set of adversaries possible. Without a broadcast channel, Γ_C and Γ_S instead follow from e.g. standard bounds on Byzantine agreement. The protocol can only be simulated if $\mathcal{F}_{\mathrm{NIZK}}$ is straight-line extractable.

P1: Secret-Share a Proof Suppose we have a single access structure Γ over the verifiers, we let $\langle \cdot \rangle$ denote an information theoretic secret sharing scheme which respects this access structure. A trivial protocol is to take a non-interactive two party ZKPoK, for the prover to generate a proof π and then simply generate a sharing $\langle \pi \rangle$ of that proof and distribute it to the verifiers. The verifiers then (simply) publish their received share.

In terms of correctness we require $\Gamma_C = \Gamma$ is $\mathcal{Q}_3$[5]. This follows as we require, in the presence of dishonest verifiers, that honest verifiers output either success with consensus, or output abort with consensus, and identify the cheater.

In terms of soundness we also require that $\Gamma_S = \Gamma$ is $\mathcal{Q}_3$, this follows as the proof π is already sound. Thus, we require that for a (real or fake) proof that the verifiers come to a consensus and either identify a cheating verifier, or identify (in the case of a fake proof) that the prover has generated a fake proof.

In terms of zero-knowledge we have $\Gamma_Z = \emptyset$ since the initial proof π is zero-knowledge.

P2: Secret Share a Witness Instead of sharing the proof, the prover simply shares the witness according to some access structure Γ, and then the verifiers engage in an MPC protocol respecting Γ evaluating the circuit which verifies the witness. The zero-knowledge property is weaker than before, as we have $\Gamma_Z = \Gamma$. If the dishonest verifiers are not in the allowed adversary structure Δ then they can recover the witness and break the zero-knowledge property. The correctness, and the associated Γ_C, follow from the underlying MPC protocol (which needs to be a protocol which is either robust, or with identifiable abort). For soundness, and the associated Γ_S, we obtain $\Gamma_S = \Gamma_C$ by the correctness of the MPC protocol.

The advantage of this example, over P1 is that the prover has *almost no overhead* over secret-sharing the witness - it itself is not required to compute any kind of proof. In comparison to this generic protocol is highly likely to be significantly less efficient than our specialized protocols Π_{4t} and Π_{3t}, which can be seen as variants of this protocol idea. Our protocols Π_{4t} and Π_{3t} perform this optimization by removing the expensive circuit evaluation needed in a generic MPC solution; this is done at the expense of the prover needing to provide more share values for the circuit evaluation and not just sharing a witness.

P3: Joint MPC It may seem from the previous examples that we always have $\Gamma_C = \Gamma_S$, but this does not have to be the case. Consider the following construction, where we assume an MPC protocol run between the prover and the verifiers. The verifiers have no input, but the prover inputs the witness w.

[5]A $\mathcal{Q}_3$ access structure can be simply thought of as one which admits robust opening, see [HM97]

The common output (for the verifiers) is the evaluation of the checking circuit on the witness, or an identified cheater.

The proof is interactively performed between the prover and the verifiers by running the MPC protocol. Consider the case where Γ_C is a threshold structure on the n verifiers, with threshold t_C. In this case we can have that $t_C < (n+1)/3$ (because the prover acts honestly) and we can use an information theoretic robust protocol to ensure correctness. This also ensures that we have $t_Z < (n+1)/3$.

Now consider Γ_S with a threshold structure with threshold t_S. For the same protocol and soundness we actually have an additional adversary (the prover), and now require that $t_S + 1 < (n+1)/3$. Thus, depending on n, we can have different bounds on the maximum values of t_C and t_S and thus Γ_C may not be equal to Γ_S.

6.4 Preprocessing for distributed proofs with honest majority $t < n/2$

We begin by outlining the preprocessing phase for our proof in the presence of an honest majority. This preprocessing can then be used with the actual online phases of the proof, which require $t < n/4$ (Section 6.5) or $t < n/3$ (Section 6.6) corruptions. The ideal preprocessing functionality $\mathcal{F}^{t,n}_{\text{Prep}}$ is described in Figure 6.4. Both the protocols and functionality are defined over an extension field of appropriate degree to allow for Shamir secret sharing with n parties. We focus on the case of a binary field $\mathbb{F}_{2^k}$ with $2^k > n$, but our protocols are easily adapted to $\mathbb{F}_q$ for any $q > n$. We also use a repetition factor ρ such that $2^{k\cdot\rho} > 2^{\mathsf{sec}}$, where sec is our security parameter.

In the protocol $\Pi^{\mathsf{t,n}}_{\mathsf{Prep}}$ that implements the preprocessing functionality, and given in Figure 6.5, each of the n verifiers $\mathcal{V}_i$ samples a random r_i and sends a share of $\langle r_i \rangle_t$ to each other verifier and r_i to the prover $\mathcal{P}$. These values are checked for consistency by forming a random linear combination using random values α_i. This random linear combination simultaneously guarantees the correctness of the underlying secret known to the prover and the consistency of the shares on a degree t polynomial. It can be repeated to achieve negligible soundness error. Next, let $\langle \vec{r} \rangle_t$ be the vector representing all sharings made by the verifiers, and let M_t be an $(n-t) \times n$ Vandermonde matrix. The verifiers locally compute the sharings $\langle \vec{s} \rangle_t = M_t \cdot \langle \vec{r} \rangle_t$, while the prover computes $\vec{s} = M_t \cdot \vec{r}$. This randomness extraction ensures that out of these n shares, of which t are known to the adversary, $n-t$ uniformly random shares are recovered, unknown to any other party than the prover. Several instances of this preprocessing phase

Functionality $\mathcal{F}_{\text{Prep}}^{t,n}$

This functionality communicates with $n + 1$ parties $\mathcal{P}, \mathcal{V}_1, \ldots, \mathcal{V}_n$ as well as the ideal adversary $\mathcal{S}$, where $\mathcal{P}$ denotes the prover and $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ the verifiers. Let $\mathcal{W} = \mathcal{V} \cup \{\mathcal{P}\}$ and $t < n/2$.

Corrupt: Before any other command, $\mathcal{S}$ sends $(\mathsf{Corrupt}, \mathcal{D})$ where $\mathcal{D} \subseteq \mathcal{W}$. Let $\mathcal{H} = \mathcal{W} \setminus \mathcal{D}$. If $\mathcal{P} \in \mathcal{D}$ then we call the prover "corrupted", otherwise "honest". We call $\mathcal{V}_{\mathcal{D}} = \mathcal{V} \cap \mathcal{D}$ the corrupted verifiers and $\mathcal{V}_{\mathcal{H}} = \mathcal{V} \setminus \mathcal{V}_{\mathcal{D}}$ the honest verifiers.

Distribute Shares: On input (Shares, n_S) from all parties

1. Sample n_S random values $s_i \in \mathbb{F}_{2^k}$ for $i \in [n_S]$.
2. If $\mathcal{P}$ is corrupted then send $\{s_i\}_{i\in[n_S]}$ to $\mathcal{S}$.
3. Wait for a message (Abort, C_A) from $\mathcal{S}$ where $\emptyset \neq C_A \subseteq \mathcal{D}$ or $(\mathsf{Continue}, \{\hat{s}_i^{(p)}\}_{p\in\mathcal{V}_{\mathcal{D}}, i\in[n_S]})$.
   - If $\mathcal{S}$ inputs Abort then (Abort, C_A) is returned to each party in $\mathcal{H}$ and the functionality aborts.
   - If $\mathcal{S}$ inputs $\mathsf{Continue}$ then generate a Shamir sharing of s_i of degree t for each $i \in [n_H]$, which we denote by $\langle s_i \rangle_t$. The individual Shamir shares are denoted by $s_i^{(j)} \in \mathbb{F}_{2^k}$ for $j \in [n]$. The sharing is chosen so that $s_i^{(j)} = \hat{s}_i^{(j)}$. The values s_i are passed to $\mathcal{P}$ if $\mathcal{P} \in \mathcal{H}$, whilst the values $s_i^{(p)}$ are given to $\mathcal{V}_p$ for $p \in \mathcal{V}_{\mathcal{H}}$.

Figure 6.4: Functionality $\mathcal{F}_{\text{Prep}}^{t,n}$ for preprocessing in the case when $t < n/2$

are performed in parallel to obtain more than $n - t$ secret sharings, with (at least) an additional ρ sharings produced so as to verify the entire production is correct.

The protocol assumes a PKI in which each verifier $\mathcal{V}_i$ has a public key $\mathfrak{pk}_i$ and a signing key $\mathfrak{sk}_i$, which enables them to authenticate sent messages m with a digital signature $\mathsf{Sign}(\mathfrak{sk}_i, m)$. In the case when the consistency check fails, this allows parties to reveal the shares that they obtained from each other. This means that parties can identify cheaters by either identifying incorrectly generated sharings or incorrectly formed messages. Signatures prevent dishonest parties from framing honest parties by claiming to have obtained shares that

Protocol $\Pi^{\mathsf{t,n}}_{\mathsf{Prep}}$

We let M_t be an $(n-t) \times n$ Vandermonde matrix for randomness extraction. The protocol is parametrized by the number of verifiers n, number of corruptions $t < n/2$ and two integers n_S and ρ.
The protocol uses the hybrid functionality $\mathcal{F}_{\text{Rand}}$. If $\mathcal{F}_{\text{Rand}}$ sends (Abort, C_A) then each party in the protocol outputs (Abort, C_A) and terminates.

Abort(ℓ): Each $\mathcal{V}_i$ holds $r^{(i)}_{v,j}, \sigma^{(i)}_{v,j}$ for $v \in [n]$ and $j \in [\lceil (n_S+\rho)/(n-t) \rceil]$, while $\mathcal{P}$ holds $r_{v,j}, \sigma_{v,j}$ for $v \in [n]$ and $j \in [\lceil (n_S+\rho)/(n-t) \rceil]$ (for simplicity, each $\mathcal{V}_i$ signs a share $r^{(i)}_{i,j}$ for itself).

1. Each verifier $\mathcal{V}_i$ broadcasts $\{r^{(i)}_{v,j}, \sigma^{(i)}_{v,j}\}_{v,j}$, while $\mathcal{P}$ broadcasts $\{r_{v,j}, \sigma_{v,j}\}_{v,j}$. If any signature $\sigma^{(i)}_{v,j}$ does not hold then identify $\mathcal{V}_i$ as a cheater and abort. If any $\sigma_{v,j}$ does not hold then identify $\mathcal{P}$ as cheater and abort.
2. If for some $i \in [n]$ it holds that $T^{(i)}_\ell \neq \sum_{v,j} \alpha_{v,j,\ell} \cdot r^{(i)}_{v,j}$ then identify $\mathcal{V}_i$ as cheater and abort. If it holds that $T_\ell \neq \sum_j \sum_v \alpha_{v,j,\ell} \cdot r_{v,j}$ then identify $\mathcal{P}$ as a cheater and abort.
3. For any $\mathcal{V}_v$, if $r^{(1)}_{v,j}, \dots, r^{(n)}_{v,j}$ do not form a valid degree-t sharing of $r_{v,j}$ then identify $\mathcal{V}_v$ as a cheater and abort.

Figure 6.5: Protocol for preprocessing with $t < n/2$

the honest party never sent.

Theorem 6.1

Assuming that Sign is an unforgeable signature scheme, then the protocol $\Pi^{\mathsf{t,n}}_{\mathsf{Prep}}$ in Figure 6.5 securely implements the functionality $\mathcal{F}^{t,n}_{\text{Prep}}$ in the $\mathcal{F}_{\text{Rand}}$-hybrid model against any static adversary corrupting at most $t < n/2$ parties except with probability $2^{-\rho \cdot k+1}$.

Before proving the theorem, we give three lemmas that will simplify the proof. First, we show that if a dishonest party creates an incorrect sharing, then the protocol enters **Abort** with overwhelming probability. Second, we show that if a verifier sends an incorrect share to an honest prover, then the protocol enters

Abort with overwhelming probability. Finally, we show that upon entering **Abort** at least one dishonest party is identified, and only dishonest parties are identified.

Lemma 6.2

Let $\mathcal{V}_\mathcal{H} = \mathcal{V} \cap \mathcal{H}$ and assume $t < n/2$. For $v \in [n]$, consider the shares $r_{v,j}^{(i)}$ for $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and let $S_{v,j}$ be the unique polynomials of smallest degree over $\mathbb{F}_{2^k}$ such that $S_{v,j}(i) = r_{v,j}^{(i)}$. If there exist v, j such that[a] $deg(S_{v,j}) > t$, then the protocol enters **Abort** except with probability $2^{-k\cdot\rho}$.

[a]Here we use that $t < n/2$, as $S_{v,j}$ could otherwise not be of degree $> t$.

Proof. Computing $T_\ell^{(i)} = \sum_j \sum_v \alpha_{v,j,\ell} r_{v,j}^{(i)}$ is the same as computing the polynomials $S_\ell = \sum_{v,j} \alpha_{v,j,\ell} \cdot S_{v,j}$ first and then evaluating S_ℓ at points i to obtain the shares $T_\ell^{(i)}$ of the honest parties. This follows from the linearity of Lagrange interpolation.

Any additional point $T_\ell^{(v)}$ provided by the adversary through party $\mathcal{V}_v$ can either lie on the polynomial S_ℓ or not. If it does then S_ℓ will keep its degree, if not then the points $T_\ell^{(1)}, \ldots, T_\ell^{(n)}$ must lie on a polynomial of larger minimal degree. This means that the protocol enters **Abort** if any of the protocols S_ℓ is of degree $> t$, independent of the values $T_\ell^{(v)}$ sent by $\mathcal{S}$.

Let $r = \max_{v,j}\{\deg(S_{v,j})\}$, by definition we have $r > t$. This means that for some $S_{v,j}$ the monomial X^r has a non-zero coefficient. Then any S_ℓ will only be of degree $< r$, i.e. the shares of honest parties will lie on a degree-$< r$ polynomial, if the coefficients of the monomials X^r of all $S_{v,j}$ sum to 0 in S_ℓ. By the random choice of the $\alpha_{v,j,\ell}$ through $\mathcal{F}_{\text{Rand}}$ after these $S_{v,j}$ are fixed, this only happens with probability 2^{-k} for a single S_ℓ and with probability $2^{-k\rho}$ for all $S_1, \ldots, S_\rho$ simultaneously. □

Lemma 6.3

Let $\mathcal{V}_\mathcal{H} = \mathcal{V} \cap \mathcal{H}$ and $t < n/2$ and assume $\mathcal{P} \in \mathcal{H}$. For $v \in [n]$, consider the shares $r_{v,j}^{(i)}$ of $\mathcal{V}_i \in \mathcal{V}_\mathcal{H}$ and let $S_{v,j}$ be the unique polynomials of degree t over $\mathbb{F}_{2^k}$ such that $S_{v,j}(i) = r_{v,j}^{(i)}$. Furthermore, let $r_{v,j}$ be the values received by $\mathcal{P}$. If there exist v, j such that $S_{v,j}(0) \neq r_{v,j}$, then the protocol enters **Abort** except with probability $2^{-k\cdot\rho}$.

Proof. Observe that $\alpha_{v,j,\ell}$ are only chosen through $\mathcal{F}_{\text{Rand}}$ after all $r_{v,j}^{(i)}, r_{v,j}$ have been fixed, $v \in [n]$.

Assume that the protocol does not enter **Abort**, then for each $\ell \in [\rho]$ it holds that

$$\sum_{v,j} \alpha_{v,j,\ell} r_{v,j} = \sum_{v,j} \alpha_{v,j,\ell} \cdot S_{v,j}(0)$$

which can be rewritten as

$$0 = \sum_{v,j} \alpha_{v,j,\ell} \cdot (r_{v,j} - S_{v,j}(0))$$

Write $S_{v,j}(0) = r_{v,j} + \delta_{v,j}$. By assumption, there must exist v, j such that $\delta_{v,j} \neq 0$. Hence, it must hold that the $\delta_{v,j}$ chosen by the adversary lie in the kernel of $\alpha_{v,j,\ell}$ which are chosen uniformly at random after $\delta_{v,j}$ are fixed. For any ℓ, this happens with probability at most 2^{-k} and with probability at most $2^{-k\rho}$ for all $\ell \in [\rho]$ simultaneously. □

Lemma 6.4

Assuming unforgeability of Sign, then **Abort** always terminates with at least one dishonest party being identified. Furthermore, it only terminates identifying dishonest parties.

Proof. In Step 1 of **Abort** the protocol only identifies dishonest parties. This is because honest parties would have asked for shares with valid signatures in Step 1(a)iv of **Distribute Shares**. Similarly, we identify a dishonest prover as an honest prover would have asked for correctly signed data in Step 1(a)v of **Distribute Shares**.

In Step 2 we only identify dishonest parties, as honest parties would have computed $T_\ell^{(i)}, T_\ell$ correctly.

Assuming we reach Step 3 without aborting, then all $T_\ell^{(i)}, T_\ell$ were computed correctly but either $T_\ell^{(i)}$ do not form a polynomial of degree t or do not share the secret T_ℓ. If for each v, j the shares $r_{v,j}^{(i)}$ would form a degree-t sharing of $r_{v,j}$ then the condition for entering **Abort** cannot be reached. Thus, there must exist v, j such that the polynomial formed by $r_{v,j}^{(i)}$ is of larger degree or reconstructs to a value that is not $r_{v,j}$.

If $\mathcal{V}_v$ was honest then all $r_{v,j}^{(1)}, \ldots, r_{v,j}^{(n)}$ revealed during Step 1 lie on a degree-t polynomial. The protocol only identifies an honest party $\mathcal{V}_v$ in Step 3 if

$r_{v,j}^{(1)}, \ldots, r_{v,j}^{(n)}$ lie on a polynomial of degree $t+1$ or higher. As honest parties report the shares of $\mathcal{V}_v$ honestly, this only happens if an incorrect $\tilde{r}_{v,j}^{(i)}$ is broadcast by a corrupt $\mathcal{V}_i$, together with a valid signature under $\mathfrak{sk}_v$ (as we would have otherwise aborted in Step 1). So an honest $\mathcal{V}_v$ is only identified as a cheater if a signature was forged by $\mathcal{V}_i$, contradicting the assumption that the signature scheme is unforgeable. Similarly, an honest $\mathcal{V}_v$ would always send the correct shared $r_{v,j}$ to $\mathcal{P}$ so $\mathcal{P}$ can only reveal $\tilde{r}_{v,j}$ that is inconsistent with $r_{v,j}^{(1)}, \ldots, r_{v,j}^{(n)}$ if it can forge a signature, contradicting the assumption. Therefore, any $\mathcal{V}_v$ identified by Step 3 must be corrupted. □

We can now give the simulation-based proof of Theorem 6.1.

Proof. (of Theorem 6.1) The simulator $\mathcal{S}$ obtains as input from the environment the set $\mathcal{D}$ of corrupted parties and forwards this to $\mathcal{F}_{\text{Prep}}^{t,n}$. It furthermore sets up a copy of $\mathcal{F}_{\text{Rand}}$. If $\mathcal{P} \in \mathcal{H}$ then $\mathcal{S}$ will simulate an honest prover. Moreover, for each $\mathcal{V}_i \in \mathcal{H}$ $\mathcal{S}$ will simulate an honest verifier. It will generally follow the protocol, except if specified otherwise below. Initially, let $C_A = \emptyset$. Send (Shares, n_S) in the name of all simulated honest parties to $\mathcal{F}_{\text{Prep}}^{t,n}$. If $\mathcal{P} \in \mathcal{D}$ then $\mathcal{S}$ obtains the shares s_i from $\mathcal{F}_{\text{Prep}}^{t,n}$. If at any point $\mathcal{F}_{\text{Rand}}$ outputs (Abort, C_A) then $\mathcal{S}$ sends (Abort, C_A) to $\mathcal{F}_{\text{Prep}}^{t,n}$.

$\mathcal{S}$ simulates the honest verifiers $\mathcal{V}_i$ in Step 1(a)ii by sending uniformly random $r_{i,j}^{(v)}$ to each corrupted $\mathcal{V}_v$. It then waits for the sharings of the dishonest parties being sent to the simulated honest verifiers. If any of these sharings is of degree $> t$ for a dishonest verifier $\mathcal{V}_v$ then add v to the set C_A, otherwise denote $\langle r_{v,j} \rangle_t$ as the secret sharings of the dishonest parties.

If $\mathcal{P} \in \mathcal{D}$ then choose $r_{i,j}$ for the honest verifiers such that a prover following Step 3 will obtain s_i as output, and send these $r_{i,j}$ to the corrupt $\mathcal{P}$. This is always possible using [BTH08]. If instead $\mathcal{P} \notin \mathcal{D}$ then choose uniformly random $r_{i,j}$ for each honest verifier and wait for values $\tilde{r}_{v,j}$ being sent from the dishonest verifiers to the simulated $\mathcal{P}$. For any of these shares $\langle r_{v,j} \rangle_t$ that does not reconstruct to $\tilde{r}_{v,j}$ add v to C_A. Finally, choose suitable $r_{i,j}^{(p)}$ for all honest $\mathcal{V}_p$ to create valid sharings $\langle r_{i,j} \rangle_t$.

If the protocol enters **Abort**, then $\mathcal{S}$ follows **Abort** honestly but aborts the simulation when a dishonest party provides a forged signature in Step 1 of **Abort**. Additionally, it adds to C_A any dishonest party that sent incorrect $T_\ell^{(i)}$ or T_ℓ if $\mathcal{P} \in \mathcal{D}$, as identified in **Abort**.

If $C_A \neq \emptyset$ then $\mathcal{S}$ sends (Abort, C_A) to $\mathcal{F}_{\text{Prep}}^{t,n}$, independent if **Abort** of the protocol was entered or not. Otherwise, it computes $\langle s_i \rangle_t$ as parties would do in the protocol and sends the shares of the dishonest parties to $\mathcal{F}_{\text{Prep}}^{t,n}$.

Indistinguishability. We first observe that the shares of the honest parties which the environment obtains from $\mathcal{F}_{\text{Prep}}^{t,n}$ are consistent with those of the dishonest parties if the simulation finishes successfully. This is because if $\mathcal{P}$ is corrupted then the shares will be consistent with the s_i, while they are otherwise consistent with the s_i unknown to the adversary during the protocol run as the adversary does not have enough shares to reconstruct (and $\mathcal{F}_{\text{Prep}}^{t,n}$ chooses the shares of the honest parties accordingly). Moreover, $\mathcal{S}$ always aborts $\mathcal{F}_{\text{Prep}}^{t,n}$ if the adversary provides inconsistent shares to honest parties or if they provably send visibly incorrect $T_\ell^{(i)}, T_\ell$. We now show through a sequence of hybrids that the output of $\mathcal{S}$ when interacting with the dishonest parties is indistinguishable from the real protocol running with the dishonest parties.

Define the output of the simulation as H_0 and let H_1 be exactly like H_0, but where dishonest $\mathcal{V}_v$ that send invalid $r_{v,j}$ to an honest $\mathcal{P}$ are only added to C_A if the protocol actually enters **Abort**. By Lemma 6.3, these two hybrids are indistinguishable except with probability $2^{-k\rho}$.

Let H_2 be the same hybrid as H_1, but where dishonest $\mathcal{V}_v$ are only added to C_A if they were identified to have sent incorrect sharings in **Abort**. By Lemma 6.2, these two hybrids are indistinguishable except with probability $2^{-k\rho}$.

Observe that in the computation of C_A, only dishonest parties are contained and the simulation would abort. Now, let H_3 be the same as H_2 but where the simulation does not abort. As abort of the simulation happens iff the adversary succeeds in forging a signature, any distinguisher of H_2 and H_3 can be used to successfully break the unforgeability of Sign. Finally, observe that the distribution of the shares of the honest parties, the identified corrupted parties as well as the abort events are identical between H_3 and the protocol. □

Protocol $\Pi^{\mathsf{t,n}}_{\mathsf{Prep}}$ (cont.)

