




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

Robin Jadoul¹, Nigel P. Smart¹, and Barry Van Leeuwen¹

imec-COSIC, KU Leuven, Leuven, Belgium.

`robin.jadoul@esat.kuleuven.be`, `nigel.smart@kuleuven.be` `barry.vanleeuwen@kuleuven.be`

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 realised 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.

Table of Contents

MPC for \mathbb{Q}_2 Access Structures over Rings and Fields	1
<i>Robin Jadoul^{ID}, Nigel P. Smart^{ID}, and Barry Van Leeuwen^{ID}</i>	
1 Introduction	3
2 Preliminaries	7
2.1 Notation	7
2.2 ℓ -Good Rings and the Schwartz-Zippel Lemma	7
2.3 Monotone and Extended Span Programs	8
2.4 Linear Secret Sharing Schemes Induced from MSPs and ESPs	10
2.5 Shamir over Rings, an Example:	11
2.6 Basic Multi-Party Computation Protocols	12
3 Generating an ESP from an MSP	16
4 Opening Values to One Player and to All Players	18
4.1 Open to One	19
4.2 Open to All	21
5 Multiplication Check	22
5.1 MultCheck ₁	23
5.2 MultCheck' ₁	24
5.3 MultCheck ₂	25
5.4 MacCheck	26
5.5 Summary	27
6 Offline Preprocessing Protocols	29
6.1 Comparing Actively Secure Offline Protocols	32
7 Complete Protocols	33
A Proof of Theorem 2.1	37
B KRSW Multiplication Costs	39
B.1 Replicated (3, 1) Sharing	43
B.2 Replicated (5, 2) Sharing	44
B.3 Replicated (10, 4) Sharing	46
B.4 Shamir (3, 1) for large p	47
B.5 Shamir (5, 2) for large p	49
B.6 Shamir (10, 4) for large p	51
B.7 Shamir (3, 1) for \mathbb{Z}_{2^k}	54
B.8 Shamir (5, 2) for \mathbb{Z}_{2^k}	56
B.9 Shamir (10, 4) for \mathbb{Z}_{2^k}	59

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 paper 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 realised, using a variety of methodologies to obtain active-security-with-abort. The natural generalisation 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 paper 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 generalisation 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 protocols¹ 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 generalise² to an arbitrary setting of p^k . The first, which we denote by Offline₁, 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

¹ Note the kind-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.

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 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 [EKO+20].

A third offline variant, which we shall denote by 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 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 generalisation 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 generalise 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 Protocol_2), or we can completely remove the need for a preprocessing step (which we dub 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 Protocol_4 .

The final protocol variant we consider, which we dub 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 1 as a means for the reader to maintain a quick overview as they read the paper.

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

Protocol	Offline Phase		Online Phase	Post-Processing	
	Passive	Active		Heavy	Light
Protocol ₁	-	✓	Beaver	-	-
Protocol ₂	✓	-	Passive	✓	-
Protocol ₃	-	-	Passive	✓	-
Protocol ₄	-	✓	Passive	✓	-
Protocol ₅	-	-	2 × Passive	-	✓

Table 1. Summary of our five protocol variants. A “heavy” post-processing phase denotes a phase akin to sacrificing, where as 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.

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 manner³.

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 Schwarz-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 generalise 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

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.

the total amount of data sent per multiplication⁴. 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).

Paper Outline: In Section 2 we summarize some basic definitions and work from other papers which we will utilize. In Section 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 4 we generalize the results on \mathcal{Q}_2 MSPs over \mathbb{F}_p of [SW19], to ESPs over \mathbb{Z}_{p^k} . This enables us to open values to players, and ensure the opened values are “correct”. Then in Section 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 6 (resp. Section 7) we examine the various offline (resp. complete) protocols and do a comparison.

2 Preliminaries

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 its i th 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 $\ker(M)$ denote the subspace of K^m which maps to $\mathbf{0}$ under left multiplication by M , and we let $\text{Im}(M)$ denote the subspace of K^n which is the image of all elements in K^m 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 i th canonical basis vector, that is $\mathbf{e}_i^{(j)} = \delta_{i,j}$ where δ is the Kronecker Delta.

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

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

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.

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 2.1 (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, $\omega_1, \dots, \omega_{d+1} \in S$. 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. \square

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 paper in access structures which are \mathcal{Q}_2 :

Definition 2.1 (\mathcal{Q}_2 Access Structure). *Let $\mathcal{P} = \{P_1, \dots, P_n\}$ 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 $\mathcal{P} \setminus Q \in \Delta$ and vice versa. In this paper we will only consider complete access structures.

Monotone Span Programs over Fields: Using this notation, the definition of a Monotone Span Program follows.

Definition 2.2. 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 $\varphi : [m] \rightarrow \mathcal{P}$ 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 **supp** : $\mathbb{F}^d \rightarrow 2^{[d]}$ with $\mathbf{s} \mapsto \{i \in [d] : \mathbf{s}^{(i)} \neq 0\}$.

Definition 2.3. 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

$$A \in \Gamma \Rightarrow \varepsilon \in \text{Im}(M_A^T), \quad (1)$$

$$A \notin \Gamma \Rightarrow \varepsilon \notin \text{Im}(M_A^T). \quad (2)$$

Note that this could be presented as a single if and only if condition, but to emphasise the difference with the generalization to rings, we present the condition in two equations. Also note that, an equivalent formulation for requirement (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 paper 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 $\varphi : [m] \rightarrow \mathcal{P}$ is a surjective map of the rows of M to the parties in \mathcal{P} . The analogue of Definition 2.3 is

Definition 2.4. An ESP \mathcal{M} is to compute an access structure (Γ, Δ) if for every set $A \subset 2^{\mathcal{P}}$ it holds that

$$A \in \Gamma \Rightarrow \varepsilon \in \text{Im}(M_A^T), \quad (3)$$

$$A \notin \Gamma \Rightarrow \exists \mathbf{v} \in \ker(M_A) \subset R^d : \langle \varepsilon, \mathbf{v} \rangle \in R^*. \quad (4)$$

Note that, (4) is a stronger assumption than the corresponding requirement for an MSP in the ESP case, namely (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 $\varepsilon \in \text{Im}(M_A^T)$.

For the rest of this paper 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 $\varphi : [m] \rightarrow \mathcal{P}$. 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.

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 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 (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 (2) of the MSP, or requirement (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 2.3 implies that we can solve linear equations in the quotient rings \mathbb{Z}_{p^k} .

Lemma 2.2. *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 2.3. *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{J} for all finitely generated ideals \mathfrak{J} 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 4.

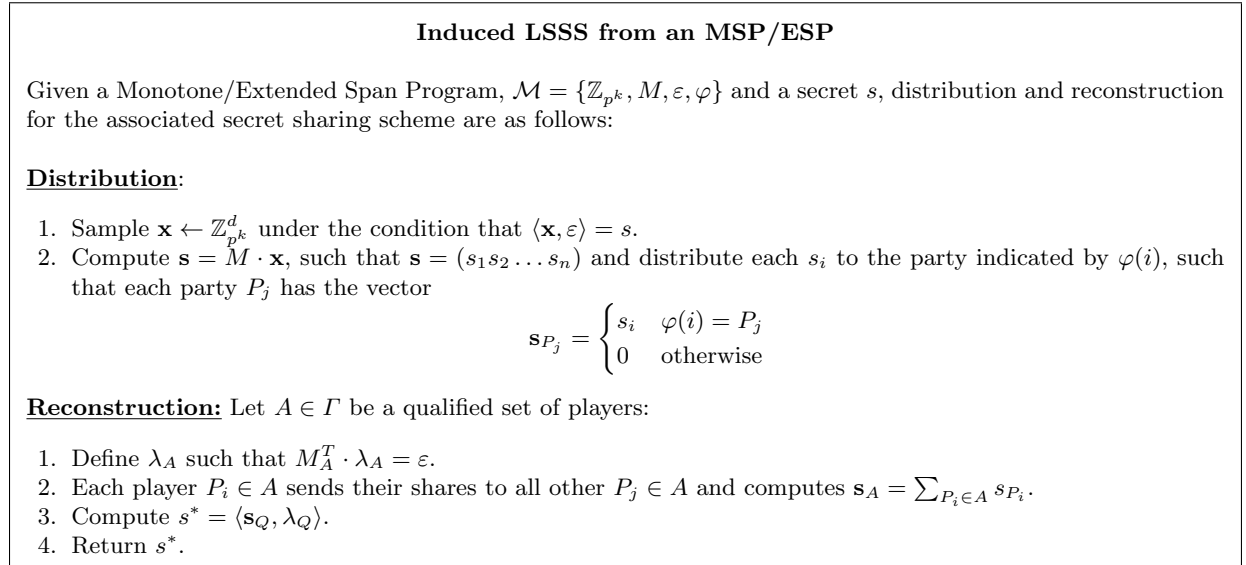


Fig. 1. Induced LSSS from a Monotone/Extended Span Program.