Distribute Shares:

1. Each party $\mathcal{V}_i \in \mathcal{V}$ executes the following protocol:
   (a) For $j \in [\lceil (n_S + \rho)/(n-t) \rceil]$ do
      i. Sample $r_{i,j} \in \mathbb{F}_{2^k}$ and generate a sharing $\langle r_{i,j} \rangle_t$.
      ii. Send $(r^{(p)}_{i,j}, \mathsf{Sign}(\mathfrak{sk}_i, r^{(p)}_{i,j}))$ to $\mathcal{V}_p$ for $p \neq i$. Note this is done as a single message for all j values needed.
      iii. Send $(r_{i,j}, \mathsf{Sign}(\mathfrak{sk}_i, r_{i,j}))$ to $\mathcal{P}$, again this is done as a single message for all j values needed.
      iv. On receiving $(r^{(i)}_{p,j}, \sigma^{(i)}_{p,j}) = (r^{(i)}_{p,j}, \mathsf{Sign}(\mathfrak{sk}_p, r^{(i)}_{p,j}))$ from party $\mathcal{V}_p$, verify the signature. If the signature $\sigma^{(i)}_{p,j}$ does not hold or if $\mathcal{V}_p$ did not send any message at all
         A. Broadcast $(\mathsf{Complaint}, i, \mathcal{V}_p)$.
         B. Upon receiving $(\mathsf{Complaint}, i, \mathcal{V}_p)$ party $\mathcal{V}_p$ publicly sends $(r^{(i)}_{p,j}, \sigma)$ to all parties, who forward it to $\mathcal{V}_i$.
      v. Similarly, do the same for the signatures that $\mathcal{P}$ should obtain.
2. For $\ell \in [\rho]$ do as follows.
   (a) Execute $(\alpha_{1,j,\ell}, \ldots, \alpha_{n,j,\ell}) \leftarrow \mathcal{F}_{\text{Rand}}(\{\mathcal{V}_1, \ldots, \mathcal{V}_n, \mathcal{P}\}, n, \mathbb{F}_{2^k})$.
   (b) Compute $T^{(i)}_\ell \leftarrow \sum_j \sum_{v \in [n]} \alpha_{v,j,\ell} \cdot r^{(i)}_{v,j}$ and broadcast $T^{(i)}_\ell$.
   (c) The prover $\mathcal{P}$ computes $T_\ell \leftarrow \sum_j \sum_{v \in [n]} \alpha_{v,j,\ell} \cdot r_{v,j}$ and broadcasts T_ℓ.
   (d) If the $T^{(i)}_\ell$ do not form a valid degree-t sharing of T_ℓ then go to **Abort(ℓ)**.
3. For $j \in \lceil n_S/(n-t) \rceil$ do
   (a) $c \leftarrow (j-1) \cdot (n-t)$.
   (b) The prover $\mathcal{P}$ computes and outputs $(s_{1+c}, \ldots, s_{n-t+c})^T = M_t \times (r_{1,j}, \ldots, r_{n,j})^T$,
   (c) $\mathcal{V}_i \in \mathcal{V}$ compute and output $(\langle s_{1+c} \rangle_t, \ldots, \langle s_{n-t+c} \rangle_t)^T = M_t \times (\langle r_{1,j} \rangle_t, \ldots, \langle r_{n,j} \rangle_t)^T$.

Figure 6.6: Protocol for preprocessing with $t < n/2$ (continued)

Protocol $\Pi_{4\text{t}}$

Let C be the circuit to be proved; the prover $\mathcal{P}$ is assumed to know an input witness w such that $C(w) = 0$.
Let n_S denote the number of AND gates in the circuit, n_W the length of the witness w and ρ a positive integer.
Let CheckMult and OutputRec be two additional flags initially set to $\top$ and $\perp$ respectively.

Init: Call $\mathcal{F}^{t,n}_{\text{Prep}}$, so that $\mathcal{P}$ obtains s_i and the verifiers $\mathcal{V}_1, \dots, \mathcal{V}_n$ obtain $\langle s_i \rangle_t$ for $i \in [n_S + n_W + 3\rho]$, i.e. $\mathcal{V}_j$ obtains $s_i^{(j)}, j \in [n]$. Set $x_j = s_{j+n_W+n_s+\rho}$ and $y_j = s_{j+n_W+n_s+2\rho}, j \in [\rho]$.

Prove: The prover "evaluates" the circuit as follows:

1. Compute the difference between the input wire values w_i and the pre-processed values s_i, i.e. $w_i - s_i, i \in [n_W]$.
2. Evaluate the circuit gate-by-gate:
   (a) For every linear gate, simply compute the resulting wire value
   (b) For each AND gate, compute the resulting wire value $c_j \leftarrow a_j \cdot b_j$ and $c_j - s_{j+n_W}, j \in [n_S]$.
   (c) Compute ρ additional random triples as $x_j \cdot y_j = z_j$, and $z_j - s_{j+n_W+n_S}$, $j \in [\rho]$
3. Set the proof to be the concatenation of all the values $\{w_i - s_i\}_{i\in[n_W]}$, $\{c_j - s_{j+n_W}\}_{j\in[n_S]}$, and $\{z_j - s_{j+n_W+n_S}\}_{j\in[\rho]}$.

Figure 6.7: Protocol $\Pi_{4\text{t}}$ for $t < n/4$

6.5 Distributed proof with $t < n/4$ corruptions

In this section we describe a protocol which deals with $t < n/4$ corruptions of the verifiers, i.e. Γ_C, Γ_S and Γ_Z are access structures consisting of all sets with more than $n - t$ verifiers in them. The protocol $\Pi_{4\text{t}}$, given in Fig. 6.7, forms the basis of our following protocol in the case of $t < n/3$, indeed it shares the same pre-processing phase from the previous section.

In the setting where we have $t < n/4$ corruptions we can rely on the Reed-Solomon decoding to robustly open secret sharings of degree up to $2t$. Thus, we can efficiently verify multiplications. We assume the statement to be verified is given by a circuit C over $\mathbb{F}_{2^k}$ which will evaluate to zero only on input of the witness w, i.e. $C(w) = 0$.

Given the values $\vec{s}$ generated in pre-processing, the prover can trivially "commit" to the witness w as well as the outputs of all the multiplication gates of C by broadcasting the difference between $\vec{s}$ and these values towards the verifiers. The verifiers can then evaluate the circuit as follows: to obtain the wire output values of a gate, they can either simply apply the corresponding linear operation directly on their shares, or obtain a sharing for the output wire from the prover's broadcast for multiplications. After evaluating the entire circuit in this manner, the verifiers can robustly open $\langle C(w)\rangle_t$ and verify it correctly evaluates to zero.

The verifiers also have to check that the commitments the prover provided for the outputs of the multiplication gates are consistent. For each verifier $\mathcal{V}_i$, let $a_j^{(i)}$ be the share of the left input corresponding to the jth multiplication/AND gate, $j \in [n_S]$. Correspondingly, $b_j^{(i)}$ is the share for the right input and $c_j^{(i)}$ for the output. Then $c_j^{(i)} = a_j^{(i)} \cdot b_j^{(i)}$ is a degree $2 \cdot t$ sharing of the value $c_j = a_j \cdot b_j$ output by this multiplication gate. We represent this sharing by $\langle c_j\rangle_{2\cdot t}$. The proof proceeds by verifying that the values held in $\langle c_j\rangle_{2\cdot t}$ are identical with the values held in $\langle c_j\rangle_t = \langle s_j\rangle_t - (s_j - c_j)$, and provided by the prover, therefore checking that all committed multiplication gate outputs were correct.

To achieve this, the verifiers check that a random linear combination over all products of the inputs corresponds to the same linear combination over the gate outputs. More precisely, for each multiplication gate $j \in [n_S]$, the verifiers sample a uniformly random multiplier β_j and locally compute shares $A^{(i)} = \sum_j \beta_j \cdot a_j^{(i)} \cdot b_j^{(i)}$, and $C^{(i)} = \sum_j \beta_j \cdot c_j^{(i)}$. Then, since $t < n/4$, the verifiers reliably reconstruct $\langle A\rangle_{2t}$ and $\langle C\rangle_t$. If $A = C$ then the verifiers accept the proof, otherwise they reject. Cheater identification can be achieved in a straightforward manner thanks to the error correction during the robust reconstruction. Moreover, the check is made zero-knowledge by letting $\mathcal{P}$ share additional valid random multiplication triples.

Theorem 6.2

If $t < n/4$, then protocol $\Pi_{4\mathrm{t}}$ secure implements the functionality $\mathcal{F}_{\mathrm{DV-ZK}}$ in the $(\mathcal{F}_{\mathrm{Prep}}^{t,n}, \mathcal{F}_{\mathrm{Rand}})$-hybrid model with $\Gamma_C = \Gamma_S = \Gamma_Z$ being the set of all subsets of verifiers of size $n - t$ or more, except with probability $1/|\mathbb{F}|$.

In the proof, we use the following lemma.

Lemma 6.5

Let $\langle x_j\rangle_t, \langle y_j\rangle_t, \langle z_j\rangle_{2t}, \langle z_j\rangle_t, \langle a\rangle_t, \langle b\rangle_t, \langle c\rangle_{2t}, \langle c\rangle_t$ the inputs of the multiplications check. If either $x_j \cdot y_j \neq z_j$, for some $j \in [n_S]$, or $a \cdot b \neq c$, then $T \neq 0$, except with probability $\frac{1}{|\mathbb{F}|}$.

Proof. (of Lemma 6.5) We recall that $\langle z_j\rangle_t = \langle s_j\rangle_t - (s_j - z_j)$, $j \in [n_S]$, and $\langle c\rangle_t = \langle s\rangle_t - (s - c)$, where $\langle s_j\rangle_t$ and $\langle s\rangle_t$ are correct sharings provided by the preprocessing functionality. Let $f_{j,t}(\cdot), g_{j,t}(\cdot), s_{j,t}(\cdot)$ be the unique t-degree polynomials such that, for $j \in [n_S]$,

$$f_{j,t}(i) = x_j^{(i)}, \quad f_{j,t}(0) = x_j,$$

$$g_{j,t}(i) = y_j^{(i)}, \quad g_{j,t}(0) = y_j,$$

$$s_{j,t}(i) = s_j^{(i)}, \quad s_{j,t}(0) = s_j,$$

and $p_t(\cdot), q_t(\cdot), s_t(\cdot)$ the unique t-degree polynomials such that

$$p_t(i) = a^{(i)}, \quad p_t(0) = a.$$

$$q_t(i) = b^{(i)}, \quad q_t(0) = b,$$

$$s_t(i) = s^{(i)}, \quad s_t(0) = s.$$

Then the shares $A^{(i)}$ and $C^{(i)}$ are given by

$$\sum_{j\in[n_S]} \beta_j \cdot (f_{j,t}(i) \cdot g_{j,t}(i)) + \beta_{n_S+1} \cdot (p_t(i) \cdot q_t(i))$$

and

$$\sum_{j\in[n_S]} \beta_j \cdot (s_{j,t}(i) - (s_j - z_j)) + \beta_{n_S+1} \cdot (s_t(i) - (s - c)).$$

If all the triples are correct, then $A - C = T = 0$.

Otherwise, suppose that $f_{j,t}(0) \cdot g_{j,t}(0) = \tilde{z}_j$ and $p_t(0) \cdot q_t(0) = \tilde{c}$ with $\tilde{z}_j = z_j + \delta_j$ and $\tilde{c} = c + \delta_{n_S+1}$. Then the reconstructed value T is given by

$$A - C = \sum_{j\in[n_S]} \beta_j(\tilde{z}_j - z_j) + \beta_{n_S+1}(\tilde{c} - c)$$

$$= \sum_{j\in[n_S]} \beta_j \cdot \delta_j + \beta_{n_S+1} \cdot \delta_{n_S+1},$$

where not all δ_j's are zero. Let $\vec{b} = (\beta_1, \ldots, \beta_{n_S}, \beta_{n_S+1})$ and $\vec{d} = (\delta_1, \ldots, \delta_{n_S}, \delta_{n_S+1})$, and consider the linear map $f_d = \vec{d} \cdot \vec{b}^T$. The probability that T is zero is equal to the probability that $\vec{b} \in \ker(f_d)$. Since $\dim(\ker(f_d)) = n_S$, and $\vec{b}$ is random and unknown to $\mathcal{A}$ when they choose $\vec{d}$, the probability that $\vec{b} \in \ker(f_d)$ is $\frac{|\mathbb{F}|^{n_S}}{|\mathbb{F}|^{n_S+1}} = \frac{1}{|\mathbb{F}|}$. □

Proof. (of Theorem 6.2) The simulator $\mathcal{S}$ obtains as input from the environment the set $\mathcal{D}$ of corrupted parties and forwards $(\mathsf{Corrupt}, \mathcal{D})$ to the functionality. On input (Init) from $\mathcal{F}_{\mathrm{DV-ZK}}$, if $\mathcal{A}$ sends Abort, it forwards Abort to the functionality, otherwise it forwards $(\mathtt{OK})$. $\mathcal{S}$ sets up a copy of $\mathcal{F}_{\mathrm{Rand}}$.

$\mathcal{S}$ emulates $\mathcal{F}^{t,n}_{\mathrm{Prep}}$ obtaining s_i and the $s_i^{(j)}$, for $i \in [n_S + n_W + 3\rho]$, held by the corrupted parties. Since $\mathcal{V}_{\mathcal{D}} \in \Delta_Z$, it receives (Prove, x) from the functionality. If $\mathcal{P} \in \mathcal{H}$, then it randomly samples the shares of the proof for corrupted parties, and sets honest shares consistently, i.e., such that the multiplication values are correct and $o = 0$; otherwise it receives from $\mathcal{A}$ the proof, consisting of the masked input values and masked multiplication values for AND gates and ρ masked random triples. In this case, $\mathcal{S}$ reconstructs the input $\tilde{w}$ and forwards $(\mathsf{Prove}, x, \tilde{w})$ to the functionality. The simulator $\mathcal{S}$ starts the simulation of the verification step, i.e. it evaluates the circuit honestly, for each gate computes the shares held by corrupted parties and sends the shares of the honest parties needed to run $\mathsf{RobustReconstruct}(\langle o \rangle_t, t)$. Receiving the shares of corrupted parties, it checks if those shares are the same as the ones computed by $\mathcal{S}$. We distinguish two different cases:

- If $\mathcal{P} \in \mathcal{H}$: The simulator $\mathcal{S}$ sets the flag accept. If the shares are consistent then $C_A = \emptyset$; else, if the shares are inconsistent, it identifies the cheating verifiers with incorrect shares and updates C_A with those parties.
- If $\mathcal{P} \notin \mathcal{H}$: if either $(x, w) \notin \mathcal{R}$ or some of the multiplication values given by the prover are incorrect, it sets the flag reject. If some of the shares are inconsistent, then $\mathcal{S}$ identifies the cheaters and updates C_A.

After this, $\mathcal{S}$ emulates the Multiplications check. To do this, it obtains random $\beta_1, \ldots, \beta_{n_S+1}$ from $\mathcal{F}_{\mathrm{Rand}}$, and sends these values to $\mathcal{A}$. If at any time $\mathcal{F}_{\mathrm{Rand}}$ sends (Abort, C_A), the simulator forwards (Abort, C_A) to the functionality. It also sends to $\mathcal{A}$ the honest shares $A^{(i)}, C^{(i)}, i \in \mathcal{H}$, necessary to run $\mathsf{RobustReconstruct}$, and receives from $\mathcal{A}$ the values $A^{(j)}, C^{(j)}, j \in \mathcal{D}$. If some of these shares are incorrect, it updates C_A with the corresponding corruptions.

Finally, if the flag accept or reject is true, $\mathcal{S}$ sends $(\mathsf{Abort}, 1, C_A)$ to $\mathcal{F}_{\mathrm{DV-ZK}}$, otherwise it sends $(\mathsf{Abort}, 0, C_A)$ to the functionality.

Indistinguishability. We now argue indistinguishability of the real and ideal executions to an environment, $\mathcal{Z}$. Recall that $\mathcal{Z}$ chooses the inputs of all parties. The view of $\mathcal{Z}$ in the real world then consists of these inputs, the messages received by the adversary and all the output values.

Indistinguishability of the proof follows from the privacy of Shamir's secret sharing scheme and from the fact that the input and the multiplication values are masked by the preprocessed values s_i, that are unknown to the adversary if the prover is honest. The messages received by the adversary in the multiplications check are randomized by a triple x, y, z, different for each of the ρ executions and randomly chosen by the simulator, if $\mathcal{P}$ is honest, and unknown to $\mathcal{Z}$. From this and privacy of Shamir's sharings, simulation of these messages is perfect.

To argue indistinguishability of the output, we distinguish two cases as follows.

- If $\mathcal{P} \in \mathcal{H}$, the simulator always accepts the proof and outputs $(\mathsf{Abort}, 1, C_A)$ to the functionality, where the set $C_A = \emptyset$ if all the shares provided by $\mathcal{A}$ are correct and consistent. Irrespective of what the adversary does, $\mathsf{RobustReconstruct}$ always reconstructs the correct values, even with $\mathsf{flag} = (\mathsf{incorrect}, C)$, since $t < n/4$. In the ideal execution, the simulator outputs the set of parties that provided incorrect shares, in the real one this same set is provided by $\mathsf{RobustReconstruct}$. Indeed, since the sharing is correct, it is possible to efficiently and correctly detect all the $t < n/4$ possible errors. Hence, in this case the simulation is perfect.

- If $\mathcal{P} \notin \mathcal{H}$, the simulator honestly evaluates the circuit with inputs extracted from the masked proof given by $\mathcal{A}$. Therefore, if $(x, \tilde{w}) \notin \mathcal{R}$ and the multiplication values that are part of the proof are correct, then $\mathcal{S}$ rejects the proof by sending $(\mathsf{Abort}, 1, C_A)$ and the outputs of the two executions are identical.

  If $(x, \tilde{w}) \notin \mathcal{R}$ and the multiplication values are incorrect, then the simulator rejects the proof by sending $(\mathsf{Abort}, 1, C_A)$, whereas in the real execution the probability of acceptance is given by Lemma 6.5.

  Since this test is repeated ρ times, the probability of passing the multiplications check with incorrect inputs is $(\frac{1}{|\mathbb{F}|})^{\rho}$. Finally, we note that also in this case the set C_A of corrupted verifiers given by the simulation and the protocol are identical and only consists of dishonest parties: while $\mathcal{S}$ can directly check inconsistent shares, in a real execution this set is guaranteed to be correct by the correctness of the sharing provided by $\mathcal{F}^{t,n}_{\text{Prep}}$.

$\square$

Protocol Π_{4t} (cont.)

Verify: The verifiers $\mathcal{V}_1, \dots, \mathcal{V}_n$ jointly check the circuit evaluation:

1. Evaluate the circuit within the Shamir secret sharing, computing a share of the output wire $\langle o \rangle_t$:
   (a) Shares of the input wires can be computed as $\langle w_i \rangle_t \leftarrow \langle s_i \rangle_t + (w_i - s_i)$ for $i \in [n_W]$.
   (b) Shares of the output wire values for an AND gate can be computed as
   $$\langle c_j \rangle_t \leftarrow \langle s_{j+n_W} \rangle_t + (c_j - s_{j+n_W}), \quad \text{for } j \in [n_S].$$
   (c) A degree $2 \cdot t$ sharing $\langle c_j \rangle_{2 \cdot t}$ of this same value is computed by each $\mathcal{V}_i$ as $c_j^{(i)} \leftarrow a_j^{(i)} \cdot b_j^{(i)}$.
   (d) Linear gates can be evaluated linearly over the shares in the degree t sharing.
   (e) Recompute $\langle x_i \rangle_t = \langle s_{i+n_W+n_s+\rho} \rangle_t, \langle y_i \rangle_t = \langle s_{i+n_W+n_s+2\rho} \rangle_t$ and $\langle z_i \rangle_t \leftarrow \langle s_{i+n_W+n_s} \rangle_t + (z_i - s_{i+n_W+n_s})$. Furthermore, compute a degree-$2t$ sharing of z_i by locally multiplying the shares of x_i, y_i as in Step 1c.
2. The verifiers call $\mathsf{RobustReconstruct}(\langle o \rangle_t, t)$, to obtain (o, flag_o).
3. If $o \neq 0$:
   - If $\mathsf{flag}_o = (\mathsf{correct}, \emptyset)$ then output Fail.
   - If $\mathsf{flag}_o = (\mathsf{incorrect}, C_o)$, then output the dishonest verifiers in C_o and Fail.
4. Else, set $\mathsf{OutputRec} = \top$. If $\mathsf{flag}_o = (\mathsf{incorrect}, C_o)$, identify the dishonest verifiers in C_o.
5. *Multiplications check:* Verifiers repeat ρ times the following.
   (a) Call $(\beta_1, \dots, \beta_{n_S+1}) \leftarrow \mathcal{F}_{\text{Rand}}(\{\mathcal{V}_1, \dots, \mathcal{V}_n\}, n_S + 1, \mathbb{F}_{2^k})$.
   (b) Compute $\langle A \rangle_{2t} = \sum_{j \in [n_S]} \beta_j \cdot \langle c_j \rangle_{2t} + \beta_{n_S+1} \cdot \langle z_i \rangle_{2t}$ and $\langle C \rangle_t = \sum_{j \in [n_S]} \beta_j \cdot \langle c_j \rangle_t + \beta_{n_S+1} \cdot \langle z_i \rangle_t$
   (c) Run $\mathsf{RobustReconstruct}(\langle A \rangle_{2t} - \langle C \rangle_t, t)$, to obtain (T, flag_T)
   (d) If $T \neq 0$, set $\mathsf{CheckMult} = \bot$. Moreover,
      - If $\mathsf{flag}_T = (\mathsf{correct}, \emptyset)$, then output Fail.
      - If $\mathsf{flag}_{T_v} = (\mathsf{incorrect}, C_M)$, then identify the dishonest verifiers in flag_{T_v} and output Abort
6. If both $\mathsf{CheckMult} = \top$ and $\mathsf{OutputRec} = \top$, accept the proof and identify possible dishonest verifiers $C_A = C_o \cup \{C_{M_v}\}_{v \in [\rho]}$

Figure 6.8: Protocol Π_{4t} for $t < n/4$ (continued)

Protocol-Π_{3t} (Init)

Let C be the circuit to be proved; the prover $\mathcal{P}$ is assumed to know an input witness w such that $C(w) = 0$.
Let n_S denote the number of AND gates in the circuit, n_W the length of the witness w, and $\sigma = \lceil \log_2 n_S \rceil$. Let $K = \mathbb{F}_{2^k}$ and $L = \mathbb{F}_{2^{\rho \cdot k}}$, with $\phi : K \to L$ the field homomorphism that embeds K in L. The protocol uses a hash function modelled as a random oracle. For secret sharings in L, let the evaluation points of verifier $\mathcal{V}_i$ be $\phi(i \in K)$.

Init: Call $\mathcal{F}_{\text{Prep}}^{t,n}$ in K, so that $\mathcal{P}$ obtains s_i and the verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ obtain $\langle s_i \rangle_t$ for $i \in [n_S + n_W]$, i.e. $\mathcal{V}_j$ obtains $s_i^{(j)}, j \in [n]$. Call $\mathcal{F}_{\text{Prep}}^{t,n}$ in L, so that $\mathcal{P}$ obtains S_i and the verifiers obtain $\langle S_i \rangle_t$ for $i \in [3 + 2 \cdot \sigma]$. All parties obtain a random string $\nu \in \mathbb{F}_2^{\lambda}$.

Figure 6.9: Protocol Π_{3t} for $t < n/3$ (**Init**)

6.6 Distributed proof with $t < n/3$ corruptions

The general approach for this setting will be very similar to the case $t < n/4$ described in the previous section. The main difference is that now we can no longer robustly reconstruct a degree $2t$ polynomial, so we will instead rely on the Schwartz-Zippel lemma to check the correctness of the multiplications. More precisely, we use a checking method similar to the one used in [BBC+19, BMRS21]. We first transform the n_S multiplication gates into an inner product triple by taking a random linear combination and updating the left inputs to the multiplications correspondingly. Given a challenge from the verifiers, this operation is entirely local.