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 $j = 1, \dots, n_i^2$ 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 $i = 1, \dots, n$ and $j = 1, \dots, n_i^2$ such that

$$x \cdot y = \sum_{i,j} \mu_{i,j} \cdot p_{i,j} \tag{5}$$

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 A we prove the following theorem, generalising this result to ESPs over \mathbb{Z}_{p^k} ,

Theorem 2.1. *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.*

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 one⁵ 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 $\{1, \theta, \theta + 1\}$.

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 a 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

⁵ The astute reader will be asking why bother? A simpler implementation in this case come 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.

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 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/ESP).

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.

2.6 Basic Multi-Party Computation Protocols

The general MPC functionality that we aim to implement is given in Figure 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 $\mathcal{H} : \{0, 1\}^* \rightarrow \{0, 1\}^{|\mathcal{H}|}$ 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 3

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

The functionality runs with parties P_1, \dots, 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 $\text{varid} \in I$.

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 $(\text{Input}, P_i, \text{varid}, x)$ from P_i , with $x \in \mathbb{Z}_{p^k}$, and $(\text{input}, P_i, \text{varid}, ?)$ 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 $(\text{Add}, \text{varid}_1, \text{varid}_2, \text{varid}_3)$ from all parties:

1. If $\text{varid}_1, \text{varid}_2$ 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 $(\text{varid}_3, x + y)$.

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

1. If $\text{varid}_1, \text{varid}_2$ 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 $(\text{varid}_3, x \cdot y)$.

Output: On input $(\text{Output}, \text{varid}, i)$ 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.

Fig. 2. The ideal functionality for MPC with Abort over \mathbb{Z}_{p^k}

Interface for a Cryptographic Hash Function \mathcal{H}

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

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

Fig. 3. Interface for a Cryptographic Hash Function \mathcal{H}

Commitment and Decommitment We can implement the ideal functionality $\mathcal{F}_{\text{Commit}}$ given in Figure 4 using the protocol given in Figure 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 $r = r_1 \oplus \dots \oplus r_n$, 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 6

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 $[x]_k \leftarrow \text{Share}(x, i, k)$.

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

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

Commit: On input $(\text{Commit}, v, i, \tau_v)$ 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 (Open, i, τ_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 $(\text{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs (\perp, i, τ_v) to all players.

Fig. 4. The Ideal Functionality for Commitments

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.

Fig. 5. The Protocol for Commitments.

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 is 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 $[r_i]_k \leftarrow \text{Share}(r_i, i, k)$. The resulting shared value being $\sum [r_i]_k$. We denote this functionality by $\mathcal{F}_{\text{PRSS}}$, which is given in Figure 7

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 `OpenToAll` (see later for how we define this protocol, which implements the `Output` operation in the functionality in Figure 2). The protocol is passively secure if the underlying triple is only passively secure, and actively secure otherwise⁶. We refer to this protocol as $[z]_k \leftarrow \text{BeaverMult}([x]_k, [y]_k)$.

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 $[z]_k \leftarrow \text{MaurerMult}([x]_k, [y]_k)$.

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

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

On input `AgreeRandom(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.

Fig. 6. Ideal Functionality $\mathcal{F}_{\text{AgreeRandom}}(D)$

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

On input $\text{PRSS}(\text{cnt})$ 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.

Fig. 7. Ideal Functionality $\mathcal{F}_{\text{PRSS}}(m)$

The third technique is the method of [SW19, KRSW18] which is the generalisation 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] generalises 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 $[z]_k \leftarrow \text{KRSWMult}([x]_k, [y]_k)$.

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 $[z]_k \leftarrow \text{DNMult}([x]_k, [y]_k)$.

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 $\langle t \rangle \leftarrow \text{PRZS}$. Each player can then locally compute the share 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 $v \leftarrow \text{OpenToAll}(\langle r \rangle - \langle x \cdot y \rangle)$. 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 $[x \cdot y]_k = [r]_k - v$.

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 $[z]_k \leftarrow \text{PassMult}([x]_k, [y]_k)$.

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 3.1. *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 $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ 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

$$\begin{aligned} Q \in \Gamma &\Leftrightarrow Q \in \Gamma_p, \\ Q \notin \Gamma &\Leftrightarrow Q \notin \Gamma_p. \end{aligned}$$

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

$Q \in \Gamma \Rightarrow Q \in \Gamma_p$: Let $Q \in \Gamma$, then by the first condition on the ESP it holds that $\varepsilon \in \text{Im}(M_Q^T)$ 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 :

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

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

$Q \notin \Gamma \Rightarrow Q \notin \Gamma_p$: 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 $\mathbf{a}_p \in \ker\left(\left((M_p)_Q^T\right)\right)$. By the fundamental theorem of linear algebra

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

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

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 $\varphi : [m] \rightarrow \mathcal{P}$. 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) 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 2.1.

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 $j = 1, \dots, n_i^2$ 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 $[0, \dots, p)$, 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 $a_{i,j,v,v'} \in [0, \dots, p^2)$. 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 $b_{v,v'} \in [0, \dots, p^2)$.

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

$$A \cdot \underline{\mu} = \mathbf{b} \pmod{p^k} \tag{6}$$

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 $[0, \dots, p^2)$, 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 $U \in \text{GL}_s(\mathbb{Z})$ and $V \in \text{GL}_t(\mathbb{Z})$, i.e. the matrices of determinant ± 1 , such that

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

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 (6) has a solution modulo p^k .

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

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

where $\text{ord}_p(x)$ 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 .

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 realisation of the **Output** command in Figure 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 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 $x \leftarrow \text{OpenToOne}(i, [x]_k)$ in Figure 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 $x \leftarrow \text{OpenToAll}([x]_k)$, one could execute $\text{OpenToOne}(i, [x]_k)$ 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 the 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 $\mathcal{H}.\text{Update}(\mathbf{x})$.

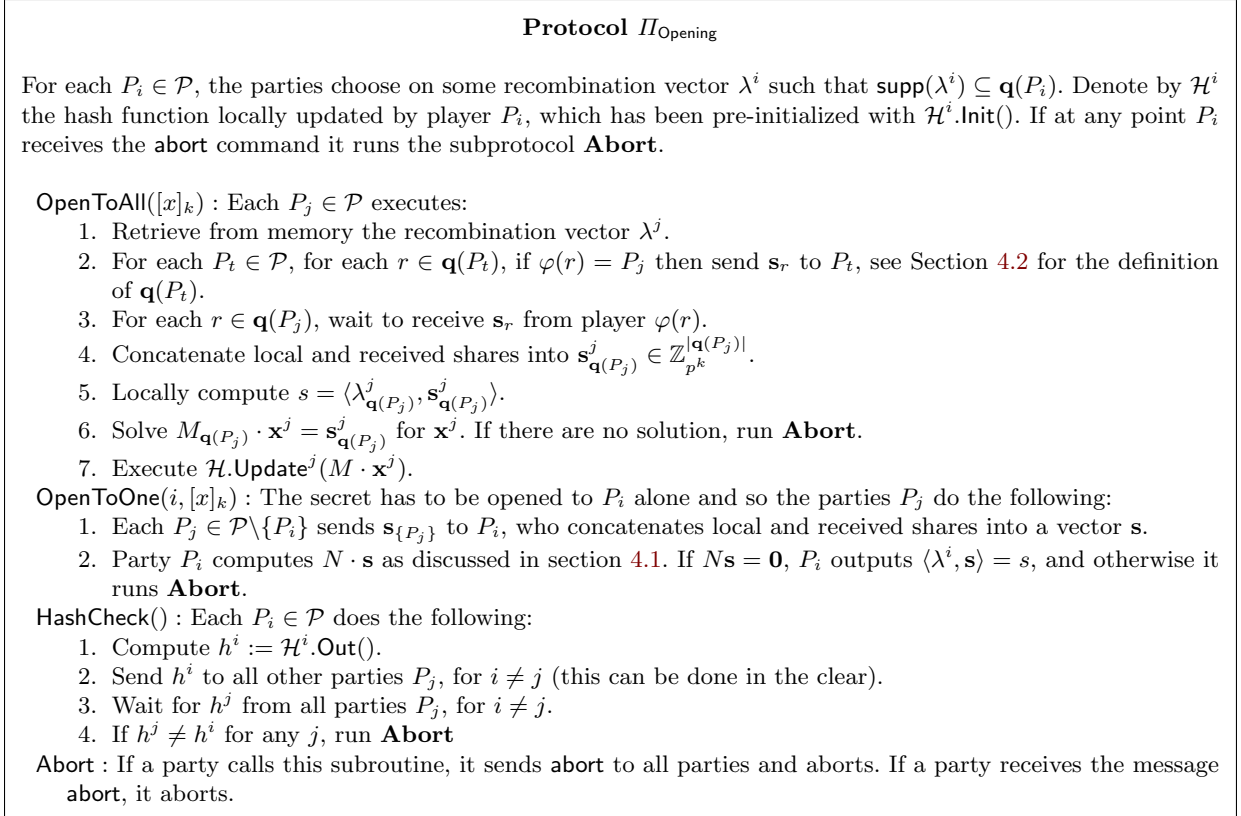


Fig. 8. Protocol Π_{Opening}

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 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, $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$, 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 4.1. *Let $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ 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*

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

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 4.2. *Let $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ 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 $\dim(\text{Im}(M^T)) = d = \text{rank}(M)$, which means M^T is surjective. Hence, by the first isomorphism theorem:

$$\begin{aligned} \mathbb{Z}_{p^k}^m / \ker(M^T) &\cong \text{Im}(M^T) \\ \Rightarrow \ker(M^T) &\cong \mathbb{Z}_{p^k}^m / \mathbb{Z}_{p^k}^d \end{aligned}$$

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

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

Proof. Consider the situation where $\varphi(\text{supp}(\mathbf{s})) \notin \Gamma$. Then, by \mathcal{Q}_2 -ness of the access structure, $\mathcal{P} \setminus \varphi(\text{supp}(\mathbf{s})) \in \Gamma$ 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 2.2 and 2.3 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. $\varphi(\text{supp}(\lambda)) \subseteq Q$, 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 $\mathbf{s} \in \text{Im}(M)$, then for all $Q \in \Gamma$ it holds that $\langle \lambda_Q, \mathbf{s}_Q \rangle \equiv 0 \pmod{p^k}$, and so by definition of the ESP $\mathbf{s} = M \cdot \mathbf{x}$ and $\langle \mathbf{x}, \varepsilon \rangle = 0$. Else $\mathbf{s} \notin \text{Im}(M)$, which proves the proposition. \square

We can now prove Lemma 4.1,

Proof. Let N be a matrix whose rows form a basis for $\ker(M^T)$, which exists by Lemma 4.3 and suppose $\mathbf{e} \in \mathbb{Z}_{p^k}^d$. Since by the Fundamental Lemma of Linear Algebra we have $\ker(M^T) = \text{Im}(M)^\perp$, we also have $\mathbf{s} \in \text{Im}(M)$ if and only if $N \cdot \mathbf{s} = 0$. By the predicate and Lemma 4.3 we have that either $\mathbf{e} \notin \text{Im}(M)$ or $\mathbf{e} \in \text{Im}(M)$ and $e = 0$. If the latter holds we are done. If $\mathbf{e} \notin \text{Im}(M)$, then $N \cdot \mathbf{e} \neq 0$, and so $N \cdot \mathbf{c} = N \cdot (\mathbf{s} + \mathbf{e}) \neq 0$ as required. \square

4.2 Open to All

The `OpenToAll` procedure of Figure 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 $\mathbf{q} : \mathcal{P} \rightarrow 2^{[m]}$, 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 $\mathbf{s}_{\mathbf{q}(P_i)}^i$ for a given secret s . Then, using this vector, P_i solves $\mathbf{s}_{\mathbf{q}(P_i)}^i = M_{S_i} \cdot \mathbf{x}_{P_i}^i$ for $\mathbf{x}_{P_i}^i$. This $\mathbf{x}_{P_i}^i$ 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 4.4. *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 $M \equiv M_p \pmod{p}$ 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 $M \equiv M_p \pmod{p}$ this means that if $\ker(M_{S_i}) \neq \mathbf{0}$ then

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

Hence assume that this is the case and denote by \mathbf{m}_i the i th 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 $\mathbf{0} \pmod{p^k}$, 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 $v_i \in \{0, p, \dots, p^{k-1}\}$. 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} . \square

Now consider the equation $\mathbf{s}^i = M \cdot \mathbf{v}x^i$ 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 $\mathbf{s}_{\mathbf{q}(P_i)}^i$ is a subvector of some share vector \mathbf{s} . If this is the case the protocol in Figure 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 4.5. *Let $\mathbf{q} : \mathcal{P} \rightarrow 2^{[m]}$ be defined with the conditions given above (for a \mathcal{Q}_2 ESP) and let $\mathbf{s}_{\mathbf{q}(P_i)}^i$ 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 $\mathbf{s}_{\mathbf{q}(P_i)}^i = M_{\mathbf{q}(P_i)} \cdot \mathbf{x}^i$, 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 [SW19] shows that \mathbf{q} exists modulo p and Lemma 4.4 has shown that the same \mathbf{q} can be used. As $\ker(M_{\mathbf{q}(P_i)}) = \mathbf{0}$, we see 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

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

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

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 $B_z = 1$.

$$\begin{aligned} u &= \lceil (\kappa + B_z) / \log_2 p \rceil \\ v &= u - 1, \\ 1 \leq B &\leq 1 + (p^w - 1) / 2^{\kappa + B_z}. \end{aligned}$$

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 2.1 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 generalisation 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.

5.1 MultCheck₁

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 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.

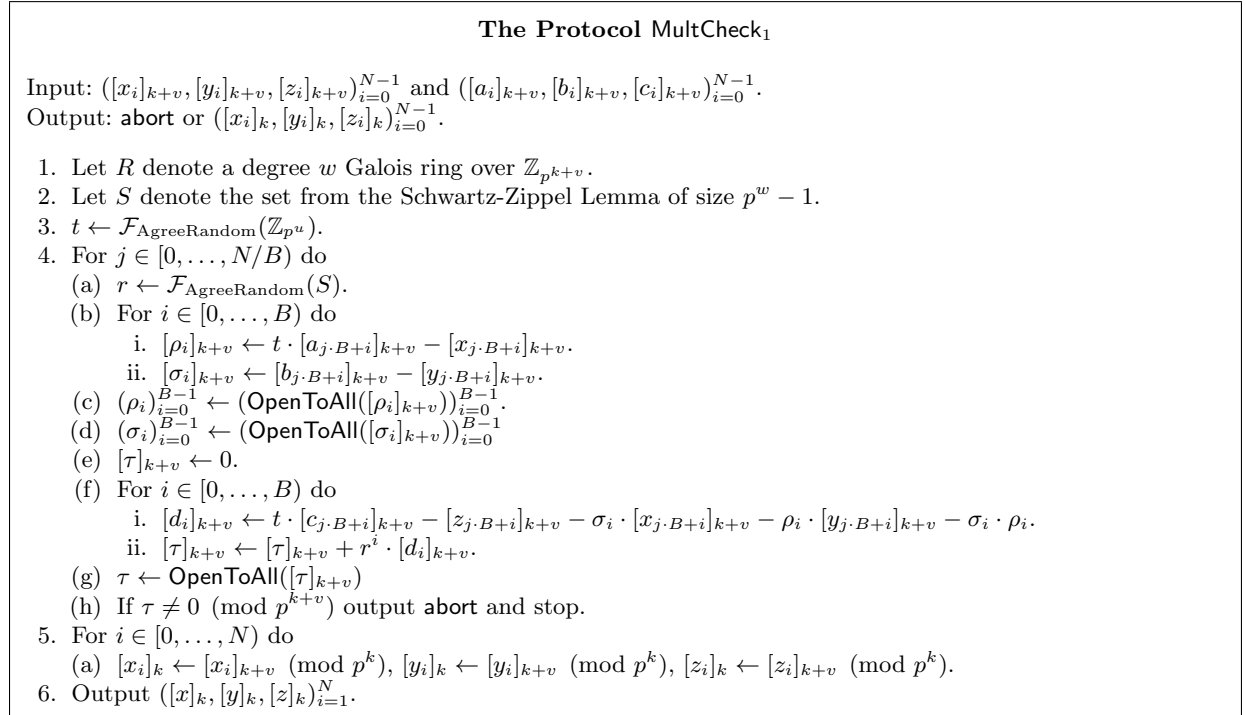


Fig. 9. The Protocol MultCheck₁

The number of calls to the procedure $\text{OpenToAll}(\cdot)$, 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 $\text{OpenToAll}(\cdot)$. 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 $\text{OpenToAll}(\cdot)$.