This inner product triple is then repeatedly compressed by applying the Schwartz-Zippel lemma, until only a final, single multiplication triple remains. This final triple can be checked by the verifiers by robustly opening it. The prover adds an extra random multiplication to preserve the zero-knowledge property in this process. To avoid $\log n_S$ rounds of communication between the prover and the verifiers, we apply the Fiat-Shamir transform to make the process of proving non-interactive. We also use the Fiat-Shamir transform to compute the initial re-randomization factors. For this to work we apply a minor change to the preprocessing functionality $\mathcal{F}_{\text{Prep}}^{t,n}$, so that it additionally outputs a random string $\nu \in \mathbb{F}_2^{\lambda}$ to all parties $\mathcal{P}, \mathcal{V}_1, \ldots, \mathcal{V}_n$. This is used in the random oracle to bind the statement and the proof to this value. The compression itself is performed as follows. Assume we have an inner product triple $((x_i)_{1\leq i\leq N}, (y_i)_{1\leq i\leq N}, z)$, such that $z = \sum_{i=1}^{N} x_i \cdot y_i$, and that N is a multiple of two (otherwise, we implicitly pad the x_i and y_i by zeroes). The prover then interpolates N polynomials of degree 1, $f_k(x)$ and $g_k(x)$, such that $f_k(j) = x_{2\cdot k+j}$ and $g_k(j) = y_{2\cdot k+j} = y_j$, for $j \in [2]$. Furthermore, define the polynomial of degree 2 $h(x) = \sum_{k=1}^{\frac{N}{2}} f_k(x) \cdot g_k(x) = h_1 + h_2 \cdot x + h_3 \cdot x^2$. Observe that by construction, $z = \sum_{j=1}^{2} h(j) = h_2 + h_3$. The prover commits to the coefficients of $h(x)$ so that the verifiers can evaluate it with only linear operations. Given the relation between z and those coefficients above, the verifiers can recover a commitment to h_3 from $\langle z\rangle_t$, and $\langle h_2\rangle_t$ through only linear operations, allowing the prover to eliminate a commitment to h_3 from the proof. The compressed inner product triple can now be obtained as $((f_k(r))_{1\leq k\leq N/2}, (g_k(r))_{1\leq k\leq N/2}, h(r))$ for a randomly chosen value of r.

To verify the proof, the verifiers check that the circuit output reconstructs to 0 and that the final multiplication triple is correct. The interpolation of $f_k(x)$ and $g_k(x)$ is linear, so the verifiers can perform the operation locally over the secret sharing, and with the shares of the coefficients of $h(x)$ the evaluation of all polynomials in r can also be performed locally.

Figure 6.9 describes the protocol in detail. To ensure the soundness of this protocol, the multiplication check and compression must be performed over a large enough finite field. It is however possible to keep the proof size small by performing the circuit evaluation over a smaller finite field $\mathbb{F}_{2^k}$ such that $2^k > n$ to allow for the secret sharing. The (shares of the) inputs and outputs of the multiplication gates are then lifted to an extension field $\mathbb{F}_{2^{\rho\cdot k}}$ to perform the multiplication check with sufficient soundness.

Theorem 6.3

Let $\mathcal{H}$ be a random oracle that maps into $\mathbb{F}_{2^{\rho\cdot k}}$. If $t < n/3$ then protocol $\Pi_{3\mathrm{t}}$ described in Fig. 6.9 securely implements the functionality $\mathcal{F}_{\mathrm{DV-ZK}}$ against a static adversary in the $(\mathcal{F}_{\mathrm{Prep}}^{t,n}, \mathcal{F}_{\mathrm{Rand}})$-hybrid model with $\Gamma_C = \Gamma_S = \Gamma_Z$ being the set of all subsets of verifiers of size $n-t$ or more, except with probability

$$\epsilon \cdot \log_2(n_S) + q(\epsilon + 2/|\mathcal{D}| + 2^{-\lambda})$$

where q is the number of random oracle queries made by a malicious prover, $\epsilon = 2^{-\rho\cdot k+2}$ and $|\mathcal{D}| = 2^{-\rho\cdot k}$.

Lemma 6.6

Let $a_\ell \cdot b_\ell \neq c_\ell$, for some $\ell \in [n_S]$, then the probability of passing the multiplication test if parties honestly perform the check is

$$\epsilon \cdot \log_2(n_S) + q(\epsilon + 2/|\mathcal{D}| + 2^{-\lambda})$$

where q is the number of random oracle queries made by a malicious prover, $\epsilon = 2^{-\rho \cdot k+2}$ is the round-by-round soundness and $|\mathcal{D}| = 2^{-\rho \cdot k}$ is the smallest challenge set in any given round of the proof.

Proof. (of Lemma 6.6) The proof follows from adapting Theorem 5 of [BMRS21]. There, the authors show that if the proof is an IP with LOVe with t rounds, 1 query and round-by-round soundness ϵ, then the compiled protocol that uses the Fiat-Shamir transform to compute the t challenges instead of having these chosen by the verifier has soundness error as defined in the statement of the lemma.

Our protocol is an IP with LOVe by observing that $\mathcal{F}^{t,n}_{\mathrm{Prep}}$ generates perfectly binding linearly homomorphic commitments (due to the reconstruction property), as used in the IP with LOVe model, and therefore permits the same queries by the verifier. As we essentially use the same proof protocol as [BMRS21] for evaluating the circuit, we obtain the same round-by-round soundness error ϵ and the same number of rounds t. Thus, by their theorem, we also obtain the same soundness error, except that we avoid their extra loss due to an adversarial guess of the MAC key of the verifier, as in our case the commitment is perfectly binding for a dishonest prover. □

We now prove theorem 6.3.

The simulator $\mathcal{S}$ obtains as input from the environment the set $\mathcal{D}$ of corrupted parties and forwards $(\mathsf{Corrupt}, \mathcal{D})$ to the functionality. Throughout the execution, $\mathcal{S}$ simulates the random oracle $\mathcal{H}$ by answering every new query with a random value from the relevant set and maintaining a list of past queries to answer repeated queries consistently. As in the real protocol, the simulator uses a deterministic expansion function that for each seed defines a distinct random tape.

The simulation is very similar to that of Theorem 6.2. On input (Init) from $\mathcal{F}_{\mathrm{DV-ZK}}$, if $\mathcal{A}$ sends Abort, it forwards Abort to the functionality, otherwise forwards $(\mathtt{OK})$. $\mathcal{S}$ emulates $\mathcal{F}^{t,n}_{\mathrm{Prep}}$ obtaining the values s_i and $s_i^{(j)}$, for $i \in [n_S + n_W + 2 \cdot n_1 + n_2 + 2]$, held by corrupted parties. Since $\mathcal{V}_\mathcal{D} \in \Delta_Z$, it receives (Prove, x) from the functionality. If $\mathcal{P} \in \mathcal{H}$, then it randomly samples values to be sent during **Prove**. In the process, it samples random R_j by honestly emulating the random oracle $\mathcal{H}$. If $\mathcal{P}$ is corrupted it receives from $\mathcal{A}$ the proof by extracting from the shares issued by $\mathcal{F}_{\text{Prep}}^{t,n}$. $\mathcal{S}$ reconstructs the input $\tilde{w}$ and forwards $(\mathsf{Prove}, x, \tilde{w})$ to the functionality.

The simulator $\mathcal{S}$ starts the simulation of the verification step, i.e. it evaluates the circuit honestly, for each gate computes the shares held by corrupted parties and sends the shares of the honest parties needed to run $\mathsf{RobustReconstruct}(\langle o\rangle_t, t)$. Here, if $\mathcal{P} \in \mathcal{H}$ it sends shares during $\mathsf{RobustReconstruct}$ of $\langle o\rangle_t$ that open it to 0. Receiving the shares of corrupted parties, it checks if those shares are the same as the ones computed by $\mathcal{S}$. We distinguish two different cases:

- If $\mathcal{P} \in \mathcal{H}$: The simulator $\mathcal{S}$ sets the flag accept. If the shares are consistent then $C_A = \emptyset$; else, if the shares are inconsistent, it identifies the cheating verifiers with incorrect shares and updates C_A with those parties.
- If $\mathcal{P} \notin \mathcal{H}$: if either $(x, w) \notin \mathcal{R}$ or some of the multiplication values given by the prover are incorrect, it sets the flag reject. If some of the shares are inconsistent, then $\mathcal{S}$ identifies cheaters and updates C_A.

It also sends to $\mathcal{A}$ the honest shares, necessary to run $\mathsf{RobustReconstruct}$ and receives from $\mathcal{A}$ the shares of dishonest verifiers. If some of these shares are incorrect, it updates C_A with the corresponding corruptions.

Finally, if the flag accept or reject is true, $\mathcal{S}$ sends $(\mathsf{Abort}, 1, C_A)$ to $\mathcal{F}_{\mathrm{DV-ZK}}$, otherwise it sends $(\mathsf{Abort}, 0, C_A)$ to the functionality.

Indistinguishability. We now argue indistinguishability of the real and ideal executions to an environment, $\mathcal{Z}$. Recall that $\mathcal{Z}$ chooses the inputs of all parties. The view of $\mathcal{Z}$ in the real world then consists of these inputs, the messages received by the adversary and all the output values.

Indistinguishability of the proof follows from the privacy of Shamir's secret sharing scheme and from the fact that the input and the multiplication values are masked by the preprocessed values s_i, that are unknown to the adversary if the prover is honest. All messages received by the adversary for the shares of h are also committed to using differences to random commitments by the simulator, if $\mathcal{P}$ is honest, and unknown to $\mathcal{Z}$. Moreover, the triple in the last round is random due to the inclusion of a random triple in the protocol. For the opening of o, the adversary does not have enough shares to distinguish its opening to 0 from the opening to the actual value that was shared by the simulator. From this and privacy of Shamir's sharings throughout the protocol, simulation of these messages is perfect.

To argue indistinguishability of the output, we distinguish two cases as follows.

- If $\mathcal{P} \in \mathcal{H}$, the simulator always accepts the proof and outputs $(\mathsf{Abort}, 1, C_A)$ to the functionality, where the set $C_A = \emptyset$ if all the shares provided by $\mathcal{A}$ are correct and consistent. Irrespective of what the adversary does, RobustReconstruct always reconstructs the correct values, even with $\mathsf{flag} = (\mathsf{incorrect}, C_A)$, since $t < n/3$ and the sharings are correct since they are obtained by calling the preprocessing functionality. In the ideal execution, the simulator outputs the set of parties that provided incorrect shares, in the real one this same set is provided by RobustReconstruct. Indeed, since the sharing is correct, it is possible to efficiently and correctly detect all the $t < n/3$ possible errors. Hence, in this case the simulation is perfect.

- If $\mathcal{P} \notin \mathcal{H}$, the simulator honestly evaluates the circuit with inputs extracted from the masked proof given by $\mathcal{A}$. Therefore, if $(x, \tilde{w}) \notin \mathcal{R}$ and the multiplication values that are part of the proof are correct, then $\mathcal{S}$ aborts the proof towards $\mathcal{F}_{\mathrm{DV-ZK}}$ and the outputs of the two executions are identical.

  If $(x, \tilde{w}) \notin \mathcal{R}$ and the multiplication values are incorrect, then the simulator rejects the proof, whereas in the real execution the probability of acceptance is given by applying Lemma 6.6.

  Finally, we conclude by observing that the set C_A of cheating verifiers is identical in both the executions and only consists of corrupted parties as this set in the protocol is given by running the Reed-Solomon reconstruction on a correct sharing with $t < n/3$.

Protocol-Π_{3t} (Prove)

The prover "evaluates" the circuit as follows:

1. Compute the difference between the input wire values w_i and the pre-processed values s_i, i.e. $w_i - s_i, i \in [n_W]$.
2. Evaluate the circuit gate-by-gate:
   (a) For every linear gate, simply compute the resulting wire value
   (b) For each AND gate, compute $c_j \leftarrow a_j \cdot b_j$ and $c_j - s_{j+n_W}, j \in [n_S]$. Let $A_j = \phi(a_j)$, $B_j = \phi(b_j)$ and $C_j = \phi(c_j)$.
3. Compute an additional random multiplication triple $(A, B, C) \in L^3$, and compute $A - S_1$, $B - S_2$, $C - S_3$.
4. Set π to be the concatenation of all committed values so far: $\{w_i - s_i\}_{i\in[n_W]}$, $\{c_j - s_{j+n_W}\}_{j\in[n_S]}$, $A - S_1$, $B - S_2$, $C - S_3$.
5. Randomize the multiplication triples $(A_j, B_j, C_j = A_j \cdot B_j)_j$ into an inner product triple $((R_j \cdot A_j)_j, (B_j)_j, \sum_j R_j \cdot C_j)$, where $n_S + 1$ random value R_j are sampled, seeded with a hash of (π, x, ν).
6. Compress the inner product triple σ times until a single multiplication triple remains. For $j \in [\sigma]$:
   (a) Parse the current inner product triple as $((X_k)_k, (Y_k)_k, Z), k \in [n_S/2^j]$
   (b) Interpolate the polynomials $f_k(x)$ and $g_k(x)$ such that $f_k(0) = X_{2\cdot k-1}$, $f_k(1) = X_{2\cdot k}$, $g_k(0) = Y_{2\cdot k-1}$ and $g_k(1) = Y_{2\cdot k}$.
   (c) Define $h(x) = \sum_k f_k(x) \cdot g_k(x) = h_1 + h_2 \cdot x + h_3 \cdot x^2$.
   (d) Append commitments to the coefficients h_1, h_2 of the polynomial $h(x)$ to π: $\{h_i - X_{3+2\cdot j+i}\}_{i\in[2]}$.
   (e) Obtain a random field element $T_j \in L$, seeded with a hash of the current value of π.
   (f) The inner product triple now becomes $((f_k(T_j))_k, (g_k(T_j))_k, h(T_j))$.

The proof consists of the final value of π.

Figure 6.10: Protocol Π_{3t} for $t < n/3$ (**Prove**)

Protocol-$\Pi_{3\mathrm{t}}$ (Verify)

The verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ jointly check the circuit evaluation:

1. Evaluate the circuit within the Shamir secret sharing on K, computing a share of the output wire $\langle o \rangle_t$:
   (a) Shares of the input wires can be computed as $\langle w_i \rangle_t \leftarrow \langle s_i \rangle_t + (w_i - s_i)$ for $i \in [n_W]$.
   (b) Shares of the output wire values for an AND gate $c_j = a_j \cdot b_j$ can be computed as
   $$\langle c_j \rangle_t \leftarrow \langle s_{j+n_W} \rangle_t + (c_j - s_{j+n_W}), \quad \text{for} \quad j \in [n_S].$$
   Let $\langle A_j \rangle_t = \phi(\langle a_j \rangle_t)$, $\langle B_j \rangle_t = \phi(\langle b_j \rangle_t)$ and $\langle C_j \rangle_t = \phi(\langle c_j \rangle_t)$.
   (c) Linear gates can be evaluated linearly over the shares in the degree t sharing.
2. The verifiers obtain A, B and C by similarly adding the commitment to the preprocessing shares.
3. The verifiers call $(o, \mathsf{flag}_o) \leftarrow \mathsf{RobustReconstruct}(\langle o \rangle_t)$.
   (a) If $o \neq 0$:
   - If $\mathsf{flag}_o = (\mathsf{correct}, \emptyset)$ then output Fail.
   - If $\mathsf{flag}_o = (\mathsf{incorrect}, C_o)$, then output the dishonest verifiers in C_o and Fail.

   (b) If $\mathsf{flag}_o = (\mathsf{incorrect}, C_o)$, identify the dishonest verifiers in C_o.
4. Obtain the same – secret-shared – randomization to an inner product triple as the prover, using the same random value R.
5. Perform the analog of the prover's compressions, for $j \in [\sigma]$:
   (a) Interpolate $\langle f_k(x) \rangle_t$ and $\langle g_k(x) \rangle_t$ similar to the prover.
   (b) The polynomial $\langle h(x) \rangle_t$ can be recovered from the commitment to its coefficients, together with $\langle h_2 \rangle_t = \langle Z \rangle_t - \langle h_2 \rangle_t$.
   (c) Update the inner product triple to be the compressed version with the same random T_j as the prover.
6. Let the final remaining multiplication triple be $(\langle X \rangle_t, \langle Y \rangle_t, \langle Z \rangle_t)$.
7. Call $(w, \mathsf{flag}_w) \leftarrow \mathsf{RobustReconstruct}(\langle w \rangle_t, t)$ for $w \in \{X, Y, Z\}$. If $\mathsf{flag}_w = (\mathsf{incorrect}, C_w)$, identify the cheaters in C_w.
8. If $X \cdot Y \neq Z$, Fail. Otherwise, accept the proof.

Figure 6.11: Protocol $\Pi_{3\mathrm{t}}$ for $t < n/3$ (**Verify**)

Protocol	Circuit	n	t	Field	Parameters	Number of preproc. element	Proof size (bytes)	Preprocessing Time (ms)	Prover Time (ms)	Verifier Time (ms)
Π_{4t}	AES	5	1	$\mathbb{F}_{2^3}$	$\rho = 14$	7000	2496	1.67	4.07	6.83
Π_{4t}	SHA-256	5	1	$\mathbb{F}_{2^3}$	$\rho = 14$	23000	8449	3.45	7.06	14.02
Π_{4t}	SHA x10	5	1	$\mathbb{F}_{2^3}$	$\rho = 14$	230000	84655	28.52	33.64	43.87
Π_{4t}	1M AND	5	1	$\mathbb{F}_{2^3}$	$\rho = 14$	1100000	375048	95.64	30.03	89.81
Π_{3t}	AES	4	1	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$6655 + 50$	2811	2.01	7.81	8.43
Π_{3t}	SHA-256	4	1	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$22530 + 35$	8808	3.41	22.83	24.24
Π_{3t}	SHA x10	4	1	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$225745 + 50$	85079	24.46	50.32	50.94
Π_{3t}	1M AND	4	1	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$1000128 + 50$	375516	98.89	180.07	200.27
Π_{3t}	AES	7	2	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$6655 + 50$	2811	2.98	7.86	8.93
Π_{3t}	SHA-256	7	2	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$22530 + 35$	8808	4.52	21.88	24.16
Π_{3t}	SHA x10	7	2	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$225745 + 50$	85079	25.16	54.23	80.28
Π_{3t}	1M AND	7	2	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^{87}}$	$1000128 + 50$	375516	113.79	187.56	212.69

Table 6.2: Experimental results for running the protocols in Figure 6.7 and Figure 6.9 on our evaluation circuits

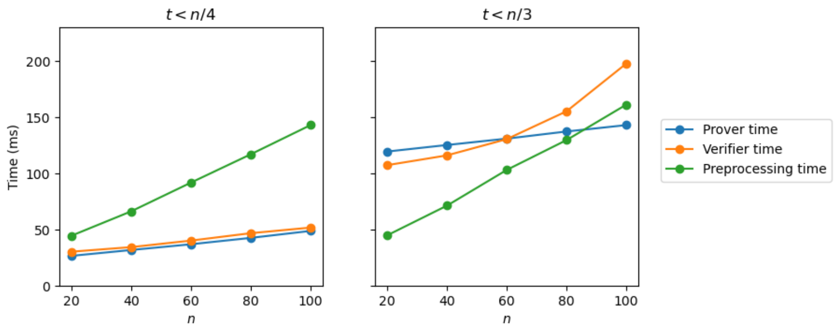


Figure 6.12: Timing on the 10-block SHA256 circuit with $t = \lfloor (n-1)/4 \rfloor$ and $t = \lfloor (n-1)/3 \rfloor$ respectively. The base field is $\mathbb{F}_{2^7}$ and for $t < n/3$ the extension field is $\mathbb{F}_{2^{91}}$.

6.7 Experiments

We implemented our protocols in C++[6] and tested with different circuits and number of verifiers.

For experiments with less than 10 parties, our tests were run on a cluster of computers running Ubuntu 20.04.2 with a ping time of roughly 0.6 ms, and a total bandwidth of 9.41Gbit/s per machine. The machines had either Intel i7-770K CPUs running at 4.2 GHz with 32 GB of RAM, or Intel i9-9900 CPUs running at 3.1 GHz with 128 GB of RAM. For the other experiments ($n > 10$),

[6]The implementation is publicly available at `https://github.com/KULeuven-COSIC/Feta`.

Protocol	Circuit	n	t	Field	Parameters	Proof size (bytes)	Preprocessing Time (ms)	Prover Time (ms)	Verifier Time (ms)
Π_{4t}	AES	100	20	$\mathbb{F}_{2^7}$	$\rho = 6$	$5,824$	33.26	2.36	10.60
Π_{4t}	SHA-256	100	20	$\mathbb{F}_{2^7}$	$\rho = 6$	$19,714$	42.09	5.47	13.39
Π_{4t}	SHA x10	100	20	$\mathbb{F}_{2^7}$	$\rho = 6$	$197,527$	166.35	46.63	51.28
Π_{4t}	1M AND	100	20	$\mathbb{F}_{2^7}$	$\rho = 6$	$875,112$	432.21	175.92	218.51
Π_{3t}	AES	100	30	$\mathbb{F}_{2^7}$	$\mathbb{F}_{2^{91}}$	$6,153$	71.55	5.56	68.06
Π_{3t}	SHA-256	100	30	$\mathbb{F}_{2^7}$	$\mathbb{F}_{2^{91}}$	$20,090$	70.20	15.34	80.52
Π_{3t}	SHA x10	100	30	$\mathbb{F}_{2^7}$	$\mathbb{F}_{2^{91}}$	$197,971$	181.13	135.09	215.65
Π_{3t}	1M AND	100	30	$\mathbb{F}_{2^7}$	$\mathbb{F}_{2^{91}}$	$875,602$	595.44	562.45	891.46

Table 6.3: Experimental results for $n = 100$ verifiers on our evaluation circuits

we utilized $n + 1$ machines on Amazon AWS. Each of these was an individual *c5.large* instance in the *eu-central-1* region, with a measured ping of roughly 0.5 ms, and 4.17Gbit/s bandwidth. Each configuration was run a total of 200 times and the median was taken to obtain the presented running times.

We present experimental validation of the efficiency of our protocols for small circuits by presenting prover and verification times for proving knowledge of an AES-128 key corresponding to a public plaintext-ciphertext pair and a boolean circuit proving the knowledge of a pre-image for the SHA-256 compression function. These functions where chosen as the boolean circuits for these are readily available, and well-studied. The AES-128 circuit has 6400 AND gates, while the SHA-256 circuit has 22573 AND gates. We also present results for a SHA-256 pre-image consisting of 10 512-bit blocks, which gives a circuit of 1,317,424 total gates and 220,369 AND gates, and a circuit consisting solely of one million AND gates, with 128 inputs. For all circuits and protocols we present results for a system tolerating a single corrupted verifier ($t = 1$) and a total of $n = 5$ verifiers in Table 6.2. For the protocol for $t < n/3$, we also provide numbers for $t = 2$ corruptions with $n = 7$ parties in total.

In Figure 6.12, we present results for $n \in \{20, 40, 60, 80, 100\}$ with the maximum value for t allowed by the protocols. Table 6.3 contains more detailed results of our experiments with $n = 100$ verifiers.

6.7.1 Results

Our experimental results are presented in Table 6.2 and Figure 6.12. We can immediately see that Π_{4t} is a small factor more efficient than Π_{3t}. Both protocols have runtimes that allow for practical deployment. Given that a threshold of $t < n/4$ may be enough in a number of practical situations, one can see that the more efficient Π_{4t} can be preferred.

We have already made some comparisons with other systems in Section

6.1. Notice that [DDOS19, BDK+21, DOT21] report comparable prover and verification times for AES, however these papers use a more compact description of the AES circuit over $\mathbb{F}_{2^8}$ with S-boxes instead of AND gates. We could utilize the same approach, obtaining better runtimes. However, our goal is different from the one in these papers as they specifically aim to obtain efficient post-quantum signature schemes based on AES, while we support general circuits, only using AES and SHA-256 as examples.

For protocol $\Pi_{4\mathrm{t}}$ from Figure 6.7, targeting $t < n/4$, we selected the finite field $\mathbb{F}_{2^3}$ to accommodate the secret sharing, when $n < 7$ and $\mathbb{F}_{2^7}$ for the other cases. We performed $\rho = 14$ (resp. $\rho = 6$) parallel repetitions of the protocol to boost the statistical security to 2^{-42}. When looking at the trade-off between the field size of $\mathbb{F}_{2^k}$ and the number of repetitions ρ for this protocol, notice that the security level will always be $2^{-k\cdot\rho}$, regardless of how we distribute the load across the two parameters. Similarly, the communication cost among the verifiers does not depend on either ρ or k individually, but only on the product $\rho \cdot k$. Using a larger field size, however, does increase the proof size and the communication cost of the preprocessing phase, as those only depend on the field size, and not on ρ. Hence, it should be preferred to use a smaller field with more parallel repetitions, rather than increasing the field size to target a security level for this protocol. The communication cost (in terms of amount of bytes sent by each verifier) in the verification protocol is $O(n)$ in the case of protocol $\Pi_{4\mathrm{t}}$.

For $\Pi_{3\mathrm{t}}$ in Figure 6.9, targeting $t < n/3$, we again choose to aim for a security level of $\mathsf{sec} = 40$ and let the maximum number of queries the prover can make to $\mathcal{H}$ be $q = 2^{40}$. Similar to the above case, we choose $\mathbb{F}_{2^k} = \mathbb{F}_{2^3}$ as the minimal field to accommodate for the secret sharing for $n \leq 7$ and $\mathbb{F}_{2^7}$ for the other cases; we let $\mathbb{F}_{2^{\rho\cdot k}} = \mathbb{F}_{2^{87}}$ be the extension field, when $n \leq 7$, and $\mathbb{F}_{2^{\rho\cdot k}} = \mathbb{F}_{2^{91}}$ for the other cases, to ensure soundness for all our evaluation circuits.

Large number of verifiers. Table 6.3 and Figure 6.12 show that increasing the number of parties has a small impact on proof and verification time for protocol $\Pi_{4\mathrm{t}}$, while the change is a little more evident in $\Pi_{3\mathrm{t}}$. There is only a small difference in the pre-processing execution between our two protocols. Our $\Pi_{4\mathrm{t}}$ protocol can prove circuits of 1 million AND gates in less than 176/220ms for proof and verification, respectively, with 100 verifiers; while our $\Pi_{3\mathrm{t}}$ requires proof/verification time of 563/892ms for the same circuit and number of verifiers. In both protocols the proof size is ≈ 0.8MB.

Communication cost. The communication cost is $O(n)$ in the case of protocol $\Pi_{3\mathrm{t}}$; which scales linearly with the number of verifiers, however, importantly it is sublinear in the total circuit size. Note, the threshold t has no effect on the round or total communication cost, it only increases the computational cost to perform a robust opening. Also note, for small n, the computation time mostly dominates for the prover, and we see only little impact of a growing number of verifiers on the prover time. When the number of verifiers grows larger, the communication starts to dominate. For the verifiers, similarly the increased amount of communication partners takes its toll, along with an increased computational cost for robust reconstruction when t starts to grow.

For our preprocessing protocol for any $t < n/2$, the dominant cost is each verifier sending $n_S/(n-t)$ shares to every other verifier, and the prover. Therefore, if t is a constant fraction of n, the communication per verifier is linear in the circuit size but essentially independent of n. For instance, with $t = n/3$ it is roughly $\frac{3}{2} \cdot n_S$ field elements, and for $t = n/4$ this becomes $\frac{4}{3} \cdot n_S$.

Acknowledgements

We thank Pratik Sarkar for identifying a bug in an earlier version. This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) under contract HR001120C0085, by the FWO under an Odysseus project GOH9718N, by CyberSecurity Research Flanders with reference number VR20192203, by the Aarhus University Research Foundation, and by the Independent Research Fund Denmark under project number 0165-00107B.

Bibliography

[ACF02] Masayuki Abe, Ronald Cramer, and Serge Fehr. Non-interactive distributed-verifier proofs and proving relations among commitments. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 206–223. Springer, Berlin, Heidelberg, December 2002.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[AKP22] Benny Applebaum, Eliran Kachlon, and Arpita Patra. Verifiable relation sharing and multi-verifier zero-knowledge in two rounds: Trading NIZKs with honest majority. Cryptology ePrint Archive, Report 2022/167, 2022.

[BBC$^+$19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Cham, August 2019.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Cham, August 2019.

[BCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Berlin, Heidelberg, August 2013.

[BD91] Mike Burmester and Yvo Desmedt. Broadcast interactive proofs (extended abstract). In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 81–95. Springer, Berlin, Heidelberg, April 1991.

[BDK$^+$21] Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 266–297. Springer, Cham, May 2021.

[Bea91] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, January 1991.

[BGKW88] Michael Ben-Or, Shafi Goldwasser, Joe Kilian, and Avi Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *20th ACM STOC*, pages 113–131. ACM Press, May 1988.

[BKZZ20] Foteini Baldimtsi, Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. Crowd verifiable zero-knowledge and end-to-end verifiable multiparty computation. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 717–748. Springer, Cham, December 2020.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Cham.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Berlin, Heidelberg, March 2008.

[CDG$^+$17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[DDOS19] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P. Smart. BBQ: Using AES in picnic signatures. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 669–692. Springer, Cham, August 2019.

[DOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021.

[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 ACM STOC*, pages 218–229. ACM Press, May 1987.

[HBHW16] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[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 PODC*, pages 25–34. ACM, August 1997.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[KGC$^+$18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[KZF$^+$18] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. Commit-Chains: Secure, scalable off-chain payments. Cryptology ePrint Archive, Report 2018/642, 2018.

[LMs05] Matt Lepinski, Silvio Micali, and abhi shelat. Fair-zero knowledge. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 245–263. Springer, Berlin, Heidelberg, February 2005.

[LSTW21] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/030, 2021.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[WZC$^+$18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 675–692. USENIX Association, August 2018.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[YW22] Kang Yang and Xiao Wang. Non-interactive zero-knowledge proofs to multiple verifiers. Cryptology ePrint Archive, Report 2022/063, 2022.

CHAPTER 7

ZK-for-Z2K: MPC-in-the-Head Zero-Knowledge Proofs for $\mathbb{Z}_{2^k}$

Lennart Braun[1], Cyprien Delpech de Saint Guilhem[2], Robin Jadoul[2], Emmanuela Orsini[3], Nigel P. Smart[2,4], and Titouan Tanguy[4]

[1]Department of Computer Science, Aarhus University, Aarhus, Denmark,
[2]COSIC, KU Leuven, Leuven, Belgium,
[3]Department of Computing Sciences, Bocconi University, Milan, Italy,
[4]Zama. Inc, Paris, France.

[BdSGJ$^+$24] Lennart Braun, Cyprien Delpech de Saint Guilhem, Robin Jadoul, Emmanuela Orsini, Nigel P. Smart, and Titouan Tanguy. ZK-for-Z2K: MPC-in-the-Head Zero-Knowledge Proofs for $\mathbb{Z}_{2^k}$. In Elizabeth A. Quaglia, editor, *Cryptography and Coding*, pages 137–157, Cham, 2024. Springer Nature Switzerland.

Abstract: In this work, we extend the MPC-in-the-Head framework, used in recent efficient zero-knowledge protocols, to work over the ring $\mathbb{Z}_{2^k}$, which is the primary operating domain for modern CPUs. The proposed schemes are compatible with any threshold linear secret sharing scheme and draw inspiration from MPC protocols adapted for ring operations. Additionally, we explore various batching methodologies, leveraging Shamir's secret sharing schemes and Galois ring extensions, and show the applicability of our approach in RAM program verification. Finally, we analyse different options for instantiating the resulting ZK scheme over rings and compare their communication costs.

My contributions: Main author
I was a main collaborator on the design and proofs for the multiplication checks and ring checks. I also contributed the concept of packing in the Galois domain.

7.1 Introduction

Zero-knowledge (ZK) proofs [GMR85] are a fundamental tool for numerous privacy-preserving applications. A proof system enables a prover to convince a verifier that a statement is true beyond reasonable doubt. The zero-knowledge property additionally ensures that the only information learnt from the interaction by the verifier (or any other listener) is the veracity of the statement, and nothing else.

A common method of expressing statements for proof systems is circuit satisfiability. In this approach, both the prover and verifier possess a circuit C, and the prover aims to demonstrate their knowledge of a witness w which satisfies the condition $C(w) = 0$. Usually, C is a circuit defined over a field, either binary or arithmetic. However, many use cases of ZK proof systems (such as program verification) require the statement to be expressed with arithmetic over a ring, such as $\mathbb{Z}_{2^k}$. In particular, the underlying structure of choice for modern CPUs, 64-bit integers, can be expressed over the ring $\mathbb{Z}_{2^{64}}$. Hence, proof systems natively compatible with this ring arithmetic allow to preserve the semantics of a conventional CPU, without the costly need to emulate it with finite field arithmetic instead.

There are few exceptions to this approach and some ZK protocols have been extended to operate over rings. In particular, when considering highly efficient and scalable zero-knowledge protocols, some works [BBMH$^+$21, BBMHS22, LXY23] have extended protocols based on vector oblivious linear evaluation (VOLE) to work over $\mathbb{Z}_{2^k}$. These kinds of proofs are able to handle very large statements, such as proving properties of complex computer programs, but are only designated-verifier, i.e., the verifier needs to keep some state secret from the prover. This means that these proofs cannot be made non-interactive and require both parties to be online at the same time.

Publicly verifiable proofs can be generated in different ways, for example following the MPC-in-the-Head (MPCitH) paradigm introduced by Ishai, Kushilevitz, Ostrovsky and Sahai in [IKOS07]. Despite its simplicity, this technique has proven efficiency and flexibility, and found a variety of different applications. In the context of zero-knowledge, MPCitH leads to very efficient protocols [AHIV17, BN20, GMO16, FR22, FMRV22, KZ22, KKW18] for proving statements that can be expressed with small to medium-size circuits, and it can be used to develop efficient post-quantum digital signature schemes [BDK$^+$21, CDG$^+$17].

MPC-in-the-Head. The core idea behind MPCitH is for the prover $\mathcal{P}$ to emulate an MPC protocol for the circuit C, amongst N parties, *in their head*, and commit to each of the emulated parties' view. The verifier $\mathcal{V}$ then asks to decommit a small enough subset of these views so as not to break the privacy of the MPC scheme. The soundness of the proof comes from the correctness of the underlying secure MPC protocol and the decommitment of parties' views. In this way, if the prover wants to cheat in the MPC protocol, they need to simulate some parties as acting maliciously, which in turn can be detected if the set of malicious parties overlaps the set of decommited parties. In addition, since the verifier sees fewer views than the privacy threshold of the MPC protocol, the zero-knowledge property holds.

The seminal work of Ishai et al. [IKOS07] describes a generic compiler which makes black-box use of the underlying MPC protocol, but only considers asymptotic complexity; on the other hand, recent concretely efficient protocols [GMO16, FR22, FMRV22, KKW18, AHIV17] provide different concrete instantiations for the MPC protocol used to evaluate the circuit C, based both on full-threshold [BN20, KZ22, KKW18, DOT21] and variable t-threshold secret-sharing schemes [GMO16, FR22, FMRV22, AHIV17]. In the latter case, the resulting ZK scheme can achieve better soundness and different choices of t result in different proof-size/efficiency/soundness trade-offs.

Another significant difference amongst these efficient MPCitH based schemes lies in the way the MPC protocol is used, i.e., whether its task consists of *computing* the circuit C or just *verifying* it. In the former approach, taken for example by [BN20, KKW18, IKOS07], the prover locally emulates the MPC protocol by secret-sharing the witness w amongst the N simulated parties as the input of the MPC evaluation; then it evaluates in MPC the circuit C and sends to the verifier commitments to each parties' input shares, random tapes and received messages (these values constitute a party's *view*) and to all output shares. Then, the verifier randomly chooses t of the views' commitments to be opened, and verifies that the committed messages are all consistent with each other and with the output shares.

In the latter approach, used for example by [AHIV17, BDK$^+$21, DOT21], instead of computing the entire circuit C in MPC, the prover, that knows the witness and all the intermediate values of the circuit evaluation, inputs (or *injects*) all these values (the witness and results of non-linear operations) in a secret-shared form as input of the MPC protocol, whose role at this point is simply checking that these inputs are indeed correct. This approach usually leads to better performance for the prover. The input of this MPC protocol is also called *extended witness*, since the role of the MPC protocol is not only that of verifying that w is a valid witness, i.e., that $C(w) = 0$, but also that the non-linear operations in C have been honestly computed.

7.1.1 Our Contribution

This work describes how to adapt some efficient MPCitH protocols, like [BN20, DOT21, FR22], to work over a ring of the form $\mathbb{Z}_{2^k}$. As said before, compared to VOLE-based schemes, MPCitH proofs have the advantage to be public coin, which enables public verifiability and the ability to obtain non-interactive proofs via the Fiat–Shamir transformation [FS87].[1] We summarize our contributions as follows.

MPCitH over $\mathbb{Z}_{2^k}$. Our approach considers MPCitH schemes such as Limbo [DOT21] and [FR22] where the MPC protocol is used to *verify* the correctness of the committed extended inputs. This type of protocols can be well suited to particular use cases, such as verifying computations or proving the correct execution of RAM programs (where an extension of existing protocols to work over $\mathbb{Z}_{2^k}$ can be practically relevant).

In recent years, MPC protocols have also been extended to work over rings; see for example [CDE+18, EXY22] for the case of dishonest majority (i.e. $t \geq N/2$), and [ACD+19, JSv22] for the case of honest majority (i.e. $t < N/2$). In the case of honest majority protocols, the natural secret-sharing scheme to instantiate a threshold MPC protocol, Shamir's secret sharing [Sha79], requires the underlying algebraic structure to be suitably large. In the case of MPC over finite fields one simply extends the base field so that it contains $N + 1$ elements (where N is the number of parties). In the case of rings it requires a large enough Galois ring extension, so that the largest *exceptional sequence*[2] in the extension ring contains $N + 1$ elements. This was originally introduced in the context of secret sharing by Fehr [Feh98].
A similar approach is also needed in our protocols, where we replace the full-threshold additive sharing scheme used in Limbo with a t-threshold secret sharing scheme to achieve better soundness. We show different options to instantiate our MPC verification procedures, and analyse their respective communication costs. While the t-threshold approach generally comes with a larger proof size than the additive sharing, it trades this for higher efficiency for the verifier, who now only needs to verify that t parties behaved honestly rather than $N - 1$.

Finally, we recall that KKW [KKW18] already works over any rings. This scheme is known for its efficiency when dealing with small to medium-sized circuits, however, as mentioned earlier, it requires an MPC evaluation of the

[1]Many VOLE proofs can be split into an interactive, witness-independent preprocessing phase and a public-coin online phase, of which the latter can be made non-interactive. Note that this still requires the designated verifier to keep secret state.

[2]Informally, an exceptional sequence of elements in a ring R is such that their pairwise difference is invertible. (See Section 7.2.2.)

entire circuit C, which may not be the most suitable approach for applications like program verification.

Packing techniques. Section 7.6, we describe a methodology for *packing* within our MPCitH proofs, that is, proving multiple statements for the same circuit in parallel, in a single proof. It consists of two orthogonal approaches that could potentially be combined to achieve better packing rates. We take advantage of Shamir's threshold secret sharing scheme by embedding multiple secrets in the roots of the sharing polynomial, and we also make use of the additional coefficients provided by Galois ring extensions by placing multiple secrets within a single ring element.

Performing batch proofs in this way additionally alleviates the extra communication cost for a threshold scheme, since the extra space that was introduced to have a large enough exceptional set becomes completely utilized. In combination with the increased verifier efficiency and the better soundness guarantees, this makes the threshold setting preferable to the additive setting for batch proofs.

RAM applications. In Section 7.7, we adapt the compilation procedure of [DOTV22] to the ring structure. The techniques used there allow to *compile* a list of read and write array accesses to a *standard* arithmetic circuit for proof systems in order to enable program verification. This compilation naturally fits the MPCitH framework extended to the ring $\mathbb{Z}_{2^k}$ that we describe in this chapter. This approach removes the need of any bit-decomposition operation; this is different from other recent works [GHAH+23] that use MPCitH schemes based on the KKW protocol [KKW18] for program verification and ring switching techniques based on edaBits [EGK+20].

In our work, to verify the correctness of the memory operations, the initial array is extended to a *checking circuit* C_{check} over $\mathbb{Z}_{2^k}$—with standard linear and multiplication gates and calls to a random oracle—that verifies the consistency of a list of access tuples which contains both the initial array and the accesses performed, encoded as a set of tuples. Given this list, C_{check} produces new multiplication triples that need to be verified via a checking procedure over rings. To perform these consistency checks, [DOTV22] describes three subcircuits $\mathsf{EqCheck}$, $\mathsf{BdCheck}$ and $\mathsf{PermCheck}$ to verify respectively equality, upper and lower bounds and permutation of a list of values in zero-knowledge.

While our compilation follows the blueprint of [DOTV22], the main difference is that, to suit the ring structure, we require a large enough exceptional sequence and the removal of the $\mathsf{EqCheck}$ sub-circuit that crucially relies on every element having an inverse. Our resulting construction inherits all the properties of the scheme described in [DOTV22], leading to a public-coin constant-overhead ZK proof system for computations over $\mathbb{Z}_{2^k}$ in the RAM model.

7.2 Preliminaries

This section establishes notation and recalls standard results.

7.2.1 Notation

We denote by λ the computational security parameter and by σ the statistical security parameter. For a set S, we let $a \leftarrow S$ denote the uniform sampling a from S. If D is a probability distribution over S, we let $a \leftarrow D$ denote sampling a from S according to D. For a probabilistic algorithm A, we let $a \leftarrow A$ denote the probabilistic assigning to a of the output of algorithm A, with the distribution being determined by the random coins of A. We let $[n] \subset \mathbb{N}$ denote the set $\{1, \dots, n\}$. We use $\mathbf{x}$ for vectors of elements, and $\mathbf{x} \circ \mathbf{y}$ for element-wise products.

Zero-knowledge proofs. We use standard definitions of zero-knowledge proofs; we construct our protocols to allow proving arbitrary NP language-membership statements. Let L be in NP and $\mathcal{R}(x, w)$ be a corresponding NP relation with statement x and witness w. That is, the statement x is a member of L if and only if a witness w exists such that $(x, w) \in \mathcal{R}$. We can then consider an arithmetic circuit C (with addition and multiplication gates) that decides (or rather confirms) membership of L when given such a witness. Concretely, the circuit satisfies $C(x, w) = 0$ if and only if $(x, w) \in \mathcal{R}$. The focus of this work are zero-knowledge proofs of knowledge for relations where C is an arithmetic circuit over the ring $\mathbb{Z}_{2^k}$.

7.2.2 Rings

While the circuits we use in our proof systems are defined over the ring $\mathbb{Z}_{2^k}$, we need to work over larger rings to enable threshold secret sharing and to achieve low soundness errors. In this work we consider two ways to obtain such larger rings as described below.

2-adic extensions. Instead of using $\mathbb{Z}_{2^k}$, we increase the modulus and work over $\mathbb{Z}_{2^{k+s}}$, where s depends on the security parameter. This methodology of extending the ring 2-adically in order to check various relations was first introduced in the SPD$\mathbb{Z}_{2^k}$ protocol [CDE+18]. While this is a well-studied technique in the MPC literature, there are some limitations inherent to our application to MPCitH. Many soundness checks that use such an extension only guarantee consistency for the k lower bits; this may therefore require iterating such extensions to $\mathbb{Z}_{2^{k+n\cdot s}}$. Moreover, since $\mathbb{Z}_{2^k}$ is not a subring of $\mathbb{Z}_{2^{k+s}}$, we cannot easily lift $\mathbb{Z}_{2^k}$ elements to $\mathbb{Z}_{2^{k+s}}$ if we also wish to retain some auxiliary algebraic relationship between the lifted values. The converse direction—truncating elements of $\mathbb{Z}_{2^{k+s}}$ to $\mathbb{Z}_{2^k}$—is a well-defined ring homomorphism.

Galois extensions. We extend the base ring $\mathbb{Z}_{2^k}$ by forming the Galois ring $GR(2^k, d) = \mathbb{Z}_{2^k}[X]/(p(X))$, the ring of polynomials with $\mathbb{Z}_{2^k}$ coefficients reduced modulo an irreducible polynomial $p(X)$ of degree d. One advantage of this technique is that reduction modulo 2 results in the field $\mathbb{F}_{2^d}$, i.e., we have $GR(2^k, d)/(2) \simeq \mathbb{F}_{2^d}$. Also, while taking a degree-d extension increases the size of elements by a multiplicative factor d, it can be used for several different checks—unlike the 2-adic extensions. Moreover, a $\mathbb{Z}_{2^k}$ element can be easily "lifted" into a $GR(2^k, d)$ element by using zero for the coefficients of non-constant terms. This lift often retains algebraic relationships between the lifted elements.

Note that both techniques can also be combined to obtain rings of the form $GR(2^{k+s}, d)$.

Definition 7.1: (Maximal) Exceptional Sequence

Let $GR(2^k, d)$ be a degree-d Galois extension of $\mathbb{Z}_{2^k}$. A set $\{\alpha_1, \dots, \alpha_n\}$ is an *exceptional sequence (of length n)* in $GR(2^k, d)$ if for all $i \neq j \in [n]$ we have $\alpha_i - \alpha_j \in GR(2^k, d)^*$.
An exceptional sequence of length n is *maximal* if there does not exist an exceptional sequence of length $n' > n$.

In $GR(2^k, d)$, there exists a maximal exceptional sequence of length 2^d, see [ACD+19, Prop. 2]. We use $\mathsf{Ex}(R)$ to denote a maximal exceptional sequence of a Galois ring R and assume that we can efficiently sample uniformly random elements from it. For $\mathsf{Ex}(R)$ we can take the 2^d polynomials with $\{0, 1\}$ coefficients as an exceptional sequence.

To perform soundness checks in our proof systems, we will often reduce these to equality checks between two polynomials. While the Schwartz–Zippel Lemma is frequently used for this purpose when the polynomials are defined over finite fields, we require a generalized variant that is adapted to our ring-based setting.

Lemma 7.1: Generalized Schwartz–Zippel Lemma [CCKP19]

Let R be a ring, and $f\colon R^n \to R$ an n-variate non-zero polynomial of total degree (the sum of degrees of each variable) D over R. Let $A \subseteq R$ be a finite exceptional sequence with $|A| \geq D$. Then, $\Pr_{\mathbf{x} \in_R A^n}[f(\mathbf{x}) = 0] \leq \frac{D}{|A|}$.

For soundness checks over 2-adic extensions, we also introduce the following lemma to bound the soundness error over $\mathbb{Z}_{2^k}$ when performing computations over $\mathbb{Z}_{2^{k+s}}$.

Lemma 7.2: 2-adic Random Linear Combinations

Let $\delta_1, \ldots, \delta_n$ be elements of $GR(2^{k+s}, d)$, such that at least one $\delta_i \not\equiv 0 \pmod{2^k}$. Also let $\alpha_1 = 1$ and $\alpha_2, \ldots, \alpha_n \leftarrow GR(2^{s+1}, d)$ be chosen uniformly at random. Then we have the probability bound $\Pr\left[\sum \alpha_i \cdot \delta_i \equiv 0 \pmod{2^{k+s}}\right] \leq 2^{-(s+1)\cdot d}$.

Proof. Let δ_j (for $j \neq 1$)[3] be a value that is nonzero modulo 2^k and $w < k$ be the maximal integer such that $2^w \mid \delta_j$. Then $\sum \alpha_i \cdot \delta_i \equiv 0 \pmod{2^{k+s}}$ only when

$$\alpha_j \equiv \frac{-\sum_{i \neq j} \alpha_i \cdot \delta_i}{2^w} \cdot \left(\frac{\delta_j}{2^w}\right)^{-1} \pmod{2^{k+s-w}},$$

where the inverse used is guaranteed to exist due to the maximality of w. Since α_j is uniformly random from $GR(2^{s+1}, d)$ and $k + s - w \geq s + 1$, our claim holds. □

7.2.3 Secret-Sharing Schemes over Rings

We consider additive (A) as well as threshold (T) secret sharing schemes over our commutative finite rings R, e.g. $R = GR(2^k, d)$, which we denote as $[\![\cdot]\!]^A$ and $[\![\cdot]\!]^T$ respectively. Our protocols work with any *linear* secret sharing scheme. Only the overall soundness and the communication cost depend on the instantiation. Hence, we will often drop the A or T from the notation and just write $[\![\cdot]\!]$. Both schemes allow the parties to compute linear functions on shared values such as $[\![\gamma]\!] = a \cdot [\![\alpha]\!] + b \cdot [\![\beta]\!] + c$ by performing only local computations on their

[3] if only $\delta_1 \not\equiv 0$, the equality holds with probability 0.

individual shares.