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 $\text{OpenToAll}(\cdot)$, but for our protocol we can achieve two calls to $\text{OpenToAll}(\cdot)$.

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 `MultCheck1` 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 5.1. *In the presence of an active adversary, who can introduce arbitrary additive errors into the input triples, the protocol `MultCheck1` 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 2.1 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 $e_{c,i} \not\equiv 0 \pmod{p^k}$, 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 $e_{c,i} \not\equiv 0 \pmod{p^k}$ 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 $p^{-u} < 2^{-(\kappa+Bz)}$. \square

5.2 `MultCheck'1`

We will also use the `MultCheck1` 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

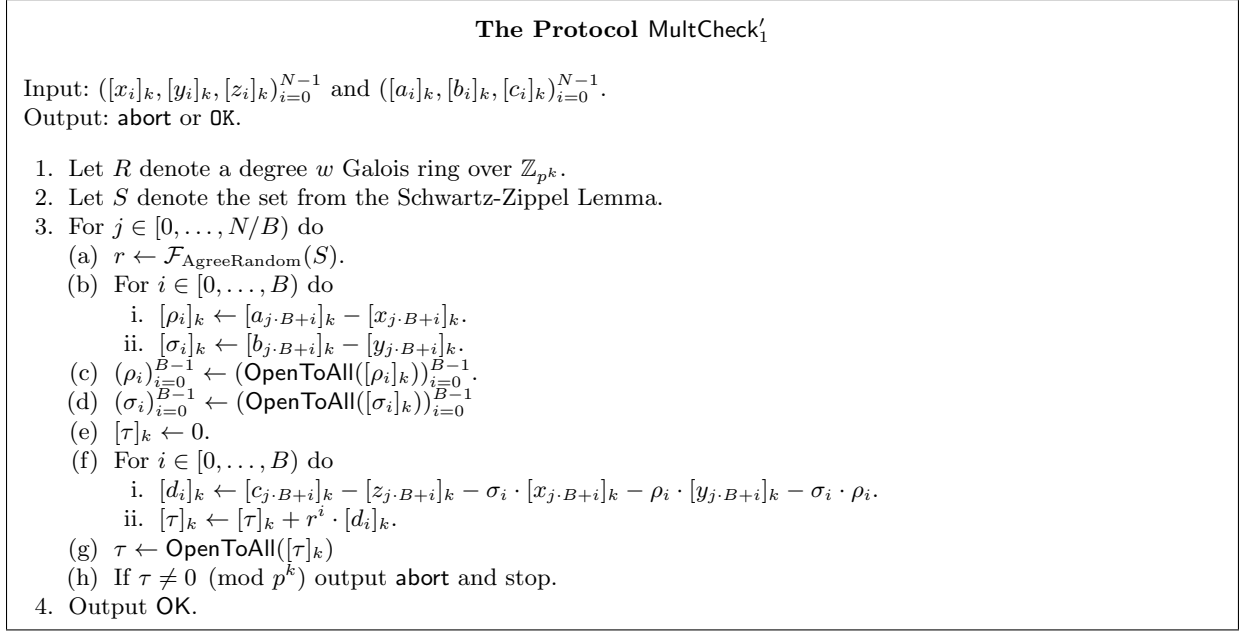


Fig. 10. The Protocol MultCheck'₁

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 MultCheck'₁ and it is presented in Figure 10. The round complexity is the same as that of MultCheck₁, 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,

Lemma 5.2. *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 MultCheck'₁ will output OK incorrectly with probability $(B-1)/(p^w-1) \leq 2^{-(\kappa+B_z)}$.*

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 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 $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$. \square

5.3 MultCheck₂

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 MultCheck₂, is described in Figure 11. The argument for security is roughly the same as that for protocol MultCheck₁.

The Protocol MultCheck₂

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) $[c_i]_{k+v} \leftarrow \text{PassMult}([a_i]_{k+v}, [y_i]_{k+v})$.
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) $([\rho_i]_{i=0}^{B-1} \leftarrow (\text{OpenToAll}([\rho_i]_{k+v}))_{i=0}^{B-1}$.
 - (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) $\tau \leftarrow \text{OpenToAll}([\tau]_{k+s})$.
 - (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 $([x_i]_k, [y_i]_k, [z_i]_k)_{i=0}^{N-1}$.

Fig. 11. The Protocol MultCheck₂

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 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 12

Lemma 5.3. *Protocol MacCheck in Figure 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 $e_i \neq 0 \pmod{p^k}$.*

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

$$\begin{aligned}
 t = v - \alpha \cdot u &= \sum_{i=0}^{N-1} r_i \cdot y_i - r_i \cdot \alpha \cdot x_i \\
 &= \sum_{i=0}^{N-1} r_i \cdot (y_i - \alpha \cdot x_i) \\
 &= \sum_{i=0}^{N-1} r_i \cdot e_i = e.
 \end{aligned}$$

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. $\alpha \leftarrow \text{OpenToAll}([\alpha]_{k+v})$.
6. $[t]_{k+v} \leftarrow [v]_{k+v} - \alpha \cdot [u]_{k+v}$.
7. $[s]_{k+v} \leftarrow \text{PassMult}([t]_{k+v}, [c]_{k+v})$.
8. $s \leftarrow \text{OpenToAll}([s]_{k+v})$.
9. If $s = 0$ then return **OK**, else return **abort**.

Fig. 12. The Protocol MacCheck

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 Summary

We summarize the costs of various protocols in Table 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 $|s_i|$ denote the share size of player P_i for the given ESP. The data column indicates the total amount of data sent for *all* players⁷ 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 **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 **MultCheck₂** and **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 **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.

⁷ i.e. not the per-player amount

Protocol	General MSP			
	Rounds	Data	PRSS/PRZS Triples	
Share	1	$m - \mathbf{s}_i $	0	0
OpenToOne	1	$m - \mathbf{s}_i $	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 2. Costs of the Base Protocols for a General Access Structures

To provide more concrete values we also give, in Table 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 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 **OpenToAll** per player is $d - |\mathbf{s}_i|$, and thus the total cost, \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 3⁸. The derivation of these costs can be found in Appendix 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 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

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

will assume the most efficient choice of passive multiplication is used for a given ESP for the rest of our analysis in this paper. An interesting observation is that the KRWSMult 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.

Protocol	Replicated (3, 1)				Replicated (5, 2)				Replicated (10, 4)			
	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
KRWSMult	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

Protocol	Shamir (3, 1) for large prime				Shamir (5, 2) for large prime				Shamir (10, 4) for large prime			
	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
KRWSMult	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

Protocol	Shamir (3, 1) for \mathbb{Z}_{2^k}				Shamir (5, 2) for \mathbb{Z}_{2^k}				Shamir (10, 4) for \mathbb{Z}_{2^k}			
	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
KRWSMult	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 3. Costs of the Base protocols for Various Access Structures

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 $\text{Offline}_X(p^{\text{output}}, p^{\text{internal}})$ for an offline protocol which outputs triples modulo p^{output} , whilst working internally modulo p^{internal} . Note, if $\text{output} = k + v$ then we must have $\text{internal} = k + v$ 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 whichever is the best out of KRSW or DN for the specific parameter sets⁹.