Additive Secret-Sharing. An additive $(N-1)$-out-of-N secret sharing over R is straightforward. To share a value $v \in R$, first sample values $v_1, \ldots, v_N \leftarrow R$ and then set $\Delta_v = v - \sum_{i\in[N]} v_i$. The share of party P_i is then defined as $[\![v]\!]_i^A := (v_i; \Delta_v)$. We denote this procedure as $[\![v]\!]^A \leftarrow \mathsf{Share}^A(v)$. Reconstruction is performed by computing $v = \Delta_v + \sum_{i\in[N]} v_i$, which we denote as $v \leftarrow \mathsf{Rec}^A([\![v]\!]^A)$.

Threshold Secret-Sharing. The well-known threshold secret sharing scheme due to Shamir [Sha79] relies on polynomial interpolation which usually requires a field structure. We follow the work of Abspoel et al. [ACD+19], who have shown how to use Galois rings to realize Shamir-style threshold secret sharing over rings in the context of MPC.

Let $\alpha_0, \ldots, \alpha_N$ be an exceptional sequence of length $N+1$ within $GR(2^k, d)$. To share a value $v \in \mathbb{Z}_{2^k}$ among parties $P_1, \ldots, P_N$ with threshold t, first sample a random degree-t polynomial f from $GR(2^k, d)[X]^{\leq t}$ with the condition that $f(\alpha_0) = v$. To then create shares, give each party P_i, for $i \in [N]$, the value $[\![v]\!]_i^T := y_i := f(\alpha_i)$. We denote such a sharing with $[\![v]\!]^T \leftarrow \mathsf{Share}^T(v)$.

To reconstruct a value v, we use Lagrange interpolation using any index set $S \subseteq [1, N]$ of at least $t+1$ shares:

$$f(X) = \sum_{i\in S} y_i \cdot \prod_{j\in S\setminus\{i\}} \frac{X - \alpha_j}{\alpha_i - \alpha_j}$$

This interpolation over $GR(2^k, d)$ is well-defined since, by definition of an exceptional sequence, all differences $\alpha_i - \alpha_j$ are invertible. Let the reconstruction procedure be denoted by $v \leftarrow \mathsf{Rec}^T(\{[\![v]\!]_i^T\}_{i\in S})$.

Note that, in general, one needs to check whether a shared value lies in the base ring $\mathbb{Z}_{2^k}$ or (strictly) in the ring extension $GR(2^k, d) \setminus \mathbb{Z}_{2^k}$. To deal with this, we describe a checking procedure $\Pi_{\text{Ring-Check}}$, which ensures a set of shares corresponds to values in $\mathbb{Z}_{2^k}$ without violating t-privacy, in Section 7.4. This procedure can then be applied to the input shares. In our protocols, no other wires or shares, such as the rest of the extended witness, need be validated in this way, as either these shares are obtained through linear operations that preserve this property, or the property is guaranteed by the correctness of our subprotocol to check multiplications.

7.2.4 MPC-in-the-Head via Linear Secret Sharing

This section presents a general framework for MPCitH protocols based on threshold linear secret sharing schemes, built on the framework of Feneuil et al. [FR22] that provides a generic transformation for MPC protocols based on threshold linear secret sharing. We first describe a generic MPC protocol for circuit verification, then show how it can be used to obtain a ZK proof system, and finally analyse the resulting soundness.

MPC Protocol for MPCitH. The MPC protocol presented in Figure 7.1 is generic for threshold LSSS over $\mathbb{Z}_{2^k}$, in the sense that it can be instantiated with any *multiplication checking protocol* and any suitable LSSS. It involves an *input party* who distributes secret shared values to the computing parties. Looking ahead, we refer to the totality of these input values as the *extended witness* of the resulting proof system. In addition, computing parties have access to two oracles: a *hint oracle* $\mathcal{O}_H$ which provides the parties with a sharing of an arbitrary secret value from the input party; and a *randomness oracle* $\mathcal{O}_R$ which outputs random public values.

These oracles are mainly used in the following subprotocols whose goal is to verify some properties on shares of (extended) witness values:

$\Pi_{\text{Zero-Check}}$ takes as input a value $[\![v]\!]$ (resp. a vector of values $[\![\mathbf{v}]\!]$) and returns $\top$ when $v = 0$ (resp. every entry of $\mathbf{v}$ is zero), or $\perp$ otherwise. This can be achieved similarly to share reconstruction, with the difference that the opened value is not sent.

$\Pi_{\text{Mult-Check}}$ takes a triple $([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ and returns $\top$ if and only if $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$. In some cases, this equality can be checked over a different ring than that in which the input values are shared. We provide three instantiations of $\Pi_{\text{Mult-Check}}$ in Section 7.3, and these form the main contribution of this chapter.

$\Pi_{\text{Ring-Check}}$ takes as input a vector of values $[\![\mathbf{v}]\!]$, shared over a 2-adic extension $GR(2^{k+s_{\text{rc}}}, d_0)$ and outputs $\top$ if and only if the truncation of $\mathbf{v}$ to $GR(2^k, d_0)$ lies in the subring $\mathbb{Z}_{2^k}$. It also truncates the elements of $\mathbf{v}$ to the ring $GR(2^{k+s}, d_0)$. (See Section 7.4.)

We write $\Pi^{\tau}_{\text{Mult-Check}}$ to denote the parallel repetition of τ instances. By *verifying* a property through one of these subprotocols, we mean that the subprotocol is run, and reject is returned by the MPC protocol when the output differs from $\top$. Reconstructing a shared value is performed by each party P_j first broadcasting its share $[\![v]\!]_j$ and then running $v \leftarrow \mathsf{Rec}([\![v]\!])$ In the threshold setting, only $t+1$ shares are required since the other shares are determined by these.

Generic MPC Protocol Π_C for Circuit Verification

Parameters: A circuit C over $\mathbb{Z}_{2^k}$ consisting of linear and multiplication gates with $\#\mathsf{inputs}$ inputs and m multiplications Mul; a LSSS sharing scheme $[\![.]\!]$ defined over $GR(2^{k+s}, d_0)$ for parameters s and d_0. The inputs w_i are defined over $GR(2^{k+s_{\mathsf{rc}}}, d_0)$, for parameter $s_{\mathsf{rc}} \geq s$ which matches the parameter for $\Pi_{\text{Ring-Check}}$.

Inputs: The input party calls Share on its input w_i, $i \in [\#\mathsf{inputs}]$ and w_γ for each gate $(\alpha, \beta, \gamma, \mathsf{Mul})_i$ for $i \in [m]$, and send $[\![w_*]\!]_j$ to the computing party P_j.

Protocol: Each P_j initializes an empty checklist $\mathcal{M}$

1. Verify the inputs are in $\mathbb{Z}_{2^k}$: $\Pi_{\text{Ring-Check}}(w_1, \ldots, w_{\#\mathsf{inputs}})$
2. For each gate $(\alpha, \beta, \gamma, T) \in C$, in topological order:
   (a) Case $T = \mathsf{Lin}$: $[\![v_\gamma]\!] := a \cdot [\![v_\alpha]\!] + b \cdot [\![v_\beta]\!] + c$ done locally by each party.
   (b) Case $T = \mathsf{Mul}$:
   - Party P_j retrieves $[\![w_\gamma]\!]_j$ received from the input party and sets $[\![v_\gamma]\!]_j = [\![w_\gamma]\!]_j$.
   - Each P_j adds a tuple to (their share of) the multiplication checklist $\mathcal{M}_j \leftarrow \mathcal{M}_j \cup \{([\![v_\alpha]\!]_j, [\![v_\beta]\!]_j, [\![v_\gamma]\!]_j)\}$
3. Verify circuit output: $\Pi_{\text{Zero-Check}}([\![v_o]\!])$.
4. Verify multiplications: parties parse $\mathcal{M}$ column-wise as $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ and run $\Pi^{\tau_{\mathsf{in}}}_{\text{Mult-Check}}([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$.

Figure 7.1: Generic MPC protocol for circuit verification

In essence, this protocol does not compute the circuit C, but only checks that the values given by the input party are consistent with an honest evaluation of C. To do so, the computation parties parse C in topological order but only (locally) compute the linear gates, whereas output of non-linear gates and Rec are provided as input and hence need to be checked. This is necessary because the input party is not trusted to provide the correct values. The output of the protocol is either accept or reject. To decrease the false-positive rate of the multiplication checking procedure, the parties execute it τ_{in} times in parallel.

From MPC to ZK. The compilation technique of Ishai et al. [IKOS07], applied to this MPC protocol, provides our interactive zero-knowledge scheme between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

The prover executes, in their head, the MPC protocol $\Pi_C(x, w)$ between N parties using an LSSS with t-privacy. To do so, $\mathcal{P}$ first evaluates $C(x, w)$ in the clear, and secret shares w as well as the intermediate values required for a local computation of C. After recording these N input views, it plays the role of the input party and distributes these shares to virtual computing parties. These parties execute $\Pi_C(x, w)$ and its checking sub-protocols. When the protocol queries $\mathcal{O}_H$, the requested shared values are provided by $\mathcal{P}$ to the virtual parties and recorded in the input views. Queries to $\mathcal{O}_R$ are replaced by an interaction with the verifier, where first $\mathcal{P}$ commits to the input views so far, and then $\mathcal{V}$ responds with a random value.

In the final interaction, after Π_C terminates, $\mathcal{V}$ asks to open t of the N views, which it checks for consistency. If the consistency check succeeds, and the output of $\Pi_C(x, w)$ is accept, then $\mathcal{V}$ also outputs accept.

ZK Protocol Soundness. The MPC protocol may output accept for an invalid witness with some bounded false-positive rate p, i.e., the probability that $\Pi_C(x, w)$ outputs accept when in fact $C(x, w) \neq 0$. When p is not sufficiently small, we increase the detection probability by performing τ_{in} parallel *inner repetitions* of the circuit check *inside* the MPC protocol. This leads to an overall false-positive rate of $\mathsf{err}_{\mathsf{MPC}} = p^{\tau_{\mathsf{in}}}$.

The framework of Feneuil et al. [FR22] provides a generic transformation for any such MPC protocol with N parties and tolerating up to t corruptions into an MPCitH proof, with a soundness error of

$$\mathsf{err}_{\mathsf{ZK}} = \frac{1}{\binom{N}{t}} + \mathsf{err}_{\mathsf{MPC}} \cdot \frac{t \cdot (N - t)}{t + 1}. \tag{7.1}$$

For an additive full-threshold secret sharing scheme ($t = N - 1$), this becomes

$$\mathsf{err}_{\mathsf{ZK}} = \frac{1}{N} + \mathsf{err}_{\mathsf{MPC}} \cdot \left(1 - \frac{1}{N}\right).$$

By setting N and t, we obtain a certain $\mathsf{err}_{\mathsf{ZK}}$ for the soundness error of a *single execution* of the protocol. Since this may be too high for a given security setting, we can repeat the transformed protocol τ_{out} times (*outer repetitions*) to obtain any desired soundness error, $\mathsf{err}_{\mathsf{ZK}}^{\tau_{\mathsf{out}}}$.

We denote the overall proof size by $\mathsf{size}_{\mathsf{Proof}}$, which one can think of as the communication cost in bits, required to commit to the parties' views and open t of them in τ_{out} repetitions.

7.3 Checking Multiplications over Rings

We now describe three instantiations for $\mathsf{CheckMult}$. The three protocols have appeared previously in the context of MPCitH over fields, but their extension to MPC over rings is mostly new, although a protocol similar to our sacrificing check can be found in [BBMH+21] for VOLE-based zero-knowledge proofs over $\mathbb{Z}_{2^k}$.

We analyse their soundness in the ring-based setting, and compare their performance. For each of the checking procedures, we analyse the false-positive rate $\mathsf{err}_{\mathsf{MPC}}$ of the resulting MPC protocol. It then suffices to use the generic transformation of Feneuil and Rivain [FR22] to compile our MPC protocol into an MPCitH proof system with soundness error as in eq. (7.1).

Our three different checking procedures are: 1) A simple sacrifice-based check, $\Pi_{\text{Sac-Check}}$ (described in Section 7.3.1), 2) an inner product multiplication check, $\Pi_{\text{IP-Check}}$ (in Section 7.3.2), and 3) a compressed multiplication check, Π_{Compress} (in Section 7.3.3). For the first two of these, one can improve the soundness by utilizing either 2-adic or Galois extensions. The third, compressed multiplication check, is adapted from the methodology in [BBC+19, DOT21], and requires a Galois ring extension.

Looking ahead, in the next section we also present a fourth procedure which checks that a set of shares (typically the input to the circuit) all correspond to values in $\mathbb{Z}_{2^k}$ (as in line 1 of Figure 7.1). This procedure takes its inputs as shares in $GR(2^{k+s_{\mathsf{rc}}}, d_0)$, has a soundness error of $\mathsf{err}_{\mathsf{Ring\text{-}Check}}$. When the chosen multiplication checking procedure would have sufficient soundness with smaller $s < s_{\mathsf{rc}}$, it is possible to locally truncate the input shares correspondingly before performing the procedure.

The false-positive rate of the MPC protocol becomes $\mathsf{err}_{\mathsf{MPC}} := \mathsf{err}_{\mathsf{Check}}^{\tau_{\mathsf{in}}} + \mathsf{err}_{\mathsf{Ring\text{-}Check}}$ where $\mathsf{err}_{\mathsf{Check}}$ denotes the false-positive rate of a single execution of the checking procedure. In Section 7.5, we investigate the differences in communication cost for our different multiplication checks and sharing scheme choices.

7.3.1 Sacrifice Based Check

Our first multiplication checking procedure is a sacrificing based check. This is based on the checking protocol of Baum and Nof [BN20], combined with an optimization of Kales and Zaverucha [KZ22, Sec. 2.5, Optimization 3], transferred to the ring setting. The algorithm is presented in Figure 7.2.

$\Pi_{\text{Sac-Check}}$: Sacrificing Check

Parameters: Additional Galois extension size d_1.

Inputs: $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ shared over $GR(2^{k+s}, d_0)$.

Protocol:

1. Lift $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ to $GR(2^{k+s}, d_0 \cdot d_1)$.
2. $(\llbracket \mathbf{a} \rrbracket, \llbracket \mathbf{c} \rrbracket) \leftarrow \mathcal{O}_H$ uniformly random with $\mathbf{a} \circ \mathbf{y} = \mathbf{c}$ over $GR(2^{k+s}, d_0 \cdot d_1)$
3. $\varepsilon \leftarrow \mathcal{O}_R$ such that $\varepsilon \in GR(2^{1+s}, d_0 \cdot d_1)$
4. $\boldsymbol{\alpha} \leftarrow \mathsf{Rec}(\varepsilon \cdot \llbracket \mathbf{x} \rrbracket - \llbracket \mathbf{a} \rrbracket)$
5. Output $\Pi_{\text{Zero-Check}}(\varepsilon \cdot \llbracket \mathbf{z} \rrbracket - \llbracket \mathbf{c} \rrbracket - \boldsymbol{\alpha} \circ \llbracket \mathbf{y} \rrbracket)$

Figure 7.2: The sacrificing check over rings.

As inputs, it receives the vectors $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ of multiplication input and output values, secret-shared over the "computation ring" $GR(2^{k+s}, d_0)$. In case of $d_1 > 1$, it first lifts these vectors to the "checking ring" $GR(2^{k+s}, d_0 \cdot d_1)$. Then, the hint oracle $\mathcal{O}_H$ distributes to the parties secret shares of $\llbracket \mathbf{a} \rrbracket$ and $\llbracket \mathbf{c} \rrbracket$, correlated in such a way that $\mathbf{a} \circ \mathbf{y} = \mathbf{c}$. After receiving a random coefficient ε from the randomness oracle $\mathcal{O}_R$, the parties "sacrifice" the vector $\llbracket \mathbf{a} \rrbracket$ by using it to mask the randomized vector $\varepsilon \cdot \llbracket \mathbf{x} \rrbracket$ and reconstruct the masked value as $\boldsymbol{\alpha}$. Finally, the protocol checks whether both $\mathbf{z}$ and $\mathbf{c}$ were computed correctly by $\mathcal{O}_H$ by checking that the sacrificing equation $\varepsilon \cdot \llbracket \mathbf{z} \rrbracket - \llbracket \mathbf{c} \rrbracket - \boldsymbol{\alpha} \circ \llbracket \mathbf{y} \rrbracket$ is equal to 0. The argument is that if either $\mathbf{z}$ or $\mathbf{c}$ is incorrect, then the probability that the equality holds, taken over the choice of $\varepsilon \in GR(2^{1+s}, d_0 \cdot d_1)$, is very small.

We first take a brief look at the correctness of the protocol. If the input is valid, then the protocol always outputs accept, since

$$\varepsilon \cdot \mathbf{z} - \mathbf{c} - \boldsymbol{\alpha} \circ \mathbf{y} = \varepsilon \cdot \mathbf{x} \circ \mathbf{y} - \mathbf{a} \circ \mathbf{y} - (\varepsilon \cdot \mathbf{x} - \mathbf{a}) \circ \mathbf{y}$$

$$= \varepsilon \cdot \mathbf{x} \circ \mathbf{y} - \mathbf{a} \circ \mathbf{y} - \varepsilon \cdot \mathbf{x} \circ \mathbf{y} + \mathbf{a} \circ \mathbf{y} = 0.$$

The zero-knowledge property remains preserved by virtue of $\boldsymbol{\alpha}$ being uniformly random as a result of the mask $\mathbf{a}$ being uniformly random.

Soundness follows from the following theorem.

Theorem 7.1: Soundness of $\Pi_{\text{Sac-Check}}$

For invalid input, i.e., $\exists i \in [m] \,.\, x_i \cdot y_i \neq z_i$, the check passes with probability at most $\mathsf{err}_{\mathsf{Sac\text{-}Check}} := 2^{-(s+1)\cdot d_0 \cdot d_1}$.

Proof. Write $\mathbf{x} \circ \mathbf{y} = \mathbf{z} + \boldsymbol{\delta}_z$ and $\mathbf{a} \circ \mathbf{y} = \mathbf{c} + \boldsymbol{\delta}_c$. The protocol outputs accept if and only if for all $i \in [m]$, we have

$$\begin{aligned}
0 &= \varepsilon \cdot z_i - c_i - \alpha_i \cdot y_i \\
&= \varepsilon \cdot (x_i \cdot y_i + \delta_{z,i}) - (a_i \cdot y_i + \delta_{c,i}) - (\varepsilon \cdot x_i - a_i) \cdot y_i \\
&= \varepsilon \cdot x_i \cdot y_i + \varepsilon \cdot \delta_{z,i} - a_i \cdot y_i - \delta_{c,i} - \varepsilon \cdot x_i \cdot y_i + a_i \cdot y_i \\
&= \varepsilon \cdot \delta_{z,i} - \delta_{c,i}.
\end{aligned}$$

Recall that $\varepsilon \in_R GR(2^{s+1}, d_0 \cdot d_1)$, $\delta_{z,j} \in GR(2^{k+s}, d_0)$, and $\delta_{c,j} \in GR(2^{k+s}, d_0 \cdot d_1)$. Assume that $\delta_{z,j} \neq 0 \pmod{2^k}$ for some $j \in [m]$. By Lemma 7.2, we can bound the probability that a malicious prover chooses $\delta_{z,j}, \delta_{c,j}$ such that $0 = \varepsilon \cdot \delta_{z,j} + \delta_{c,j}$ holds over $GR(2^{k+s}, d_0 \cdot d_1)$. □

7.3.2 Inner Product Multiplication Check

Our second checking procedure, which is based on inner product checks, is described as a precursor to the Limbo protocol [DOT21], together with optimizations from Kales and Zaverucha [KZ22], adapted to the ring setting. We present the algorithm in Figure 7.3.

This second checking procedure $\Pi_{\text{IP-Check}}$ works very similarly to the sacrificing check $\Pi_{\text{Sac-Check}}$ of Figure 7.2, the main difference is that the hint oracle $\mathcal{O}_H$ produces a single correlated inner product tuple ($(\mathbf{a}, c)$ such that $\langle \mathbf{a}, \mathbf{y} \rangle = c$) rather than m correlated multiplication tuples ($(\mathbf{a}, \mathbf{c})$ such that $\mathbf{a} \circ \mathbf{y} = \mathbf{c}$). This change then requires the random oracle $\mathcal{O}_R$ to produce m random values (contained in the vector $\boldsymbol{\eta}$), instead of a single one, and it also changes the checking equation so that it checks a single equality, rather than m. This time, the security rationale is that if either $\mathbf{z}$ or c is incorrect, then the single checking equation will not equal 0 except with small probability (over the choice of $\boldsymbol{\eta}$). The rationale for the zero-knowledge property is again due to the random mask $[\![\mathbf{a}]\!]$.

$\Pi_{\text{IP-Check}}$: Inner Product Check

Parameters: Additional Galois extension size d_1.

Inputs: $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ shared over $GR(2^{k+s}, d_0)$.

Protocol:

1. Lift $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ to $GR(2^{k+s}, d_0 \cdot d_1)$.
2. $([\![\mathbf{a}]\!], [\![c]\!]) \leftarrow \mathcal{O}_H$ uniformly random with $\langle \mathbf{a}, \mathbf{y} \rangle = c$ over $GR(2^{k+s}, d_0 \cdot d_1)$.
3. $\boldsymbol{\eta} \leftarrow \mathcal{O}_R$ such that $\boldsymbol{\eta} \in GR(2^{1+s}, d_0 \cdot d_1)^m$.
4. $\boldsymbol{\alpha} \leftarrow \mathsf{Rec}(\boldsymbol{\eta} \circ [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!])$
5. Output $\Pi_{\text{Zero-Check}}(\langle \boldsymbol{\eta}, [\![\mathbf{z}]\!] \rangle - [\![c]\!] - \langle \boldsymbol{\alpha}, [\![\mathbf{y}]\!] \rangle)$

Figure 7.3: The inner product check over rings.

Here as well, the protocol is correct, since if the input is valid, then the protocol always outputs accept as

$$\langle \boldsymbol{\eta}, \mathbf{z} \rangle - c - \langle \boldsymbol{\alpha}, \mathbf{y} \rangle = \langle \boldsymbol{\eta}, \mathbf{x} \circ \mathbf{y} \rangle - \langle \mathbf{a}, \mathbf{y} \rangle - \langle \boldsymbol{\eta} \circ \mathbf{x} - \mathbf{a}, \mathbf{y} \rangle$$

$$= \langle \boldsymbol{\eta}, \mathbf{x} \circ \mathbf{y} \rangle - \langle \mathbf{a}, \mathbf{y} \rangle - \langle \boldsymbol{\eta} \circ \mathbf{x}, \mathbf{y} \rangle + \langle \mathbf{a}, \mathbf{y} \rangle = 0.$$

Soundness follows from the following theorem.

Theorem 7.2: Soundness of $\Pi_{\text{IP-Check}}$

For invalid input, i.e., $\exists i \in [m] \,.\, x_i \cdot y_i \neq z_i \pmod{2^k}$, the check passes with probability at most $\mathsf{err}_{\mathsf{IP\text{-}Check}} := 2^{-(s+1) \cdot d_0 \cdot d_1}$.

Proof. Write $\mathbf{x} \circ \mathbf{y} = \mathbf{z} + \boldsymbol{\delta}_z$ and $\langle \mathbf{a}, \mathbf{y} \rangle = c + \delta_c$. If the input is invalid, then there is an index $j \in [m]$ such that $\delta_{z,j} \neq 0 \pmod{2^k}$. The protocol accepts if and only if

$$0 = \langle \boldsymbol{\eta}, \mathbf{z} \rangle - c - \langle \boldsymbol{\alpha}, \mathbf{y} \rangle = \langle \boldsymbol{\eta}, \mathbf{z} \rangle - c - \langle \boldsymbol{\eta} \circ \mathbf{x}, \mathbf{y} \rangle + \langle \mathbf{a}, \mathbf{y} \rangle$$

$$= \sum_{i \in [m]} \eta_i \cdot (z_i - x_i \cdot y_i) - c + \langle \mathbf{a}, \mathbf{y} \rangle = \sum_{i \in [m]} \eta_i \cdot (-\delta_{z,i}) + \delta_c$$

With this equality, we can conclude by Lemma 7.2. □

Π_{Compress} Subroutine for Inner Product Compression

Parameters: compression factor ν, dimension ℓ, flag $\mathsf{rand} \in \{\top, \bot\}$

Inputs: ν shared dimension-ℓ inner product tuples $(\llbracket \mathbf{x}_i \rrbracket, \llbracket \mathbf{y}_i \rrbracket, \llbracket z_i \rrbracket)_{i \in [\nu]}$ shared over $GR(2^k, d)$

Outputs: one shared dimension-ℓ inner product tuple $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket z \rrbracket)$ shared over $GR(2^k, d)$

Protocol:

Let $\{\alpha_1, \dots, \alpha_{2\nu+1}\} \subset \mathsf{Ex}(GR(2^k, d))$.

1. If $\mathsf{rand} = \bot$ define two shared dimension-ℓ vectors of degree-$(\nu - 1)$ polynomials $\llbracket \mathbf{f} \rrbracket, \llbracket \mathbf{g} \rrbracket$:

$$\mathbf{f}(\alpha_i) = \begin{pmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_\nu \end{pmatrix}^T$$

$$\mathbf{g}(\alpha_i) = \begin{pmatrix} \mathbf{y}_1 & \cdots & \mathbf{y}_\nu \end{pmatrix}^T$$

   where $i \in [\nu]$. Note, the parties can compute the shared coefficients $\llbracket f_j \rrbracket, \llbracket g_j \rrbracket$ locally from the $\llbracket \mathbf{x}_i \rrbracket, \llbracket \mathbf{y}_i \rrbracket$ by Lagrange interpolation.

   If $\mathsf{rand} = \top$, obtain random shares $\llbracket \mathbf{v} \rrbracket, \llbracket \mathbf{w} \rrbracket \leftarrow \mathcal{O}_H$ and define $\mathbf{f}, \mathbf{g}$ instead of degree ν with the additional points $\mathbf{f}(\alpha_{\nu+1}) = \mathbf{v}$ and $\mathbf{g}(\alpha_{\nu+1}) = \mathbf{w}$.
2. Inject $\llbracket z_i \rrbracket \leftarrow \mathcal{O}_H$ for $i \in [\nu+1, 2\nu-1]$ such that $z_i := \langle \mathbf{f}(\alpha_i), \mathbf{g}(\alpha_i) \rangle$.

   If $\mathsf{rand} = \top$, similarly inject $\llbracket z_i \rrbracket$ for $i \in \{2\nu, 2\nu + 1\}$.
3. If $\mathsf{rand} = \bot$ define shared polynomial $\llbracket h \rrbracket$ of degree $2(\nu - 1)$ by $h(\alpha_i) = z_i$ for $i \in [\nu, 2\nu - 1]$. Again, the parties can compute the shared coefficients $\llbracket h_j \rrbracket$ locally from the $\llbracket z_i \rrbracket$ by Lagrange interpolation.

   If $\mathsf{rand} = \top$, instead define h of degree 2ν with the additional points $h(\alpha_i) = z_i$ for $i \in \{2\nu, 2\nu + 1\}$.
4. Obtain challenge $\varepsilon \leftarrow \mathcal{O}_R$ such that $\varepsilon \in \mathsf{Ex}(GR(2^k, d)) \setminus \{\alpha_i\}_{i \in [\nu]}$.
5. Output $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket z \rrbracket) := (\llbracket \mathbf{f}(\varepsilon) \rrbracket, \llbracket \mathbf{g}(\varepsilon) \rrbracket, \llbracket h(\varepsilon) \rrbracket)$.

Figure 7.4: The subroutine for inner product compression

7.3.3 Compressed Multiplication Check

Our third and final check is adapted from Limbo [DOT21]. In contrast to the previous checks, we do not use 2-adic extensions here, since we would have to extend the modulus repeatedly at least $\log_\nu(m)$ times. To apply the compressed protocol with compression factor ν, the check must happen over an algebraic structure where an exceptional sequence of length at least $2\nu+1$ exists.

We first give the subprotocol of [DOT21] to compress a sequence of ν inner product tuples into a single inner product tuple in Figure 7.4; then we present the main protocol in Figure 7.5. Correctness and zero-knowledge for this checking protocol follow the same arguments as the original version over fields. Soundness follows from the following theorem.

Theorem 7.3: Soundness of $\Pi_{\text{Comp-Check}}$

Let $d := d_0 \cdot d_1$. For invalid input, i.e., $\exists i \in [m] \,.\, x_i \cdot y_i \neq z_i \pmod{2^k}$, the check passes with probability at most

$$\begin{aligned}\mathsf{err}_{\text{Comp-Check}} :=& 2^{-d} + (1-2^{-d}) \cdot \left(\left(\frac{2(\nu-1)}{2^d-\nu}\right) \cdot \sum_{j=0}^{\log_\nu(m)-2} \left(1 - \frac{2(\nu-1)}{2^d-\nu}\right)^j \right.\\ &\left. + \left(\frac{2\nu}{2^d-\nu}\right) \cdot \left(1-\frac{2(\nu-1)}{2^d-\nu}\right)^{\log_\nu(m)-1}\right)\\ \leq& 2^{-d} + \frac{2\nu}{2^d-\nu} \cdot \log_\nu(m).\end{aligned}$$

Proof. We follow the corresponding proof by [DOT21] and define a sequence of events given that the input is invalid:

- Let A be the event that the protocol outputs accept.
- Let A_1 be the event that the tuple $(\llbracket \mathbf{x}^0 \rrbracket, \llbracket \mathbf{y}^0 \rrbracket, \llbracket z^0 \rrbracket)$ obtained through the $\mathsf{ConstructIP}$ subprotocol is correct.
- Let A_2^j for $j \in [\log_\nu(m)]$ be the event that the tuple $(\llbracket \mathbf{x}^j \rrbracket, \llbracket \mathbf{y}^j \rrbracket, \llbracket z^j \rrbracket)$ obtained through the $\mathsf{Compress}$ subprotocol is correct, and write $A_2^0 = A_1$.

We relate the probabilities as follows:

$$\Pr[A] = \Pr[A_1] + \Pr[\neg A_1] \cdot \Pr[A \mid \neg A_1]$$

$$\Pr[A \mid \neg A_2^j] = \Pr[A_2^{j+1}] + \Pr[\neg A_2^{j+1}] \cdot \Pr[A \mid \neg A_2^{j+1}] \quad \text{for } j \in [1, \log_\nu(m) - 1]$$

$$\Pr[A \mid \neg A_2^{\log_\nu(m)}] = 0$$

$\Pi_{\text{Comp-Check}}$: Compressed Multiplication Check

Parameters: number of multiplications m, compression factor ν (assume $\log_\nu(m) \in \mathbb{N}$), Galois extension degree d_1

Inputs: $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ of length m shared over $GR(2^k, d_0)$.

Protocol:

1. Lift $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ to $GR(2^k, d_0 \cdot d_1)$.
2. Create inner product tuple $\left([\![\mathbf{x}^0]\!], [\![\mathbf{y}^0]\!], [\![z^0]\!]\right)$:
   (a) $\boldsymbol{\eta} \leftarrow \mathcal{O}_R$ such that $\boldsymbol{\eta} \in GR(2, d_0 \cdot d_1)^m$.
   (b) Set $[\![\mathbf{x}^0]\!] := \boldsymbol{\eta} \circ [\![\mathbf{x}]\!]$, $[\![\mathbf{y}^0]\!] := [\![\mathbf{y}]\!]$, and $[\![z^0]\!] := \langle \boldsymbol{\eta}, [\![\mathbf{z}]\!] \rangle$.
3. For each round $j \in [\log_\nu(m)]$:
   (a) Parse $\left([\![\mathbf{x}^{j-1}]\!], [\![\mathbf{y}^{j-1}]\!], [\![z^{j-1}]\!]\right)$ (of length m/ν^{j-1}) as
   $$[\![\mathbf{x}^{j-1}]\!] = \left([\![\mathbf{a}_1^j]\!], \dots, [\![\mathbf{a}_\nu^j]\!]\right)$$
   $$[\![\mathbf{y}^{j-1}]\!] = \left([\![\mathbf{b}_1^j]\!], \dots, [\![\mathbf{b}_\nu^j]\!]\right)$$
   where the $\mathbf{a}_i^j, \mathbf{b}_i^j$ are of length m/ν^j.
   (b) For $i \in [\nu - 1]$, obtain $[\![c_i^j]\!] \leftarrow \mathcal{O}_H$ such that $c_i^j = \left\langle \mathbf{a}_i^j, \mathbf{b}_i^j \right\rangle$.
   (c) Set $[\![c_\nu^j]\!] = [\![z^{j-1}]\!] - \sum_{i \in [\nu-1]} [\![c_i^j]\!]$
   (d) If $j < \log_\nu(m)$, run
   $$\left([\![\mathbf{x}^j]\!], [\![\mathbf{y}^j]\!], [\![z^j]\!]\right) \leftarrow \Pi_{\text{Compress}}(([\![\mathbf{a}_i^j]\!], [\![\mathbf{b}_i^j]\!], [\![c_i^j]\!])_{i \in [\nu]}),$$
   else if $j = \log_\nu(m)$, run
   $$\left([\![\mathbf{x}^j]\!], [\![\mathbf{y}^j]\!], [\![z^j]\!]\right) \leftarrow \Pi^{\mathsf{Rand}}_{\text{Compress}}(([\![\mathbf{a}_i^j]\!], [\![\mathbf{b}_i^j]\!], [\![c_i^j]\!])_{i \in [\nu]}).$$
   Both yield inner product tuples of length m/ν^j.
4. Open $\mathbf{x}^{\log_\nu(m)} \leftarrow \mathsf{Rec}([\![\mathbf{x}^{\log_\nu(m)}]\!])$.
5. Output $\Pi_{\text{Zero-Check}}(\mathbf{x}^{\log_\nu(m)} \cdot [\![\mathbf{y}^{\log_\nu(m)}]\!] - [\![z^{\log_\nu(m)}]\!])$.

Figure 7.5: The compressed multiplication check

We get from Lemmas 7.4 and 7.3.3 (see below), that

$$\Pr[A_1] \overset{L.~7.4}{=} 2^{-d}$$

$$\Pr[A_2^j] \overset{L.~7.3.3}{=} \frac{2(\nu-1)}{2^d - \nu} \qquad \text{for } j \in [1, \log_\nu(m) - 1]$$

$$\Pr[A_2^{\log_\nu(m)}] \overset{L.~7.3.3}{=} \frac{2\nu}{2^d - \nu}$$

Combining them (and using $A_1 = A_2^0$) yields

$$\begin{aligned}
\Pr[A] &= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Pr[A \mid \neg A_2^0] \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot (\Pr[A_2^1] + \Pr[\neg A_2^1] \cdot \Pr[A \mid \neg A_2^1]) \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Pr[A_2^1] + \Pr[\neg A_2^0] \cdot \Pr[\neg A_2^1] \cdot \Pr[A \mid \neg A_2^1] \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \Pr[A_2^1] \\
&\quad + \Pr[\neg A_2^0] \cdot \Pr[\neg A_2^1] \cdot (\Pr[A_2^2] + \Pr[\neg A_2^2] \cdot \Pr[A \mid \neg A_2^2]) \\
&= \cdots \\
&= \sum_{j=0}^{\log_\nu(m)} \Pr[A_2^j] \cdot \prod_{i=0}^{j-1} \Pr[\neg A_2^i] + \Pr[A \mid \neg A_2^{\log_\nu(m)}] \cdot \prod_{j=0}^{\log_\nu(m)} \Pr[\neg A_2^j] \\
&= \sum_{j=0}^{\log_\nu(m)} \Pr[A_2^j] \cdot \prod_{i=0}^{j-1} \Pr[\neg A_2^i] \\
&= \Pr[A_2^0] + \Pr[\neg A_2^0] \cdot \left(\sum_{j=1}^{\log_\nu(m)-1} \Pr[A_2^j] \cdot \prod_{i=1}^{j-1} \Pr[\neg A_2^i] \right. \\
&\qquad\qquad \left. + \Pr[A_2^{\log_\nu(m)}] \cdot \prod_{i=1}^{\log_\nu(m)-1} \Pr[\neg A_2^i] \right) \\
&= 2^{-d} + (1 - 2^{-d}) \cdot \left(\left(\frac{2(\nu-1)}{2^d - \nu} \right) \cdot \sum_{j=0}^{\log_\nu(m)-2} \left(1 - \frac{2(\nu-1)}{2^d-\nu} \right)^j \right. \\
&\qquad\qquad \left. + \left(\frac{2\nu}{2^d-\nu} \right) \cdot \left(1 - \frac{2(\nu-1)}{2^d-\nu} \right)^{\log_\nu(m)-1} \right) \\
&\leq 2^{-d} + \frac{2\nu}{2^d-\nu} \cdot \log_\nu(m),
\end{aligned}$$

which concludes this proof. □

Lemma 7.3: Soundness of Π_{Compress}

If one of the inner product tuples

$$(\llbracket \mathbf{x}_i \rrbracket, \llbracket \mathbf{y}_i \rrbracket, \llbracket z_i \rrbracket)_{i \in [\nu]}$$

is incorrect, or any of the values z_i, $i \in [\nu+1, 2\nu-1]$, is defined incorrectly, then the output inner tuple $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket z \rrbracket)$ is also incorrect, except with probability at most $\frac{2(\nu-1)}{2^d-\nu}$ if $\mathsf{Rand} = \perp$ and $\frac{2\nu}{2^d-\nu}$ if $\mathsf{Rand} = \top$.

Proof. For now assume that $\mathsf{Rand} = \perp$. Suppose there is an error at index $j \in [2\nu-1]$. Then we have $z_j \neq \langle \mathbf{f}(\alpha_j), \mathbf{g}(\alpha_j) \rangle$, where z_j is either part of the input ($j \in [\nu]$) or an injected value ($j \in [\nu+1, 2\nu-1]$). In both cases, we have $h(\alpha_j) \neq \langle \mathbf{f}(\alpha_j), \mathbf{g}(\alpha_j) \rangle$, and therefore $h \neq \langle \mathbf{f}, \mathbf{g} \rangle$.

We now apply the generalized Schwartz-Zippel Lemma (Lemma 7.1). Note that the challenge ε is sampled from the exceptional sequence $\mathsf{Ex}(GR(2^k, d)) \setminus \{\alpha_i\}_{i \in [\nu]}$ of size $2^d - \nu$. Hence, we obtain that $\langle \mathbf{x}, \mathbf{y} \rangle \neq z$ iff $\langle \mathbf{f}, \mathbf{g} \rangle (\varepsilon) \neq h(\varepsilon)$ with probability at most $\frac{2(\nu-1)}{2^d-\nu}$.

In the case $\mathsf{Rand} = \top$, we analogously obtain an error probability of at most $\frac{2\nu}{2^d-\nu}$. □

Lemma 7.4

For invalid input $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ into $\Pi_{\text{Comp-Check}}$, i.e., such that $\mathbf{x} \circ \mathbf{y} \neq \mathbf{z}$, we have $\left\langle \mathbf{x}^0, \mathbf{y}^0 \right\rangle \neq z^0$ except with probability $2^{-d_0 \cdot d_1}$.

Proof. Write $\mathbf{x} \circ \mathbf{y} = \mathbf{z} + \boldsymbol{\delta}_z$ Let $j \in [m]$ such that $x_j \cdot y_j \neq z_j$ and, hence, $\delta_{z,j} \neq 0$. Then

$$\left\langle \mathbf{x}^0, \mathbf{y}^0 \right\rangle = z^0$$

$$\iff \sum_{i \in [m]} \eta_i \cdot x_i \cdot y_i = \sum_{i \in [m]} \eta_i \cdot z_i$$

Hence we can apply Lemma 7.2 □

7.4 Checking Base Ring Sharings

To ensure the prover knows and inputs a witness over the base ring $\mathbb{Z}_{2^k}$, we devise a check for the parties to ensure this in Figure 7.6. We can perform a batched check that all the values we wish to inspect are simultaneously correct by taking a random linear combination with coefficients from $\mathbb{Z}_{2^{1+s_{\mathsf{rc}}}}$, and opening that. Since this would leak a linear combination of secret values, we also allow the prover to input an additional sharing of a value in $\mathbb{Z}_{2^{k+s_{\mathsf{rc}}}}$ to mask this relation (before receiving the random coefficients from the verifier). This is conceptually similar to the recent approach by Shoup and Smart in [SS23].

$\Pi_{\text{Ring-Check}}$

Inputs: $[\![\mathbf{x}]\!] = ([\![x_1]\!], \ldots, [\![x_\ell]\!])$ shared over $GR(2^{k+s_{\mathsf{rc}}}, d_0)$

Protocol:

1. Obtain $[\![x_0]\!]$, corresponding to a value in the ring $\mathbb{Z}_{2^{k+s_{\mathsf{rc}}}}$ from $\mathcal{O}_H$.
2. Receive ℓ random coefficients $r_1, \ldots, r_\ell \in \mathbb{Z}_{2^{1+s_{\mathsf{rc}}}}$ from $\mathcal{O}_R$.
3. Compute and open $[\![v]\!] = [\![x_0]\!] + r_1[\![x_1]\!] + \ldots + r_\ell[\![x_\ell]\!]$.
4. If $v \in \mathbb{Z}_{2^{k+s_{\mathsf{rc}}}}$, return $\top$, otherwise return $\bot$.

Figure 7.6: The check to ensure sharings correspond to values in the base ring.

In [ACD+19], Abspoel et al. consider a similar problem for the case of non-MPCitH MPC protocols. They solve this problem by generating random secret shared masks hiding values in the correct ring by means of hyperinvertible matrices, after which these masks can be adjusted with a public value to hide the wanted secret. In an MPCitH context however, this becomes both less convenient, since all computing parties need to contribute their own randomness, as well as requiring a higher communication cost in the final proof size. Soundness follows from the following theorem.

Theorem 7.4: Soundness of $\Pi_{\text{Ring-Check}}$

For invalid input, that is if any of $x_0, x_1, \ldots, x_\ell$ are a value in $GR(2^k, d_0) \setminus \mathbb{Z}_{2^k}$ when reduced modulo 2^k, the check passes with probability at most $\mathsf{err}_{\text{Ring-Check}} := 2^{-(s_{\mathsf{rc}}+1)}$.

Table 7.1: Rings and numbers of primitive operations used by the three multiplication checking protocols.

	Multiplication Check		
	$\Pi_{\text{Sac-Check}}$	$\Pi_{\text{IP-Check}}$	$\Pi_{\text{Comp-Check}}$
small ring $\mathcal{R}_{\mathsf{small}}$	$GR(2^{k+s}, d_0)$	$GR(2^{k+s}, d_0)$	$GR(2^{k}, d_0)$
big ring $\mathcal{R}_{\mathsf{large}}$	$GR(2^{k+s}, d_0 \cdot d_1)$	$GR(2^{k+s}, d_0 \cdot d_1)$	$GR(2^{k}, d_0 \cdot d_1)$
challenge space $\mathcal{C}$	$GR(2^{1+s}, d_0 \cdot d_1)$	$GR(2^{1+s}, d_0 \cdot d_1)$	$GR(2, d_0 \cdot d_1)$
rounds μ	1	1	$\log_\nu(m) + 1$
input over $\mathcal{R}_{\mathsf{small}}$	$\#\mathsf{inputs} + m$	$\#\mathsf{inputs} + m$	$\#\mathsf{inputs} + m$
hint over $\mathcal{R}_{\mathsf{large}}$	m	1	$(2\nu - 1) \cdot \log_\nu(m) + 2$
uniform hint over $\mathcal{R}_{\mathsf{large}}$	m	m	2
reconstruction over $\mathcal{R}_{\mathsf{large}}$	m	m	1
challenge from $\mathcal{C}$	1	m	$m + \log_\nu(m)$

Proof. This is simply Lemma 7.2, applied to only a single coefficient of the Galois extension. Hence, we get a bound of $2^{-(s_{\mathsf{rc}}+1)\cdot 1}$. □

When dealing with additive sharings, the parties can instead simply check their own local shares to lie in the correct ring and return $\perp$ when this is not the case. For semi-honest parties, this is guaranteed to have no false positives.

7.5 Protocol Communication Costs

The communication costs of the zero-knowledge proofs depends greatly on the used secret sharing scheme and the multiplication check protocol, as well as a large set of parameters. To simplify notation, we use $\mathcal{R}_{\mathsf{small}}$ for the ring used to share the witness, $\mathcal{R}_{\mathsf{large}}$ for the ring extension in which the checks are performed. Moreover, the random challenges from $\mathcal{O}_R$ live in the challenge space $\mathcal{C}$, and μ denotes the number of rounds of the MPC protocol, i.e., the number of calls to $\mathcal{O}_R$. For brevity of notation, we use $\mathcal{B}(S) = \lceil \log_2 |S| \rceil$ to denote the number of bits needed to represent an element from S.

Table 7.1 shows how many primitive operations we need for each checking protocol, and Table 7.2 gives the communication cost of each operation in both sharing types. The costs of the challenges are $\mathcal{B}(\mathcal{C}) \cdot \mu \cdot \tau_{\mathsf{in}}$, since they can be shared across the 'outer repetitions'.

Table 7.2: Communication costs in bits of the primitive operations. Here $\mathcal{B}(\cdot)$ denotes the number of bits required to encode an element of the set passed as argument.

	Sharing Scheme	
	Additive	Threshold
input over $\mathcal{R}_{\mathsf{small}}$	$\mathcal{B}(\mathcal{R}_{\mathsf{small}})$	$\mathcal{B}(\mathcal{R}_{\mathsf{small}}) \cdot t$
hint over $\mathcal{R}_{\mathsf{large}}$	$\mathcal{B}(\mathcal{R}_{\mathsf{large}})$	$\mathcal{B}(\mathcal{R}_{\mathsf{large}}) \cdot t$
uniform hint over $\mathcal{R}_{\mathsf{large}}$	0	$\mathcal{B}(\mathcal{R}_{\mathsf{large}}) \cdot t$
reconstruction over $\mathcal{R}_{\mathsf{large}}$	$\mathcal{B}(\mathcal{R}_{\mathsf{large}})$	$\mathcal{B}(\mathcal{R}_{\mathsf{large}})$
challenge from $\mathcal{C}$	$\mathcal{B}(\mathcal{C})$	$\mathcal{B}(\mathcal{C})$

7.5.1 Primitive Costs

The communication costs for our basic operations can be summarized as follows.

Commitments: Before each call to $\mathcal{O}_R$ the prover commits to the current state of the computation. The $\tau_{\mathsf{out}} \cdot \mu \cdot N$ total commitments can be combined into $\tau_{\mathsf{out}} \cdot \mu$ Merkle trees, and for each round it is sufficient to send a hash of the τ_{out} Merkle roots. Thus, committing costs $2\lambda \cdot \mu$ bits. Before the verifier selects a subset of parties whose views to open, the prover sends another hash with shares of the last reconstructed values.

To open t of the commitments in each repetition, we have to send, in addition to the committed data, λ bits of randomness per commitment as well the corresponding Merkle paths. Each path is of length $\log_2(N)$, but since we open t views and the path overlap, we pay $2\lambda \cdot \log_2(N/t)$ bits per path.

Overall, this results in

$$\mathsf{size}_{\mathsf{Commit}} := 2\lambda \cdot (\mu + 1) + \tau_{\mathsf{out}} \cdot \lambda \cdot \mu \cdot t \cdot (2\log_2(N/t) + 1)$$

bits of communication for committing and opening.

Opening sharings: Since to open a sharing only the reconstructed value needs to be revealed on top of the t already decommited shares, the cost for opening a $\mathbb{Z}_{2^k}$ value is k bits (for a $GR(2^k, d)$ value this is $k \cdot d$ bits), regardless of the secret sharing scheme being used.

Providing hints: The $\mathcal{O}_H$ oracle can be instantiated in two different ways, depending on the kind of secret sharing being used. For a threshold secret sharing scheme, both specific and uniformly random values $v \in \mathbb{Z}_{2^k}$ (or $v \in GR(2^k, d)$) can be obtained by running $[\![v]\!] \leftarrow \mathsf{Share}(v)$ and distributing the shares to the corresponding parties. This costs $t \cdot k$ (or $t \cdot k \cdot d$) bits of proof size.

For additive secret sharing, uniformly random values in $\mathbb{Z}_{2^k}$ or $GR(2^k, d)$ can be obtained at zero extra cost by having all parties individually derive their shares from a PRG seed. A uniformly random sharing $[\![r]\!]^A$ can be transformed into a sharing of a specific value $[\![v]\!]^A$ by updating the public adjustment Δ_v, at the cost of only k or $k \cdot d$ bits of proof size.