Offline_{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 $N \cdot \text{PassMult}_{\text{data}}$, and a corresponding cost of $\text{PassMult}_{\text{data}}$ per triple produced. We simplify the presentation in Table 4 by writing $\text{Offline}_{\text{Pass}}(p^k)$ for $\text{Offline}_{\text{Pass}}(p^k, p^k)$ and $\text{Offline}_{\text{Pass}}(p^{k+v})$ for $\text{Offline}_{\text{Pass}}(p^{k+v}, p^{k+v})$.

Offline₁: The first actively secure protocol, Offline_1 , will follow the ideas presented in [DPSZ12], in that to generate N actively secure multiplication triples it starts by executing $\text{Offline}_{\text{Pass}}$ to produce $2 \cdot N$ triples. Then half of the obtained triples are sacrificed, using MultCheck_1 , so as to check the remaining half for correctness.

The cost of $\text{Offline}_{\text{Pass}}$ is given above. For the verification stage we apply an applications of MultCheck_1 on two vectors of triples, each of length N . This requires two rounds of communication and $(2 + w/B) \cdot \text{OpenToAll}_{\text{data}}$ 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

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

However, note that MultCheck_1 requires 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

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

irrespective of whether $\text{output} = k$ or $\text{output} = k + v$ or not. Thus the cost of $\text{Offline}_1(p^k, p^{k+v})$ and $\text{Offline}_1(p^{k+v}, p^{k+v})$ are identical. Thus, in our table below (Table 4) we simply write $\text{Offline}_1(p^{k+v})$.

Offline₂: For the second active offline protocol, Offline_2 , we follow [EKO⁺20]. First N passively secure triples are generated using $\text{Offline}_{\text{Pass}}$. Then these triples are checked to be resistant to additive attacks by running MultCheck_2 on the vector of N triples.

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

The communication in MultCheck_2 requires three rounds of interaction and the data cost is $\text{PassMult}_{\text{data}} + (1 + w/B) \cdot \text{OpenToAll}_{\text{data}}$. This renders the cost for the full protocol, amortizing for the number of multiplications, to be four rounds of communication and

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

in data transferred. Much like 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

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

Again, the cost of $\text{Offline}_2(p^k, p^{k+v})$ and $\text{Offline}_2(p^{k+v}, p^{k+v})$ are identical. Again, in our table below (Table 4) we simply write $\text{Offline}_2(p^{k+v})$.

Offline₃: For our third variant of the Offline protocol, which we call 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 $T = (N + C \cdot L) \cdot (\text{Bk} - 1) + N$ 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 $\text{Offline}_{\text{pass}}$ and to perform the check, the T triples are divided into Bk sets. The first D_1 of size N , whilst the rest $D_2, \dots, D_{\text{Bk}}$, of size $N + L \cdot C$. The sets $D_2, \dots, D_{\text{Bk}}$ are further subdivided into sets of size X , $D_{i,j}$ for $i = 2, \dots, \text{Bk}$ and $j = 1, \dots, L$. The elements of set $D_{i,j}$ are then randomly permuted within each other, and then a random permutation is applied to the vector $(1, \dots, L)$, so as to randomly permute the sets $D_2, \dots, D_{\text{Bk}}$. The permutation is done in this way to ensure cache efficiency. Finally the first C triples in each subarray $D_{i,j}$, for $i = 2, \dots, \text{Bk}$ and $j = 1, \dots, L$ are opened and verified to be correct. Thus this set requires $3 \cdot C \cdot (\text{Bk} - 1) \cdot L$ parallel calls to OpenToAll , resulting in one round of communication and data transfer of

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

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

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

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

The last consideration to be had regarding the cost of this protocol is that using $\text{MultCheck}'_1$ allows Offline_3 to work, not in $\mathbb{Z}_{p^{k+v}}$, but in $\mathbb{Z}_{p^{\text{output}}}$. Thus if we have $\text{output} = k$ 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 $1/N^{\text{Bk}-1}$ when used as a standalone offline procedure, or $1/N^{\text{Bk}}$ 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 $N = 2^{20}$, $\text{Bk} = 2$, $L = 512$ 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 paper we target a statistical security level of $\kappa = 128$. Thus when using Offline_3 in Protocol 1 below we use the parameters $(N, \text{Bk}, L, C) = (2^{22}, 7, 512, 1)$ and when using Offline_3 in Protocol 4 below we use the parameters $(N, \text{Bk}, L, C) = (2^{22}, 6, 512, 3)$. To simplify the presentation in Table 4 by writing $\text{Offline}_3(p^k)$ for $\text{Offline}_3(p^k, p^k)$ and $\text{Offline}_3(p^{k+v})$ for $\text{Offline}_3(p^{k+v}, p^{k+v})$, and we give the costs for the parameters $(N, \text{Bk}, L, C) = (2^{22}, 6, 512, 3)$ in the table.

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 etc for Offline_3 . Comparing Offline_1 vs Offline_2 , is simpler as Offline_1 is better in terms of number of rounds of communication, whereas 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 4. We assume a security parameter of $\kappa = 128$, and choices of (u, v, w, B) from Section 5 so that w/B can be ignored. In Table 4 we present the number of bits per triples that needs to be transferred.

Access				$\text{Offline}_{\text{Pass}}(p^k)$	$\text{Offline}_{\text{Pass}}(p^{k+v})$	$\text{Offline}_1(p^{k+v})$	$\text{Offline}_2(p^{k+v})$	$\text{Offline}_3(p^k)$	$\text{Offline}_3(p^{k+v})$
Structure	Ring	Scheme	Mult						
(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 4. Costs of the Offline Protocols in number of bits per multiplication, for various access structures; $\kappa = 128$, $p \approx 2^{128}$

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 Offline_x , for x either 1, 2 or 3, given in Section 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.

Protocol₁: This protocol executes an actively secure offline phase to produce N triples in \mathbb{Z}_{p^k} , i.e. we execute $\text{Offline}_x(p^k, p^*)$ for \star being either k or $k+v$, depending on the precise protocol choice x . Note, this means we have three choices for Protocol_1 depending on which offline protocol the main protocol is combined with. The online phase is executed, using these triples, using 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 BeaverMult requires one round of communication and a total of $2 \cdot \text{OpenToAll}_{\text{data}}$ in data transferred. Thus our we have the online phase requires d rounds of communication and

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

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 $\text{Offline}_{\text{rounds}} + d$ rounds of communication and a data communication cost of

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

per multiplication, where $\text{Offline}_{\text{data}}$ is taken from the relevant columns of Table 4. In Table 5 we refer to the three combined costs per multiplication as Total_x , depending on which Offline phase we are utilizing.

Protocol₂: In this protocol we optimistically use a passively secure online multiplication protocol 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 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 MultCheck_1 , and the total communication for the online phase, per multiplication gate, is

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

where $\text{PassMult}_{\text{data}}$ and $\text{MultCheck}_{1,\text{data}}$ are taken from Table 2.

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

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

per multiplication, where $\text{Offline}_{\text{Pass}}(p^{k+v})_{\text{data}}$ is taken from Table 4.

Protocol₃: This proceeds very much as **Protocol₂** except instead of using an offline phase and the MultCheck_1 procedure, one uses the 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 as we require five rounds to execute MultCheck_2 , and the total communication for the online phase, per multiplication gate, is

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

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

Protocol₄: 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 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 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 $\text{MultCheck}'_1$. 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 $\text{MultCheck}'_1$, and the total communication for the online phase, per multiplication gate, is

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

where $\text{PassMult}_{\text{data}}$ and $\text{MultCheck}_{1,\text{data}}$ are taken from Table 2.

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

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

per multiplication, where $\text{Offline}_{\text{data}}$ is taken from Table 4. Again in Table 5 we will refer to the three different combined costs per multiplication as Total_x .

Protocol₅: 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 MacCheck procedure. To ensure the security of the 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

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

where $\text{PassMult}_{\text{data}}$ and $\text{MacCheck}_{\text{data}}$ are taken from Table 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) 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 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 Protocol₁. We also mark in gray the most efficient protocol option when one is interested in the total cost. For small rings this is always Protocol₄ with Offline₃ chosen as the pre-processing, for the others it is Protocol₅.