7.5.2 Protocol Costs

We can now summarize the communication costs per checking protocol:

$\Pi_{\text{Sac-Check}}$: The sacrificing check requires

$$\mathsf{size}^A_{\mathsf{Sac\text{-}Check}} := 2 \cdot m \cdot (k + s) \cdot d_0 \cdot d_1$$

$$\mathsf{size}^T_{\mathsf{Sac\text{-}Check}} := (2 \cdot m \cdot t + m) \cdot (k + s) \cdot d_0 \cdot d_1$$

bits of additional communication for additive, resp. threshold, sharing.

$\Pi_{\text{IP-Check}}$: The inner product check results requires

$$\mathsf{size}^A_{\mathsf{IP\text{-}Check}} := (m + 1) \cdot (k + s) \cdot d_0 \cdot d_1$$

$$\mathsf{size}^T_{\mathsf{IP\text{-}Check}} := ((m + 1) \cdot t + m) \cdot (k + s) \cdot d_0 \cdot d_1$$

bits of additional communication for additive, resp. threshold, sharing.

$\Pi_{\text{Comp-Check}}$: The compressed multiplication check results requires

$$\mathsf{size}^A_{\mathsf{Comp\text{-}Check}} := ((2\nu - 1) \cdot \log_\nu(m) + 3) \cdot k \cdot d_0 \cdot d_1$$

$$\mathsf{size}^T_{\mathsf{Comp\text{-}Check}} := (((2\nu - 1) \cdot \log_\nu(m) + 4) \cdot t + 1) \cdot k \cdot d_0 \cdot d_1$$

bits of additional communication for additive, resp. threshold, sharing.

$\Pi_{\text{Ring-Check}}$: For additive sharing, this check has no overhead. In the threshold case, this procedure requires one additional share input and one share reconstruction in $GR(2^{k+s_{\mathsf{rc}}}, d_0)$ to the overall proof size, hence the total costs are

$$\mathsf{size}^{A}_{\mathsf{Ring\text{-}Check}} := 0$$

$$\mathsf{size}^{T}_{\mathsf{Ring\text{-}Check}} := (t+1) \cdot (k + s_{\mathsf{rc}}) \cdot d_0$$

bits of communication for additive, resp. threshold, sharing.

Here we do not take into account the cost of the verifier sending a challenge or a seed for outputs of the $\mathcal{O}_R$ oracle. In the non-interactive case, these are obtained from the Fiat–Shamir transform and therefore free in terms of communication; in the interactive case however, the verifier sends λ bits per "round" of dependent calls to $\mathcal{O}_R$.

7.5.3 Overall Costs

Finally, we can present the overall communication cost, i.e., the proof size. Note here that the cost for $\mathsf{size}_{\mathsf{Input}}$ depends on $k + s_{\mathsf{rc}}$, rather than the potentially smaller $k + s$.

$$\mathsf{size}_{\mathsf{Proof}} = \mathsf{size}_{\mathsf{Commit}} + \tau_{\mathsf{out}} \cdot (\mathsf{size}_{\mathsf{Input}} + \tau_{\mathsf{in}} \cdot \mathsf{size}_{\mathsf{Check}}) + \tau_{\mathsf{in}} \cdot \mathsf{size}_{\mathsf{Challenge}}$$

7.5.4 Concrete Comparison of the Three $\Pi_{\text{Mult-Check}}$ Subprotocols

To compare our different protocols concretely with one another, we fix certain choices for σ, k and m and examined the per-multiplication-gate communication cost of a full proof σ bits of security. The size presented in the tables corresponds to the communication cost of an entire proof, except for the challenges sent from the verifier. That is, we only examine the communication from the prover towards the verifier, which also gives a good idea of the proof size that would be incurred when the protocol is transformed to a non-interactive proof by the Fiat-Shamir transform.

All our experimental validations were computed with $\#\mathsf{inputs} = 128$ elements in $\mathbb{Z}_{2^k}$. Since the additive sharing has some optimizations for random sharings and $\Pi_{\text{Ring-Check}}$ and does not require $d_0 > 1$ to enable sharing values across N parties, it generally comes out as the optimal choice for the configurations examined here.

When combining our protocols with the packing techniques of Section 7.6, the balance shifts since a threshold $t < N - 1$ gives better soundness per parallel repetition, allows for more packing, and compensates for the larger d_0 by performing more parallel proofs. Out of interest for this trade-off, we present the parameter sets and associated costs for additive and threshold secret sharing separately.

We observe that for $\Pi_{\text{Sac-Check}}$ and $\Pi_{\text{IP-Check}}$, which require at least m openings each, the optimal choice for d_1 is one since the overhead for $d_0 \cdot s$ extra bits is generally smaller than $d_0 \cdot (d_1 - 1) \cdot k$ extra bits, even though the size of inputs and injected multiplications grows as well. When the communication due to the check is asymptotically smaller than the communication due to the input of the extended witness, it becomes preferable to avoid the extra $d_0 \cdot s$ bits per multiplication cost in the input already.

Table 7.3: Cost comparison for $\sigma = 40$, $m = 1024$ with threshold secret sharing.

k	Protocol	N	t	d_0	d_1	s	s_{rc}	ν	τ_{in}	τ_{out}	Proof size in kB
32	$\Pi_{\text{Sac-Check}}$	63	1	6	1	2	17	/	1	7	748
	$\Pi_{\text{IP-Check}}$	255	3	8	1	3	31	/	1	2	539
	Π_{Compress}	63	1	6	4	/	18	4	1	7	236
64	$\Pi_{\text{Sac-Check}}$	255	3	8	1	3	31	/	1	2	1 413
	$\Pi_{\text{IP-Check}}$	255	3	8	1	3	31	/	1	2	1 012
	Π_{Compress}	63	1	6	4	/	18	4	1	7	452
256	$\Pi_{\text{Sac-Check}}$	255	3	8	1	3	31	/	1	2	5 399
	$\Pi_{\text{IP-Check}}$	255	3	8	1	3	31	/	1	2	3 846
	Π_{Compress}	63	1	6	4	/	18	2	1	7	1 726

Since we can observe that Π_{Compress} consistently results in the smallest proof sizes, we further also look at the overhead of this protocol. That is, we investigate the ratio of proof size to the theoretical optimum of $k \cdot (\#\mathsf{inputs} + m)$ bits for any protocol that needs to inject the results of multiplications. This rate is a constant that mostly depends on the target value of σ and decreases slightly as the number of multiplications increases. Since the choice of k doesn't influence the choice of multiplication check, it also has no further impact on the overhead.

Table 7.4: Cost comparison for $\sigma = 40$, $m = 1024$ with additive secret sharing.

k	Protocol	N	d_0	d_1	s	ν	τ_{in}	τ_{out}	Proof size in kB
	$\Pi_{\text{Sac-Check}}$	255	1	1	7	/	1	6	116
32	$\Pi_{\text{IP-Check}}$	63	1	1	8	/	1	7	82
	Π_{Compress}	15	1	12	/	4	1	11	87
	$\Pi_{\text{Sac-Check}}$	255	1	1	7	/	1	6	191
64	$\Pi_{\text{IP-Check}}$	255	1	1	7	/	1	6	137
	Π_{Compress}	63	1	14	/	4	1	7	135
	$\Pi_{\text{Sac-Check}}$	255	1	1	7	/	1	6	641
256	$\Pi_{\text{IP-Check}}$	255	1	1	7	/	1	6	443
	Π_{Compress}	63	1	14	/	4	1	7	411

Table 7.5: Cost comparison for $\sigma = 40$, $m = 32768$ with threshold secret sharing.

k	Protocol	N	t	d_0	d_1	s	s_{rc}	ν	τ_{in}	τ_{out}	Proof size in kB
	$\Pi_{\text{Sac-Check}}$	255	3	8	1	3	31	/	1	2	22 449
32	$\Pi_{\text{IP-Check}}$	255	3	8	1	3	31	/	1	2	15 729
	Π_{Compress}	63	1	6	4	/	17	4	1	7	5 459
	$\Pi_{\text{Sac-Check}}$	255	3	8	1	3	31	/	1	2	42 953
64	$\Pi_{\text{IP-Check}}$	255	3	8	1	3	31	/	1	2	30 090
	Π_{Compress}	63	1	6	4	/	17	4	1	7	10 895
	$\Pi_{\text{Sac-Check}}$	255	3	8	1	3	31	/	1	2	165 979
256	$\Pi_{\text{IP-Check}}$	255	3	8	1	3	31	/	1	2	116 252
	Π_{Compress}	63	1	6	4	/	17	2	1	7	43 476

Table 7.6: Cost comparison for $\sigma = 40$, $m = 32768$ with additive secret sharing.

k	Protocol	N	d_0	d_1	s	ν	τ_{in}	τ_{out}	Proof size in kB
32	$\Pi_{\text{Sac-Check}}$	255	1	1	7	/	1	6	2 836
	$\Pi_{\text{IP-Check}}$	255	1	1	7	/	1	6	1 900
	Π_{Compress}	255	1	16	/	8	1	6	945
64	$\Pi_{\text{Sac-Check}}$	255	1	1	7	/	1	6	5 143
	$\Pi_{\text{IP-Check}}$	255	1	1	7	/	1	6	3 439
	Π_{Compress}	255	1	16	/	8	1	6	1 745
256	$\Pi_{\text{Sac-Check}}$	255	1	1	7	/	1	6	18 985
	$\Pi_{\text{IP-Check}}$	255	1	1	7	/	1	6	12 673
	Π_{Compress}	255	1	14	/	4	1	6	6 531

Table 7.7: Cost comparison for $\sigma = 128$, $m = 32768$ with threshold secret sharing.

k	Protocol	N	t	d_0	d_1	s	s_{rc}	ν	τ_{in}	τ_{out}	Proof size in kB
32	$\Pi_{\text{Sac-Check}}$	255	2	8	1	2	23	/	1	9	68 673
	$\Pi_{\text{IP-Check}}$	255	3	8	1	4	39	/	1	6	48 550
	Π_{Compress}	63	1	6	4	/	17	4	1	22	17 158
64	$\Pi_{\text{Sac-Check}}$	255	3	8	1	4	39	/	1	6	130 798
	$\Pi_{\text{IP-Check}}$	255	3	8	1	4	39	/	1	6	91 631
	Π_{Compress}	63	1	6	4	/	17	4	1	22	34 239
256	$\Pi_{\text{Sac-Check}}$	255	3	8	1	4	39	/	1	6	499 875
	$\Pi_{\text{IP-Check}}$	255	3	8	1	4	39	/	1	6	350 119
	Π_{Compress}	63	1	6	4	/	17	2	1	22	136 637

Table 7.8: Cost comparison for $\sigma = 128$, $m = 32768$ with additive secret sharing.

k	Protocol	N	d_0	d_1	s	ν	τ_{in}	τ_{out}	Proof size in kB
32	$\Pi_{\text{Sac-Check}}$	255	1	1	9	/	1	17	8 443
	$\Pi_{\text{IP-Check}}$	255	1	1	9	/	1	17	5 655
	Π_{Compress}	255	1	16	/	8	1	17	2 677
64	$\Pi_{\text{Sac-Check}}$	255	1	1	9	/	1	17	14 980
	$\Pi_{\text{IP-Check}}$	255	1	1	9	/	1	17	10 016
	Π_{Compress}	255	1	16	/	8	1	17	4 944
256	$\Pi_{\text{Sac-Check}}$	255	1	1	9	/	1	17	54 199
	$\Pi_{\text{IP-Check}}$	255	1	1	9	/	1	17	36 179
	Π_{Compress}	255	1	16	/	8	1	17	18 549

7.6 Packing

In this section, we present two orthogonal ways in which our protocols can be extended to provide SIMD-style packing for parallel proofs of multiple independent statements. We then discuss how this packing can be applied to achieve parallelization of proofs for structured circuits.

7.6.1 Packing in the Shamir Domain

The most common way to achieve packing, when using Shamir secret sharing, is to hide multiple secrets in the same polynomial by ensuring the sharing polynomial p evaluates to $p(\alpha_0) = v_0, p(\alpha_1) = v_1, \ldots, p(\alpha_{\ell-1}) = v_{\ell-1}$ when sharing ℓ values, for $\alpha_0, \ldots, \alpha_{\ell+N-1} \in \mathsf{Ex}(\mathcal{R})$. Of course, the shares for the parties should then be evaluations at $\alpha_\ell, \ldots, \alpha_{\ell+N-1}$ in order to preserve privacy.

The degree of p now must become $t + \ell - 1$ to ensure that t parties still learn nothing (including algebraic relations between values) about the shared secrets. This implies that opening a shared value now requires $t + \ell$ shares, rather than the regular $t + 1$. In the context of our protocols however, this does not mean we need to open more commitments towards the verifier since either the opened value is assumed to be known (in the case of $\Pi_{\text{Zero-Check}}$) or provided as part of the proof (in the case of a normal reconstruction). In both of these cases, the additional knowledge effectively acts as ℓ additional known shares at the evaluation points $\alpha_0, \ldots, \alpha_{\ell-1}$.

Applying this technique to our protocols then allows us to prove ℓ separate witnesses for an identical circuit in parallel. The impact on the communication cost is twofold: $\mathsf{Ex}(GR(2^k, d_0))$ should be large enough to allow for $t + \ell$ points and hence $2^{d_0} \geq t + \ell$, and any reconstruction must provide ℓ reconstructed values as part of the proof. Importantly however, the size of sharing of the (extended) witness does not grow, resulting in an approach that is cheaper than performing ℓ separate proofs independently.

7.6.2 Packing in the Galois Domain

Our second approach to packing makes use of the "extra space" that is found in a $GR(2^k, d_0)$ element. Rather than having to send $k \cdot d_0$ bits to represent a single k-bit value, we can send d_0 such values, each in its own coefficient of the Galois ring element, considering it more as a $\mathbb{Z}_{2^k}$-module of dimension d_0.

As long as any operation the parties perform on their shares is an operation for this module (so addition and scalar multiplication by scalars in $\mathbb{Z}_{2^k}$), the actions of the secret sharing and reconstruction are not further impeded. Losing the ability to perform scalar multiplication with values from the entire space $GR(2^k, d_0)$ incurs some cost on the soundness of $\Pi_{\text{Sac-Check}}$ and $\Pi_{\text{IP-Check}}$, where the verifier's random coefficients can now only come from $\mathbb{Z}_{2^k}$ instead, leading to a soundness error of $2^{-(s+1)\cdot d_1}$ rather than $2^{-(s+1)\cdot d_1 \cdot d_0}$.

If Π_{Compress} is used, then it is necessary to deal with the polynomial interpolation needed in Π_{Compress}, which requires *some* scalar multiplication with values coming from an exceptional set of at least size $2 \cdot \nu$. To handle this case, we suggest two possible approaches.

Reducing module dimension. The first approach plays with the same concept described before. It uses the additional free space available in $GR(2^k, d_0)$, but rather than seeing it as a $\mathbb{Z}_{2^k}$-module, it treats it as a $GR(2^k, d_{\mathsf{interp}})$-module of dimension $\frac{d_0}{d_{\mathsf{interp}}}$, subject to $2^{d_{\mathsf{interp}}} \geq 2 \cdot \nu$ to allow for the interpolation.

Tweak the lifting. In the second approach, we tweak the "local lifting" from $GR(2^k, d_0)$ to $GR(2^k, d_0 \cdot d_1)$. Rather than treating the larger ring as a degree d_1 extension of the smaller one, we can choose d_1 such that $\gcd(d_0, d_1) = 1$, and construct the larger ring as a degree d_0 extension of $GR(2^k, d_1)$, even though the input lies in $GR(2^k, d_0)$. To see why this works, we can consider $GR(2^k, d_0 \cdot d_1) = \mathbb{Z}_{2^k}[\beta, \gamma]$, where $GR(2^k, d_0) = \mathbb{Z}_{2^k}[\beta]$ and $GR(2^k, d_1) = \mathbb{Z}_{2^k}[\gamma]$. Due to our restriction that $\gcd(d_0, d_1) = 1$, β and γ are algebraically independent, allowing us to reinterpret $\mathbb{Z}_{2^k}[\beta, \gamma] = (\mathbb{Z}_{2^k}[\beta])[\gamma] = (\mathbb{Z}_{2^k}[\gamma])[\beta]$. In the interpretation $(\mathbb{Z}_{2^k}[\gamma])[\beta]$, we are now left with a form that allows us to treat these values as a $GR(2^k, d_1)$-module of dimension d_0. When doing this, the only further constraint we have is that $2^{d_1} \geq 2 \cdot \nu$, while being able to fully pack all d_0 input coefficients.

These approaches incur some loss in soundness, resulting in a cheating probability for the multiplication checks of $2^{-d_1 \cdot \frac{d_0}{d_{\mathsf{interp}}}}$ or 2^{-d_1} respectively.

Although the loss in soundness necessitates more communication to return to the same level of security, the reduction in communication when averaged over the parallel proof instances brings some benefits. When performing d_0 proofs in parallel, neither the communication for the input of the extended witness, nor the communication to reconstruct a secret-shared value increase. When all coefficients in the input sharing are filled with actual inputs, we also no longer need to perform $\Pi_{\text{Ring-Check}}$, as all $GR(2^k, d_0)$ elements now correspond to a valid set of d_0 elements in $\mathbb{Z}_{2^k}$.

We also considered the use of *Reverse Multiplication-Friendly Embeddings* (RMFEs), as introduced by [CCXY18], for this sort of packing, but since this only provides $\mathbb{Z}_{2^k}$-linearity, it is incompatible with our threshold secret sharing. Additionally, RMFEs only provide a constant packing rate, whereas our technique succeeds in utilizing the available space maximally.

7.6.3 Multi-Round Computations

Instead of proving some ℓ independent instances of a circuit in parallel, one would often prefer to use this packing to prove a single instance more efficiently, either by performing multiple of the "outer" repetitions in parallel, or by performing multiple gates of the circuit in parallel. As the challenges provided by the verifier are shared across the parallel instances being proved, the former is unfortunately not possible. The latter however, can be achieved by introducing a gadget that checks whether two secret shared values $[\![a]\!]$ and $[\![b]\!]$ are (prescribed) permutations.

Depending on the efficiency of such a check, this could allow optimizing for circuits that are *wide* enough (that is, circuits that perform enough independent multiplications in parallel), to only allowing optimization for circuits that are highly structured. As an example of such structured circuits, one could consider a computation that proceeds in several identical rounds, such as a circuit that performs several consecutive RAM accesses like in Section 7.7. In an ideal scenario, the permutation check can be performed mostly entirely locally, with a final $\Pi_{\text{Zero-Check}}$ at the end of the protocol, yielding an improvement in communication cost of factor ℓ practically for free. For the permutation checks we will describe here, a highly structured/repetitive circuit should be preferred, however.

To check the reordering of a Shamir packed secret sharing, each party can *re-share* their share and enable a private reconstruction of the underlying secrets, which can then be re-ordered and eventually checked in batch with a random linear combination and $\Pi_{\text{Zero-Check}}$. This results in $t \cdot (2 \cdot N - t)$ ring elements of communication to perform the re-sharing. To check the reordering of Galois coefficients in $GR(2^k, d)$, we let the prover inject d sharings of $\mathbb{Z}_{2^k}$ elements M_i (which need to be checked through $\Pi_{\text{Ring-Check}}$), which can be used to mask corresponding coefficients in a and b identically and provide privacy of the values. Then we can perform a (batched) $\Pi_{\text{Zero-Check}}(a + \sum_i x^i M_i - b - \sum_i x^{\pi(i)} M_i)$ to validate the permutation. This incurs a cost of d ring elements per permutation check to input the mask values M_i.

7.7 RAM Application

In this section, we show how to construct the C_{check} circuit of [DOTV22] to verify the consistency of a series of T read or write accesses to an initial array $\mathcal{L}$ of size N. Our C_{check} circuit is very similar to that of [DOTV22] albeit with minor modifications to fit our ring structure. In particular, we cannot use the EqCheck sub-circuit that crucially relies on the underlying field structure and we tweak the PermCheck to use the Generalized Schwartz-Zippel (Lemma 7.1). In addition, we assume a large exceptional set. In all the sub-circuits of this section, we overload the notation $[\![.]\!]$ to denote sensitive values that cannot be revealed in the zero-knowledge proof.

First, we introduce the main building blocks, i.e. PermCheck and BdCheck, and later in 7.7.3, we describe the ring version of C_{check}.

7.7.1 Permutation Check

First, we design a procedure PermCheck, see Figure 7.7, to verify that two arrays $([\![a_1]\!], \ldots, [\![a_S]\!])$ and $([\![b_1]\!], \ldots, [\![b_S]\!])$ of S shared elements are one a permutation of the other. The idea behind the check is to define two polynomials $P_A(X) = \prod_{i\in[S]}(X - a_i)$ and $P_B(X) = \prod_{i\in[S]}(X - b_i)$ which are identical if and only if both arrays are a permutation of each other, and then use polynomial identity testing to verify this is indeed the case. Both polynomials P_A and P_B are of degree S, thus the Generalized Schwartz-Zippel (Lemma 7.1) states that if A is not a permutation of B (i.e. $P_A \neq P_B$), the check passes with probability at most $\frac{S}{2^{d_0 \cdot d_1}}$.

In addition, we also describe another procedure, given in Figure 7.8, for when the a_i and b_i are themselves tuples of 4 elements — looking ahead, the array to be checked consists of tuples of 4 elements. This protocol is similar to the previous one, except we first compress our tuple into a single element. Assuming that A and B are not a permutation of each other, then for all permutations π there exists at least one tuple $(a_i^{(1)}, a_i^{(2)}, a_i^{(3)}, a_i^{(4)})$ and one tuple $(b_{\pi(i)}^{(1)}, b_{\pi(i)}^{(2)}, b_{\pi(i)}^{(3)}, b_{\pi(i)}^{(4)})$ that differs. The probability that such tuples are compressed into a_i and $b_{\pi(i)}$ respectively such that $a_i = b_{\pi(i)}$ is bounded by the Generalized Schwartz-Zippel lemma for 4-variate polynomial of total degree 4 by $\frac{4}{2^{d_0 \cdot d_1}}$. By union bound, the check thus passes with probability at most $\frac{S+4}{2^{d_0 \cdot d_1}}$.

PermCheck

Inputs: $[\![A]\!] = ([\![a_1]\!], \ldots, [\![a_S]\!])$ and $[\![B]\!] = ([\![b_1]\!], \ldots, [\![b_S]\!])$ both over $GR(2^k, d_0)$

Protocol:

1. Lift $[\![a_i]\!]$ and $[\![b_i]\!]$ from $GR(2^k, d_0)$ to $GR(2^k, d_0 \cdot d_1)$.
2. $s \leftarrow \mathcal{O}_R$ such that $s \in \mathsf{Ex}(GR(2^k, d_0 \cdot d_1))$.
3. Add the $S-1$ multiplication gates necessary to compute $[\![P_A(s)]\!] = \Pi_{i \in [S]}(s - [\![a_i]\!])$ and similarly for $[\![P_B(s)]\!] = \Pi_{i \in [S]}(s - [\![b_i]\!])$.
4. Add $[\![P_A(s)]\!] - [\![P_B(s)]\!]$ to the list of outputs.

Figure 7.7: Permutation check

PermCheck for Tuples

Inputs: $[\![A]\!] = (([\![a_1^{(1)}]\!], [\![a_1^{(2)}]\!], [\![a_1^{(3)}]\!], [\![a_1^{(4)}]\!]) \ldots, ([\![a_S^{(1)}]\!], \ldots, [\![a_S^{(4)}]\!]))$ and $[\![B]\!] = (([\![b_1^{(1)}]\!], [\![b_1^{(2)}]\!], [\![b_1^{(3)}]\!], [\![b_1^{(4)}]\!]) \ldots, ([\![b_S^{(1)}]\!], \ldots, [\![b_S^{(4)}]\!]))$ both shared over $GR(2^k, d_0)$

Protocol:

1. Lift $[\![a_i^{(j)}]\!]$ and $[\![b_i^{(j)}]\!]$ from $GR(2^k, d_0)$ to $GR(2^k, d_0 \cdot d_1)$.
2. $r = (r^{(1)}, r^{(2)}, r^{(3)}, r^{(4)}) \leftarrow \mathcal{O}_R$ such that $r^{(j)} \in \mathsf{Ex}(GR(2^k, d_0 \cdot d_1))$
3. For $i \in [S]$ add the linear gates to compute $[\![a_i]\!] = \Sigma_{j \in [4]} [\![a_i^{(j)} \cdot r^{(j)}]\!]$ and $[\![b_i]\!] = \Sigma_{j \in [4]} [\![b_i^{(j)} \cdot r^{(j)}]\!]$
4. $s \leftarrow \mathcal{O}_R$ such that $s \in \mathsf{Ex}(GR(2^k, d_0 \cdot d_1))$.
5. Add the $S-1$ multiplication gates necessary to compute $[\![P_A(s)]\!] = \Pi_{i \in [S]}(s - [\![a_i]\!])$ and similarly for $[\![P_B(s)]\!] = \Pi_{i \in [S]}(s - [\![b_i]\!])$.
6. Add $[\![P_A(s)]\!] - [\![P_B(s)]\!]$ to the list of outputs.