Access Structure	Ring	Scheme	Mult	Protocol ₁			Protocol ₂		Protocol ₃		Protocol ₄			Protocol ₅			
				Online	Total ₁	Total ₂	Total ₃	Online	Total	Online	Total	Online	Total ₁	Total ₂	Total ₃	Online	Total
(3, 1)	\mathbb{F}_2	Replicated	KRSW	6	1554	1167	63	1161	1548	1161	1161	9	1557	1170	57	774	774
(3, 1)	\mathbb{F}_2	Shamir \mathbb{Z}_{2^k}	DN	6	2328	1941	84	1548	2322	1935	1935	12	2334	1947	78	1548	1548
(3, 1)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	768	3840	3072	8067	2304	3072	2304	2304	1152	4224	3456	7299	1536	1536
(3, 1)	$\mathbb{Z}_{2^{128}}$	Shamir \mathbb{Z}_{2^k}	DN	768	5376	4608	10756	3072	4608	3840	3840	1536	6144	5376	9988	3072	3072
(3, 1)	\mathbb{F}_p	Replicated	KRSW	768	3840	3072	8067	2304	3072	2304	2304	1152	4224	3456	7299	1536	1536
(3, 1)	\mathbb{F}_p	Shamir	KRSW	768	3840	3072	8067	2304	3072	2304	2304	1152	4224	3456	7299	1536	1536
(5, 2)	\mathbb{F}_2	Replicated	KRSW	40	7780	5200	350	6450	7740	5160	5160	50	7790	5210	310	2580	2580
(5, 2)	\mathbb{F}_2	Shamir \mathbb{Z}_{2^k}	DN	40	10360	7780	420	7740	10320	7740	7740	60	10380	7800	380	5160	5160
(5, 2)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	5120	20480	15360	44820	12800	15360	10240	10240	6400	21760	16640	39696	5120	5120
(5, 2)	$\mathbb{Z}_{2^{128}}$	Shamir \mathbb{Z}_{2^k}	DN	5120	25600	20480	53782	15360	20480	15360	15360	7680	28160	23040	48659	10240	10240
(5, 2)	\mathbb{F}_p	Replicated	KRSW	5120	20480	15360	44820	12800	15360	10240	10240	6400	21760	16640	39696	5120	5120
(5, 2)	\mathbb{F}_p	Shamir	KRSW	2560	12800	10240	26891	7680	10240	7680	7680	3840	14080	11520	24329	5120	5120
(10, 4)	\mathbb{F}_2	Replicated	KRSW	1680	231300	122940	12116	223170	229620	121260	121260	1730	231350	122990	10435	12900	12900
(10, 4)	\mathbb{F}_2	Shamir \mathbb{Z}_{2^k}	DN	260	57020	40250	2451	45150	56760	39990	39990	350	57110	40340	2191	23220	23220
(10, 4)	$\mathbb{Z}_{2^{128}}$	Replicated	KRSW	215040	670720	455680	1550803	442880	455680	240640	240640	221440	677120	462080	1335642	25600	25600
(10, 4)	$\mathbb{Z}_{2^{128}}$	Shamir \mathbb{Z}_{2^k}	DN	33280	145920	112640	313735	89600	112640	79360	79360	44800	157440	124160	280432	46080	46080
(10, 4)	\mathbb{F}_p	Replicated	KRSW	215040	670720	455680	1550803	442880	455680	240640	240640	221440	677120	462080	1335642	25600	25600
(10, 4)	\mathbb{F}_p	Shamir	KRSW	10240	56320	46080	116528	33280	46080	35840	35840	16640	62720	52480	106280	25600	25600

Table 5. Costs of the Full Protocols in number of bits per multiplication, for various access structures; $\kappa = 128$, $p \approx 2^{128}$

Note, that in the case of Protocol₄ and Offline₃ 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 Protocol₁ 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 executions to open masked data for use in various function specific optimizations. Using Protocol₁ 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 conversions 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.

References

- ABF⁺17. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.
- 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: 17th Theory of Cryptography Conference, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.
- ACD⁺20. Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, Matthieu Rabaud, 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, *Advances in Cryptology – ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 151–180, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.
- 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. <https://eprint.iacr.org/2019/1298>.
- BLW08. Dan Bogdanov, Sven Laur, and Jan Willemsen. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008: 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.
- 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, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

- CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.
- CDM00. Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.
- CGH⁺18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.
- 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. <https://eprint.iacr.org/2019/832>.
- DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.
- DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.
- 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: 1st Conference on Information-Theoretic Cryptography*, pages 5:1–5:24, Boston, MA, USA, June 17–19, 2020. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
- 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 20: 27th Conference on Computer and Communications Security*, pages 1575–1590, Virtual Event, USA, November 9–13, 2020. ACM Press.
- KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.
- KRSW18. Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Reducing communication channels in MPC. In Dario Catalano and Roberto De Prisco, editors, *SCN 18: 11th International Conference on Security in Communication Networks*, volume 11035 of *Lecture Notes in Computer Science*, pages 181–199, Amalfi, Italy, September 5–7, 2018. Springer, Heidelberg, Germany.
- 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, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 210–229, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

A Proof of Theorem 2.1

In this Appendix we discuss the proof of Theorem 2.1. To do this assume that $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ 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 $\mathcal{M}_0 = (\mathbb{Z}_{p^k}, M_0, \mathbf{e}_1, \varphi)$

and $\mathcal{M}_1 = (\mathbb{Z}_{p^k}, M_1, \mathbf{e}_1, \varphi)$ and let Γ_0 and Γ_1 be the access structures computed by \mathcal{M}_0 and \mathcal{M}_1 respectively.

Lemma A.1. *Suppose that $M_0^T \cdot M_1 = [\mathbf{e}_1, \mathbf{0}, \dots, \mathbf{0}]$, 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 LSSS₀ and LSSS₁ 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 i th 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

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

Let M_i^j be the j th 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. \square

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 A.2. *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 2.1.

Proof. Of Theorem 2.1: Let Γ be an access structure, and define a boolean function $\gamma : \mathcal{P} \rightarrow \{0, 1\}$, 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 $\overline{\cdot}$ indicates a flip, that is $\gamma(A) = 0$ then $\overline{\gamma(A)} = 1$ and $\gamma(\overline{A}) = \gamma(\mathcal{P} \setminus A)$. By our assumption the access structure is complete so if a subset A is qualified, then $\overline{A} = \mathcal{P} \setminus A$ is unqualified, therefore $\gamma(A) = 0 \Leftrightarrow \gamma^*(A) = 0$, and therefore, tautologically, $\Gamma = \Gamma \vee \Gamma^*$.

Let $\mathcal{M} = (\mathbb{Z}_{p^k}, M, \varepsilon, \varphi)$ 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 A.2 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}^* be it's dual given by [Gá95], then

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

is the dual MSP computing f^* 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

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] 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 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

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

The key part of the multiplication algorithm, and the only part which requires interaction, is the mechanism Π_{Convert} , (in Figure 15 for [KRSW18] and Figure 17 for [SW19]), to transfer an additive secret sharing $\langle x \rangle$ amongst all n parties, i.e. $x = x_1 + \dots + x_n$ 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 .

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^*} .

Fig. 13. The Functionality $\mathcal{F}_{\text{PRZS}}$

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

On input $\text{AgreeRandom}(\text{cnt})$ 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 $S \subset \mathcal{P}$.

Fig. 14. Ideal Functionality $\mathcal{F}_{\text{AgreeRandom}'}(D, S)$

It also uses a map $\chi : [n] \rightarrow [d]$, 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 $\psi : [d] \rightarrow [n]$ 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 $k \notin \text{im}(\chi)$ 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 15.

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

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 $\chi : [n] \rightarrow [d]$ and $\psi : [d] \rightarrow [n]$ described in the text.

1. For $k \in [1, \dots, d]$ 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 $(\text{cnt}, \mathcal{P})$ to obtain a PRZS, denoted hereafter by $\langle t \rangle$.
3. For $k \notin \text{im}(\chi)$, define \mathbf{s}_j for $j \in \mathcal{J}_k$ by calling $\mathcal{F}_{\text{AgreeRandom}'}(R, \mathcal{I}_k)$.
4. For $k \in \text{im}(\chi)$, 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 $j \in \mathcal{J}_k$.

Fig. 15. KRSW Protocol Π_{Convert} converting additive shares to shares in the LSSS

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 $\chi : [n] \rightarrow [d]$ 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 16.