Figure 7.8: Permutation check for tuples

7.7.2 Bound Check

The bound check BdCheck is exactly the same as [DOTV22]. For completeness, we recall it in Figure 7.9. It checks in zero-knowledge that a set of T sensitive values are contained between two public bounds, B_1, B_1, with $B_1 < B_2$.

BdCheck

Input: The lower and upper bounds $B_1 < B_2$.
$[\![\mathcal{L}]\!] = [B_1, B_1 + 1, \dots, B_2, [\![x_1]\!], \dots, [\![x_T]\!]]$ of size S
$[\![\mathcal{L}']\!]$ that contains the entries of $\mathcal{L}$ sorted from lowest to highest (with all the entries sensitive)

Protocol:

```
1. $[\![\mathsf{is\_permutation}]\!] \leftarrow \mathsf{PermCheck}([\![\mathcal{L}]\!], [\![\mathcal{L}']\!])$
2. For $i \in [S-1]$
    (a) $[\![\alpha_i]\!] \leftarrow [\![\mathcal{L}'[i+1]]\!] - [\![\mathcal{L}'[i]]\!]$
    (b) $[\![\lambda_i]\!] \leftarrow [\![\alpha_i]\!] \cdot [\![1-\alpha_i]\!]$.
3. Add all the following to the list of outputs:
    • $[\![\mathsf{is\_permutation}]\!]$
    • $[\![\lambda_i]\!]$ for $i \in [S-1]$
    • $[\![\mathcal{L}'[1]]\!] - B_1$
    • $[\![\mathcal{L}'[S]]\!] - B_2$
```

Figure 7.9: Bound Check for a batch of sensitive values

7.7.3 Array Access Check

We now describe our version of C_{check}, see Figure 7.10. We assume the memory has N slots and is first initialized with sensitive values M_i. The array $\mathcal{L}$ consists of tuples of the form

$$(\underbrace{\mathsf{memory\_address}}_{\ell}, \underbrace{\mathsf{global\_timestamp}}_{t}, \underbrace{\mathsf{operation}}_{\mathsf{op}}, \underbrace{\mathsf{data}}_{d}).$$

Here, $\ell \in [N]$, $t \in [N+T]$, $\mathsf{op} \in \{0,1\}$ (0 for read, 1 for write), and d is the data that has been read or written.

Intuition Behind the Check: The protocol takes as input the initial array M arranged into a list $\mathcal{L}$ as described before. The list of tuples is sorted first by the address ℓ, and then by the timestamp t, forming a list $\mathcal{L}'$ which consists

C_{check}

Input: $[\![\mathcal{L}]\!] = [(1,1,1,[\![M_1]\!]),\dots(N,N,1,[\![M_N]\!]),$
$([\![l_{N+1}]\!],N+1,[\![\mathsf{op}_{N+1}]\!],[\![d_{N+1}]\!]),\dots,([\![l_{N+T}]\!],N+T,[\![\mathsf{op}_{N+T}]\!],[\![d_{N+T}]\!])]$
$[\![\mathcal{L}']\!]$ containing entries of $\mathcal{L}$ sorted first by ℓ then by t.

Protocol:

```
1. $[\![\mathsf{is\_permutation}]\!] \leftarrow \mathsf{PermCheck}([\![\mathcal{L}]\!],[\![\mathcal{L}']\!])$
2. For $i \in [N+T-1]$ do
   (a) $[\![\alpha_i]\!] \leftarrow 1-([\![\ell'_{i+1}]\!]-[\![\ell'_i]\!])$
   (b) $[\![\lambda_i]\!] \leftarrow [\![\alpha_i]\!]\cdot[\![1-\alpha_i]\!]$.
   (c) $[\![\tilde{\tau}_i]\!]_j \leftarrow [\![\alpha_i]\!],[\![t'_{i+1}-t'_i]\!]$ and $[\![\tau_i]\!] \leftarrow [\![\tilde{\tau}_i]\!]+(1-[\![\alpha_i]\!])$.
   (d) $[\![\zeta_i]\!] \leftarrow [\![\mathsf{op}_i]\!]\cdot[\![1-\mathsf{op}_i]\!]$.
   (e) $[\![\beta_i]\!] \leftarrow [\![d'_i]\!]-[\![d'_{i+1}]\!]$.
   (f) $[\![\tilde{\gamma}_i]\!] \leftarrow [\![\alpha_i]\!]\cdot[\![\beta_i]\!]$ and $[\![\gamma_i]\!] \leftarrow [\![\tilde{\gamma}_i]\!]\cdot(1-[\![\mathsf{op}_{i+1}]\!])$.
3. $[\![\mathsf{is\_in\_bound}]\!] \leftarrow \mathsf{BdCheck}((\{[\![\tau_i]\!]\})_{i\in[N+T-1]},1,N+T-1)$
4. Add all the following to the list of outputs
   • $[\![\mathsf{is\_permutation}]\!]$
   • $[\![\lambda_i]\!]$ for $i \in [N+T-1]$
   • $[\![\gamma_i]\!]$ for $i \in [N+T-1]$
   • $[\![\zeta_i]\!]$ for $i \in [N+T-1]$
   • $[\![\mathsf{is\_in\_bound}]\!]$
   • $N-[\![\ell'_{N+T}]\!]$
```

Figure 7.10: Complete checking circuit for random memory accesses

of contiguous blocks for each address $\ell = 1,\dots,N$ that list the consecutive accesses to the same address ℓ sorting chronologically starting with writing the initial value M_ℓ.

We need to check the following conditions hold:

- Each block concerns one valid address and all addresses are covered
- Inside each block, the instructions are ordered by their timestamp
- If the operation is read, then the read value matches the previous value at that address

- Each operation is either a read or a write.

The used variables carry the following meaning:

- $\alpha_i = 1$ if and only if $\ell'_i = \ell'_{i+1}$ and 0 otherwise, i.e., when the next tuple describes an access to the same address
- $\lambda_i = 0$ if and only if $\alpha_i \in \{0, 1\}$
- τ_i is the difference between the timestamps of subsequent accesses otherwise, $\tau_i = 1$
- $\zeta_i = 0$ if and only if $\mathsf{op}_i \in \{0, 1\}$
- β_i is the difference between the data $d'_i - d'_{i+1}$ which is supposed to be 0 if the next tuple is a read instruction
- $\gamma_i = \beta_i$ if and only if op_{i+1} is a read operation to the same address; therefore it is supposed to be zero.

Changes Compared to [DOTV22]: The protocol of [DOTV22] uses the so-called EqCheck circuit, that takes to shared values $[\![x]\!]$, $[\![y]\!]$ and outputs a shared bit $[\![b]\!]$ such that $b = 1$ if and only if $x = y$. We cannot use the EqCheck circuit in our setting, since it relies on the existence of inverses of arbitrary non-zero elements. Hence, we introduce some changes:

- Changes to the α_i:
  - Used to be $\mathsf{EqCheck}(\ell'_\mathsf{i}, \ell'_\mathsf{i+1})$.
  - Now is $1 - (\ell'_{i+1} - \ell'_i)$ and check $\alpha_i \in \{0, 1\}$ with λ_i.
- Changes to the β_i:
  - Used to be $\mathsf{EqCheck}(\mathsf{d}'_\mathsf{i}, \mathsf{d}'_\mathsf{i+1})$.
  - Now is $d'_i - d'_{i+1}$.

Zero-knowledge. Replacing α_i this way does not impact zero-knowledge as for a honest proof, consecutive memory addresses are at most 1 apart. Replacing β_i does not impact zero-knowledge either as it only appears in γ_i when both $\alpha_i = 1$ and $\mathsf{op}'_{i+1} = 0$ (i.e. read), in which case for an honest proof we expect $\beta_i = 0$.

Soundness. Replacing α_i does not impact soundness as it is still an equality check as we ensure $\alpha_i \in \{0, 1\}$ with λ_i. Replacing β_i does not impact soundness either as for $\alpha_i = 0$ or $\mathsf{op}'_{i+1} = 1$, we allow d'_i and d'_{i+1} to be arbitrary and when $\alpha_i = 1$ and $\mathsf{op}'_{i+1} = 0$ we deterministically ensure $d'_i = d'_{i+1}$ with the $\Pi_{\text{Zero-Check}}$ on γ_i.

Acknowledgements

The work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085, by CyberSecurity Research Flanders with reference number VR20192203, by the FWO under an Odysseus project GOH9718N, and by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No. 803096 (SPEC). The work of the last author was conducted whilst they were a PhD student at KU Leuven.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the US Government, Cyber Security Research Flanders or the FWO. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

Bibliography

[ACD+19] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Cham, December 2019.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[BBC+19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Cham, August 2019.

[BBMH$^+$21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 192–211. ACM Press, November 2021.

[BBMHS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358. Springer, Cham, August 2022.

[BDK$^+$21] Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 266–297. Springer, Cham, May 2021.

[BN20] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 495–526. Springer, Cham, May 2020.

[CCKP19] Shuo Chen, Jung Hee Cheon, Dongwoo Kim, and Daejun Park. Verifiable computing for approximate computation. Cryptology ePrint Archive, Report 2019/762, 2019.

[CCXY18] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 395–426. Springer, Cham, August 2018.

[CDE$^+$18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Cham, August 2018.

[CDG$^+$17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[DOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021.

[DOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 615–638. Springer, Cham, September 2022.

[EGK$^+$20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020.

[EXY22] Daniel Escudero, Chaoping Xing, and Chen Yuan. More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 383–412. Springer, Cham, August 2022.

[Feh98] Serge Fehr. Span programs over rings and how to share a secret from a module, 1998. MSc Thesis, ETH Zurich.

[FMRV22] Thibauld Feneuil, Jules Maire, Matthieu Rivain, and Damien Vergnaud. Zero-knowledge protocols for the subset sum problem from MPC-in-the-head with rejection. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 371–402. Springer, Cham, December 2022.

[FR22] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987.

[GHAH$^+$23] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *PoPETs*, 2023(1):627–640, January 2023.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JSv22] Robin Jadoul, Nigel P. Smart, and Barry van Leeuwen. MPC for Q_2 access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 131–151. Springer, Cham, September / October 2022.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[KZ22] Daniel Kales and Greg Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Report 2022/588, 2022.

[LXY23] Fuchun Lin, Chaoping Xing, and Yizhou Yao. More efficient zero-knowledge protocols over $\mathbb{Z}_{2^k}$ via galois rings. Cryptology ePrint Archive, Report 2023/150, 2023.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SS23] Victor Shoup and Nigel P. Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. Cryptology ePrint Archive, Paper 2023/536, 2023. https://eprint.iacr.org/2023/536.

CHAPTER 8

Conclusion

In this thesis, we have tried to build bridges over the gap in algebraic structure of preference between mathematicians and computers. Due to the differences in computation each structure is optimized for, it is hard to unequivocally claim our protocols are better than those that work only over finite fields, but it is our hope and belief that it is a worthwhile direction of research. The strive for constructions over $\mathbb{Z}_{p^k}$ and $\mathbb{Z}_{2^k}$ in particular allows us to bypass the need for costly emulation of this structure within finite fields and may bring the benefits of MPC and ZK protocols closer to a general programming model and within reach of software developers who may not possess the required mathematical or cryptological background. The work presented here contributes both new and improved constructions in this area, as well as a consolidation, generalization and evaluation of previous work from the literature.

In our attempts for more efficient protocols, we have successfully applied linear secret sharing schemes, both in their most general form as MSPs and in the more commonly deployed setting of threshold structures. Our applications follow both the usual structure for building MPC protocols for arithmetic circuits, and explore the newer application of threshold secret sharing to the MPC-in-the-Head paradigm, as well as its interactions with the less studied constructions over rings.

Finally, in our search for concretely efficient protocols, we have examined the tradeoff between the public viability of Zero-Knowledge Proofs, and the efficiency of the prover. In doing so, we constructed concretely efficient ZKP protocols with a distributed designated verifier. Our protocols achieve an efficient prover with succinct verifier communication and identifiable abort for arithmetic circuits over arbitrary finite fields in a setting with an honest supermajority.

Future work

While we make several contributions towards building efficient cryptographic protocols, the work is never done. There are always more opportunities for protocol-level improvements that deserve further investigation. The techniques used by our protocols described in chapter 5, for instance, may deserve further attention in different security contexts, such as active security with identifiable abort or guaranteed output delivery for $\mathcal{Q}_3$ access structures.

With regard to our distributed verifier zero-knowledge proofs from chapter 6, several follow-up questions may arise. In the context of this thesis, a first natural consideration would be to extend our protocols to the ring setting, adapting our techniques for $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_{p^k}$ for DVZK and adapting them appropriately to the somewhat different performance characteristics and requirements inherent to a distributed verifier.

Another deficiency of our DVZK protocol is its limitation to a threshold $t < \frac{N}{3}$. By leveraging more cryptographic machinery, it would be interesting to extend this to a regular honest majority $t < \frac{N}{2}$, or even a full-threshold situation. One of the main hurdles to achieve this will be the demand for identifiable abort in our security definition, as this fits most naturally with the thresholds we already achieved.

Recently, a new construction, known as VOLE-in-the-Head,[1] has been successfully applied to VOLE-based designated-verifier ZK protocols with short (though mostly still linear) proof sizes and good prover efficiency; achieving public verifiability. A concrete analysis of the differences in performance between these new publicly verifiable proof systems and our distributed verifier system may be needed to shed further light on the tradeoffs between the two, and to determine when which option is the correct choice.

A further question open to future work is concerned with the proof size of MPCitH-based constructions — among which we can also count Feta, as it shows many similarities. While the concrete overhead of the proof size over the number of multiplication gates in the statement circuit is often only a small constant, the fact that the proof size grows linearly in the size of the circuit remains potentially problematic for large statements and applications to verifiable computation. Asking a verifier with limited computational power to perform a computation of similar size to the original circuit is, after all, in many cases not an option. Therefore, ZK proof systems that can achieve sublinear proof size and verification time, without sacrificing the concrete

[1] VOLE, or *Vector Oblivious Linear function Evaluation*, can in short be understood as a method for batch generating correlated randomness between two parties.

(prover) efficiency advantages offered by the MPC-in-the-Head paradigm, may be highly coveted, both for in the regular publicly verifiable as the distributed verifier case.

Finally, looking back at our original inspiration to bring these cryptographical protocols closer to the execution model of a regular computer, we may notice an oversight. While indeed additions and multiplications in a CPU correspond to the same operations on the ring $\mathbb{Z}_{2^k}$, the computer is not limited to performing only additions and multiplications. It is also able to look at the individual bits within those numbers, extract and recompose them, and even perform bitwise operations such as XOR, AND, and OR on them. By making the protocols work with bits (or $\mathbb{F}_2$ elements, equivalently) as its basic building blocks, existing protocols are able to perform these operations, and emulate the $\mathbb{Z}_{2^k}$ arithmetic with bits. However, an approach that can combine the decomposition into bits with native support for ring arithmetic without an additional logarithmic overhead has not been achieved yet. Such a construction, if possible in a secure manner, would enable a more faithful cryptographical execution of “normal” programs, and bring these powerful building blocks closer to broad adoption and general utility.

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. IEEE Computer Society Press, May 2017.

[ACD$^+$19] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Cham, December 2019.

[ACD$^+$20] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, Matthieu Rambaud, Chaoping Xing, and Chen Yuan. Asymptotically good multiplicative LSSS over Galois rings and applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 151–180. Springer, Cham, December 2020.

[ACF02] Masayuki Abe, Ronald Cramer, and Serge Fehr. Non-interactive distributed-verifier proofs and proving relations among commitments. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 206–223. Springer, Berlin, Heidelberg, December 2002.

[ADEN19] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. Cryptology ePrint Archive, Report 2019/1298, 2019.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[AKP22] Benny Applebaum, Eliran Kachlon, and Arpita Patra. Verifiable relation sharing and multi-verifier zero-knowledge in two rounds: Trading NIZKs with honest majority. Cryptology ePrint Archive, Report 2022/167, 2022.

[BBC$^+$19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Cham, August 2019.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Cham, August 2019.

[BBMH$^+$21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 192–211. ACM Press, November 2021.

[BBMHS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358. Springer, Cham, August 2022.

[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*, volume 5628 of *LNCS*, pages 325–343. Springer, Berlin, Heidelberg, February 2009.

[BCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Berlin, Heidelberg, August 2013.

[BD91] Mike Burmester and Yvo Desmedt. Broadcast interactive proofs (extended abstract). In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 81–95. Springer, Berlin, Heidelberg, April 1991.

[BDK$^+$21] Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 266–297. Springer, Cham, May 2021.

[BdSGJ$^+$24] Lennart Braun, Cyprien Delpech de Saint Guilhem, Robin Jadoul, Emmanuela Orsini, Nigel P. Smart, and Titouan Tanguy. ZK-for-Z2K: MPC-in-the-Head Zero-Knowledge Proofs for $\mathbb{Z}_{2^k}$. In Elizabeth A. Quaglia, editor, *Cryptography and Coding*, pages 137–157, Cham, 2024. Springer Nature Switzerland.

[Bea91] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, January 1991.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Berlin, Heidelberg, August 1992.

[BGKW88] Michael Ben-Or, Shafi Goldwasser, Joe Kilian, and Avi Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *20th ACM STOC*, pages 113–131. ACM Press, May 1988.

[BJO$^+$22] Carsten Baum, Robin Jadoul, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. Feta: Efficient threshold designated-verifier zero-knowledge proofs. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 293–306. ACM Press, November 2022.

[BKZZ20] Foteini Baldimtsi, Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. Crowd verifiable zero-knowledge and end-to-end verifiable multiparty computation. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 717–748. Springer, Cham, December 2020.

[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*, volume 5283 of *LNCS*, pages 192–206. Springer, Berlin, Heidelberg, October 2008.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Cham.

[BN20] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 495–526. Springer, Cham, May 2020.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Berlin, Heidelberg, March 2008.

[CCKP19] Shuo Chen, Jung Hee Cheon, Dongwoo Kim, and Daejun Park. Verifiable computing for approximate computation. Cryptology ePrint Archive, Report 2019/762, 2019.

[CCXY18] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 395–426. Springer, Cham, August 2018.

[CDE$^+$18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Cham, August 2018.

[CDG$^+$17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Berlin, Heidelberg, February 2005.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, Berlin, Heidelberg, May 2000.

[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, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Cham, August 2018.

[CRX19] Ronald Cramer, Matthieu Rambaud, and Chaoping Xing. Asymptotically-good arithmetic secret sharing over $Z/(p^\ell Z)$ with strong multiplication and its applications to efficient MPC. Cryptology ePrint Archive, Report 2019/832, 2019.

[DDOS19] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P. Smart. BBQ: Using AES in picnic signatures. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 669–692. Springer, Cham, August 2019.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Berlin, Heidelberg, August 2007.

[DOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021.

[DOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 615–638. Springer, Cham, September 2022.

[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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.

[EGK$^+$20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020.

[EKO$^+$20] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 5:1–5:24. Schloss Dagstuhl, June 2020.

[EXY22] Daniel Escudero, Chaoping Xing, and Chen Yuan. More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 383–412. Springer, Cham, August 2022.

[Feh98] Serge Fehr. Span programs over rings and how to share a secret from a module, 1998. MSc Thesis, ETH Zurich.

[FMRV22] Thibauld Feneuil, Jules Maire, Matthieu Rivain, and Damien Vergnaud. Zero-knowledge protocols for the subset sum problem from MPC-in-the-head with rejection. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 371–402. Springer, Cham, December 2022.

[FR22] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987.

[GHAH$^+$23] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *PoPETs*, 2023(1):627–640, January 2023.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[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 ACM STOC*, pages 218–229. ACM Press, May 1987.

[Gá95] Anna Gál. Combinatorial methods in boolean function complexity, 1995. PhD Theses, University of Chicago.

[HBHW16] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[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 PODC*, pages 25–34. ACM, August 1997.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[JSv22] Robin Jadoul, Nigel P. Smart, and Barry van Leeuwen. MPC for Q_2 access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 131–151. Springer, Cham, September / October 2022.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.

[KGC$^+$18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[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*, pages 830–842. ACM Press, October 2016.

[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*, volume 11035 of *LNCS*, pages 181–199. Springer, Cham, September 2018.

[KZ22] Daniel Kales and Greg Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Report 2022/588, 2022.

[KZF$^+$18] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. Commit-Chains: Secure, scalable off-chain payments. Cryptology ePrint Archive, Report 2018/642, 2018.

[LMs05] Matt Lepinski, Silvio Micali, and abhi shelat. Fair-zero knowledge. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 245–263. Springer, Berlin, Heidelberg, February 2005.

[LSTW21] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/030, 2021.

[LXY23] Fuchun Lin, Chaoping Xing, and Yizhou Yao. More efficient zero-knowledge protocols over $\mathbb{Z}_{2^k}$ via galois rings. Cryptology ePrint Archive, Report 2023/150, 2023.

[Mau06] Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[Mau15] Ueli Maurer. Zero-knowledge proofs of knowledge for group homomorphisms. *DCC*, 77(2-3):663–676, 2015.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, New York, August 1990.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SS23] Victor Shoup and Nigel P. Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. Cryptology ePrint Archive, Paper 2023/536, 2023. https://eprint.iacr.org/2023/536.

[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, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 210–229. Springer, Cham, March 2019.

[Unr15] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 755–784. Springer, Berlin, Heidelberg, April 2015.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[WZC$^+$18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 675–692. USENIX Association, August 2018.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[YW22] Kang Yang and Xiao Wang. Non-interactive zero-knowledge proofs to multiple verifiers. Cryptology ePrint Archive, Report 2022/063, 2022.

Statement on the Use of Generative AI

I did not use generative AI assistance tools during the research/writing process of my thesis.

The text/code/images in this thesis are my own (unless otherwise specified) and generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

Curriculum

Robin Jadoul obtained a Bachelor's degree in computer science from the University of Antwerp, Belgium, in 2018. He then continued his studies at ETH Zürich, Switzerland, where he graduated with a Master's degree in computer science, with a major in theoretical computer science, in 2020. His Master's thesis handled on the cryptographic topic of deniability in secure messaging protocols.

Subsequently, he joined the COSIC research group of KU Leuven, Belgium, as a PhD candidate under the supervision of prof. Nigel P. Smart, where he works on cryptographic protocols for computation on encrypted data and privacy-enhancing technologies. His main focus lies in the areas of secure multiparty computation and zero-knowledge proofs through the MPC-in-the-Head paradigm.

List of publications

- Robin Jadoul, Nigel P. Smart, and Barry van Leeuwen. MPC for Q_2 access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 131–151. Springer, Cham, September / October 2022
- Carsten Baum, Robin Jadoul, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. Feta: Efficient threshold designated-verifier zero-knowledge proofs. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 293–306. ACM Press, November 2022
- Lennart Braun, Cyprien Delpech de Saint Guilhem, Robin Jadoul, Emmanuela Orsini, Nigel P. Smart, and Titouan Tanguy. ZK-for-Z2K: MPC-in-the-Head Zero-Knowledge Proofs for $\mathbb{Z}_{2^k}$. In Elizabeth A. Quaglia, editor, *Cryptography and Coding*, pages 137–157, Cham, 2024. Springer Nature Switzerland
- Robin Jadoul, Axel Mertens, Jeongeun Park, and Hilder V. L. Pereira. NTRU-Based FHE for Larger Key and Message Space. In Tianqing Zhu and Yannan Li, editors, *Information Security and Privacy*, pages 141–160, Singapore, 2024. Springer Nature Singapore

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Kasteelpark Arenberg 10, bus 2452
B-3001 Leuven
robin.jadoul@esat.kuleuven.be
https://esat.kuleuven.be/cosic/