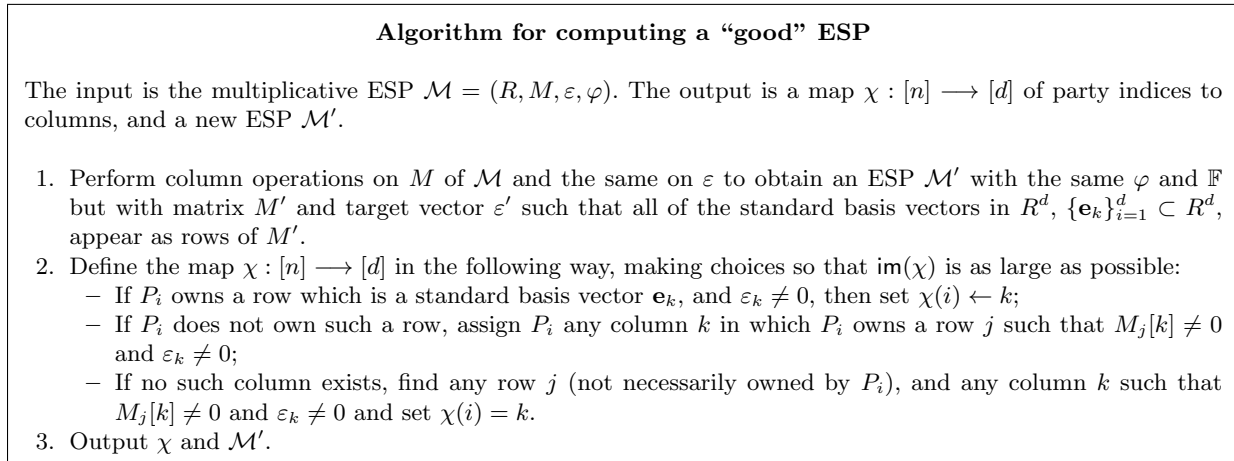


Fig. 16. Algorithm for computing a “good” ESP

We now illustrate the methods with our examples:

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

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 16)

1. The parties call $\mathcal{F}_{\text{PRZS}}$ with the command $(\text{cnt}, \mathcal{P})$ to obtain a PRZS, denoted hereafter by $\langle t^0 \rangle$.
2. Each P_i splits $x_i + t_i^0$ as $x_i + t_i^0 = \sum_{k \in K_i \cap \text{supp}(\varepsilon')} x_{i,k}$ where $K_i \leftarrow (\{\chi(P_i)\} \cup ([d] \setminus \text{im}(\chi)))$.
3. Each P_i sets $r_{i,k} \leftarrow x_{i,k}/\varepsilon'_k$ for each $k \in K_i \cap \text{supp}(\varepsilon')$.
4. Each P_i sets $r_{i,k} \leftarrow R$ for each $k \in K_i \setminus \text{supp}(\varepsilon')$.
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 $(\text{cnt}, \mathcal{P})$ to obtain a PRZS amongst them, denoted hereafter by $\langle t^j \rangle$.
 - (b) Each P_i computes

$$a_i^j \leftarrow \left(\sum_{k \in K_i \cap \text{supp}(\varepsilon')} M'_j[k] \cdot r_{i,k} \right) + t_i^j,$$

where $M_j[k]'$ denotes the k th 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 $(R, M, \varepsilon, \varphi)$).

Fig. 17. Smart–Wood Protocol Π_{Convert} converting additive shares to shares in the LSSS

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

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

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

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

KRSW Algorithm: We now discuss the methodology for this example of KRSW, i.e. Figure 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 .

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 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 $\text{im}(\chi) = \{1, 2, 3\}$.

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.

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 ,

$$M = \begin{pmatrix} 0010000000 \\ 0000100000 \\ 0000010000 \\ 0000000100 \\ 0000000010 \\ 0000000001 \\ 0100000000 \\ 0001000000 \\ 0000010000 \\ 0000001000 \\ 0000000010 \\ 0000000001 \\ 1000000000 \\ 0001000000 \\ 0000100000 \\ 0000001000 \\ 0000000010 \\ 0000000001 \\ 1000000000 \\ 0100000000 \\ 0010000000 \\ 0000001000 \\ 0000000100 \\ 0000000010 \\ 1000000000 \\ 0100000000 \\ 0010000000 \\ 0001000000 \\ 0000100000 \\ 0000010000 \end{pmatrix},$$

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

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

KRSW Algorithm: We now discuss the methodology of KRSW, in Figure 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 16 and Figure 17. The algorithm in Figure 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 $\text{im}(\chi) = \{1, 2, 3, 4, 7\}$. 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 j th 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 $\mathbf{s}_j = r_{*,j}$ 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 $i = 1, \dots, 5$. The value $r_{i,k} + t_i^k$ is sent by player i to player $\varphi(j)$ if the j th 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.

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 $\mathcal{F}_{\text{AgreeRandom}}(R, \mathcal{I}_k)$.

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 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:

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

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

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

The value shared is equal to

$$2 \cdot \mathbf{s}_1 - \mathbf{s}_2 = x_1 + t_1^0 + x_2 + t_2^0 + x_3 + t_3^0$$

$$= x_1 + x_2 + x_3 = x.$$

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

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

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 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 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

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

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 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 $\text{Im } \chi = [d]$, 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 $\alpha_{\chi(i)}^j \cdot (x_i +$

$t_i^0) + t_i^j$, 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 $\mathcal{F}_{\text{PRZS}}$, 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 :

$$\mathbf{s}_1 = (x_1 + t_1^0)/5$$

$$\mathbf{s}_2 = (-x_2 - t_2^0)/10$$

$$\mathbf{s}_3 = (x_3 + t_3^0)/10$$

$$\mathbf{s}_4 = (-x_4 - t_4^0)/5$$

$$\mathbf{s}_5 = \sum_{5 \leq i \leq 10} x_i + t_i^0 + t_i^5$$

$$\mathbf{s}_6 = (x_1 + t_1^0)/5 + (x_2 + t_2^0)/2 + x_3 + t_3^0 + 2 \cdot (x_4 + t_4^0) + \sum_{5 \leq i \leq 10} 5 \cdot (x_i + t_i^0)$$

$$\mathbf{s}_{j=7,\dots,10} = \alpha_1^j \cdot (x_1 + t_1^0) + \alpha_2^j \cdot (x_2 + t_2^0) + \alpha_3^j \cdot (x_3 + t_3^0) + \alpha_4^j \cdot (x_4 + t_4^0) + \sum_{5 \leq i \leq 10} \alpha_5^j \cdot (x_i + t_i^0)$$

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.

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 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:

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

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

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

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 $\mathcal{M}' = (\mathbb{Z}_{2^k}, M', \epsilon', \varphi)$:

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

$$\epsilon' = (1, -1, 0)$$

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 $r_{i,3} = \text{Rand}(R)$ 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 $a_2^5 = t_2^5$ 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

$$\begin{aligned} \mathbf{s}_1 &= a_1^1 = r_{1,1} = x_1 + t_1^0, \\ \mathbf{s}_2 &= a_1^2 + a_2^2 + a_3^2 = r_{1,3} + r_{2,3} + r_{3,3} = \text{Rand}_1(R) + \text{Rand}_2(R) + \text{Rand}_3(R), \\ \mathbf{s}_3 &= a_1^3 + a_2^3 + a_3^3 = r_{1,1} - r_{2,2} - r_{3,2} = x_1 + x_2 + x_3, \\ \mathbf{s}_4 &= a_1^4 + a_2^4 + a_3^4 = r_{2,2} + r_{3,2} = -x_2 - t_2^0 - x_3 - t_3^0, \\ \mathbf{s}_5 &= a_1^5 + a_2^5 + a_3^5 = r_{1,1} = x_1 + t_1^0 - x_3 - t_3^0, \\ \mathbf{s}_6 &= a_2^6 + a_3^6 = r_{2,2} + r_{3,2} = -x_2 - t_2^0 - x_3 - t_3^0. \end{aligned}$$

Then to check that this is a correct we first check the shared value, which is correct:

$$\mathbf{s}_1 - \mathbf{s}_6 = x_1 + t_1^0 + x_2 + t_2^0 + x_3 + t_3^0 = \sum_i x_i = x$$

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,

$$\mathbf{s}_3 - \mathbf{s}_1 - \mathbf{s}_6 = r_{1,1} - r_{2,2} - r_{3,2} - r_{1,1} + r_{2,2} + r_{3,3} = 0$$

$$\mathbf{s}_4 - \mathbf{s}_6 = 0$$

$$\mathbf{s}_5 - \mathbf{s}_1 = 0$$

B.8 Shamir (5, 2) for \mathbb{Z}_{2^k}

We construct the original matrix M similarly to the example in section 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 $i \in \{1, \dots, 7\}$.
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 $j \in \{8, \dots, 15\}$ 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 $a_3^{15} = r_{3,7} + t_3^{15}$. P_3 then sends $a_3^{10}, a_3^{11},$ and a_3^{12} to $P_4,$ and $a_3^{13}, a_3^{14}, a_3^{15}$ 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 $a_4^{10}, a_4^{11}, a_4^{12}$ 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_i^1 P_1 computes $\mathbf{s}_1 = a_1^1 + a_4^1.$
 - (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 $\mathbf{s}_6 = \sum_i a_i^6.$
 - (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

$$\begin{aligned} \mathbf{s}_1 &= a_1^1 + a_4^1 = r_{1,1} + r_{4,1} = x_{1,1} + t_1^0 + x_{4,1} + t_4^0 \\ \mathbf{s}_2 &= \sum_i a_i^2 = \sum_i r_{i,2} \in R \\ \mathbf{s}_3 &= \sum_i a_i^3 = \sum_i r_{i,3} \in R \end{aligned}$$

$$\begin{aligned}
\mathbf{s}_4 &= a_2^4 + a_5^4 = r_{2,4} + r_{5,4} = x_{2,4} + t_2^0 + x_{5,4} + t_5^0 \\
\mathbf{s}_5 &= \sum_i a_i^5 = \sum_i r_{i,5} \in R \\
\mathbf{s}_6 &= \sum_i a_i^6 = \sum_i r_{i,6} \in R \\
\mathbf{s}_7 &= a_3^7 = r_{3,7} = -x_{3,7} - t_3^0 \\
\mathbf{s}_8 &= \sum_i a_i^8 = 2 \cdot r_{1,1} - 2 \cdot r_{2,4} + 2 \cdot r_{4,1} - 2 \cdot r_{5,4} \\
\mathbf{s}_9 &= \sum_i a_i^9 = -2 \cdot r_{2,4} + 2 \cdot r_{3,7} - 2 \cdot r_{5,4} \\
\mathbf{s}_{10} &= \sum_i a_i^{10} = r_{1,1} + 2 \cdot r_{2,4} - 2 \cdot r_{3,7} + r_{4,1} + 2 \cdot r_{5,4} \\
\mathbf{s}_{11} &= \sum_i a_i^{11} = -r_{1,1} - r_{2,4} + 2 \cdot r_{3,7} - r_{4,1} - r_{5,4} \\
\mathbf{s}_{12} &= \sum_i a_i^{12} = -r_{1,1} + r_{3,7} - r_{4,1} \\
\mathbf{s}_{13} &= \sum_i a_i^{13} = r_{1,1} + r_{2,4} - r_{3,7} + r_{4,1} + r_{5,4} \\
\mathbf{s}_{14} &= \sum_i a_i^{14} = -r_{1,1} - r_{2,4} + 2 \cdot r_{3,7} - r_{4,1} - r_{5,4} \\
\mathbf{s}_{15} &= \sum_i a_i^{15} = -r_{1,1} + r_{3,7} - r_{4,1}
\end{aligned}$$

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)

$$\begin{aligned}
\mathbf{s}_8 - 2 \cdot \mathbf{s}_1 + 2 \cdot \mathbf{s}_4 &= 2 \cdot r_{1,1} - 2 \cdot r_{2,4} + 2 \cdot r_{4,1} - 2 \cdot r_{5,4} - 2 \cdot (r_{1,1} + r_{4,1}) + 2 \cdot (r_{2,4} + r_{5,4}) \\
\mathbf{s}_9 + 2 \cdot \mathbf{s}_4 - 2\mathbf{s}_7 &= -2 \cdot r_{2,4} + 2 \cdot r_{3,7} - 2 \cdot r_{5,4} + 2 \cdot (r_{2,4} + r_{5,4}) - 2 \cdot r_{3,7} \\
\mathbf{s}_{10} - \mathbf{s}_1 - 2 \cdot \mathbf{s}_4 + 2 \cdot \mathbf{s}_7 &= r_{1,1} + 2 \cdot r_{2,4} - 2 \cdot r_{3,7} + r_{4,1} + 2 \cdot r_{5,4} - (r_{1,1} + r_{4,1}) - 2 \cdot (r_{2,4} + r_{5,4}) \\
&\quad + 2 \cdot r_{3,7} \\
\mathbf{s}_{11} + \mathbf{s}_1 + \mathbf{s}_4 - 2\mathbf{s}_7 &= -r_{1,1} - r_{2,4} + 2 \cdot r_{3,7} - r_{4,1} - r_{5,4} + r_{1,1} + r_{4,1} + r_{2,4} + r_{5,4} - 2 \cdot (r_{3,7}) \\
\mathbf{s}_{12} + \mathbf{s}_1 - \mathbf{s}_7 &= -r_{1,1} + r_{3,7} - r_{4,1} + r_{1,1} + r_{4,1} - r_{3,7} \\
\mathbf{s}_{13} - \mathbf{s}_1 - \mathbf{s}_4 + \mathbf{s}_7 &= r_{1,1} + r_{2,4} - r_{3,7} + r_{4,1} + r_{5,4} - r_{1,1} + r_{4,1} - r_{2,4} + r_{5,4} + r_{3,7} \\
\mathbf{s}_{14} + \mathbf{s}_1 + \mathbf{s}_4 - 2 \cdot \mathbf{s}_7 &= -r_{1,1} - r_{2,4} + 2 \cdot r_{3,7} - r_{4,1} - r_{5,4} + r_{1,1} + r_{4,1} + r_{2,4} + r_{5,4} - 2 \cdot (r_{3,7}) \\
\mathbf{s}_{15} + \mathbf{s}_1 - \mathbf{s}_7 &= -r_{1,1} + r_{3,7} - r_{4,1} + r_{1,1} + r_{4,1} - r_{3,7}
\end{aligned}$$

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

$$M = \begin{pmatrix}
 10001000100010001 \\
 00010001000100010 \\
 00100010001000100 \\
 01000100010001000 \\
 11000010000101001 \\
 01001110001101011 \\
 00010100111000110 \\
 00100001010011100 \\
 11001010111111000 \\
 01011111000011001 \\
 00110110100110010 \\
 01100101001110100 \\
 10100100101100101 \\
 01100101110101110 \\
 01001011001011101 \\
 00010110010111010 \\
 10101100010100100 \\
 01110100111111100 \\
 01101001011101001 \\
 01010010011010010 \\
 11100110100011100 \\
 00101011100100101 \\
 01011111101001011 \\
 00110111010000110 \\
 11101110000011101 \\
 00111010100100111 \\
 01111101101001111 \\
 01110011010001110 \\
 10010011010101111 \\
 00110101011110001 \\
 01100010111100011 \\
 01001101111010111 \\
 10011011111111110 \\
 00100100000010011 \\
 01000000100110111 \\
 00001001101111111 \\
 11010001011110110 \\
 01111011000011010 \\
 01110110000110101 \\
 01101100101111011
 \end{pmatrix}$$

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

$$\varphi(i) = \lceil i/4 \rceil$$

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 $N \in M_{23 \times 17}(\mathbb{Z}_{2^k})$. 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 $\text{im}(\chi) = \{1, 2, 3, 4, 5, 6, 7, 9, 13, 17\}$. 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 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 $i \in \{18, \dots, 40\}$, 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 it self 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 $\text{Im}(\chi) = \{1, 3, 4, 7, 8, 11, 12, 13, 17\}$. 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 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 again call $\mathcal{F}_{\text{PRZS}}$ a total of 23 times to generate $\langle t^i \rangle$ for $i \in \{18, \dots, 40\}$, 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 $\mathcal{F}_{\text{PRZS}}$, 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 $\mathcal{F}_{\text{PRZS}}$ functionality and a total transmission of $270 = 207 + 1 + 72 = 280$ ring elements.