Error Detection in Monotone Span Programs with Application to Communication-Efficient Multi-Party Computation

Nigel P. Smart[1,2] and Tim Wood[1,2]

[1] University of Bristol, Bristol, UK.
[2] KU Leuven, Leuven, Belgium.
nigel.smart@kuleuven.be,t.wood@kuleuven.be

Abstract. Recent improvements in the state-of-the-art of MPC for non-full-threshold access structures introduced the idea of using a collision-resistant hash functions and redundancy in the secret-sharing scheme to construct a communication-efficient MPC protocol which is computationally-secure against malicious adversaries, with abort. The prior work is based on replicated secret-sharing; in this work we extend this methodology to *any* multiplicative LSSS implementing a $\mathcal{Q}_2$ access structure. To do so we need to establish a folklore property of error detection for such LSSS and their associated Monotone Span Programs. In doing so we obtain communication-efficient online and offline protocols for MPC in the pre-processing model.

1 Introduction

Secure multi-party computation (MPC) allows a set of parties to compute a function on their combined secret inputs so that all parties learn the output of the function and no party can learn anything that cannot be inferred from the output and their own inputs alone. As a field it has recently received a lot of attention and has been explored in a variety of contexts: for example, private auctions [13], secure statistical analysis of personal information [11] and protection against side-channel attacks in hardware [8, 33, 34].

Most MPC protocols fall into one of two broad categories: garbled circuits, and LSSS-based (linear-secret-sharing-based) MPC. The garbled-circuit approach, which began with the work of Yao [36], involves some collection of parties "garbling" a circuit to conceal the internal circuit evaluations, and then later a single party or a collection of parties jointly evaluating the garbled circuit. By contrast, the LSSS-based approach involves using a so-called linear secret-sharing scheme, in which the parties: "share" a secret into several *shares* which are distributed to different parties, perform computations on the shares, and then reconstruct the secret at the end by combining the shares to determine the output. Secret-sharing-based MPC is traditionally presented in the context of information-theoretic security, although many modern practical protocols that realise LSSS-based MPC often make use of computationally-secure primitives such as SHE (somewhat-homomorphic encryption) [23] or OT (oblivious transfer) [30]. In this paper, we focus on computationally-secure LSSS-based MPC.

An access structure for a set of parties defines which subsets of parties are allowed to discover the secret if they pool their information. Such quorums of parties are often called *qualified* sets of parties. An access structure is called $\mathcal{Q}_\ell$ (for $\ell \in \mathbb{N}$) if the union of any set of ℓ unqualified sets of parties is missing at least one party. We discuss this in some detail later, but for now the reader can think of an (n, t)-threshold scheme where $t < n/\ell$ which is where a subset of parties is qualified if and only if it is of size at least $t + 1$. Computationally-secure LSSS-based MPC has recently seen significant, efficient instantiations for full-threshold access structures [7, 21, 23, 30], which is where the protocol is secure if at least one party is honest, even if the adversary causes the corrupt parties to run arbitrary code (though this behaviour may cause the protocol to abort rather than provide output to the parties). In the threshold case similar efficient instantiations are known, such as the older VIFF protocol [20] which uses (essentially) information-theoretic primitives only.

While protocols providing full-threshold security are an important research goal, in the real world such guarantees of security do not always match the use-cases that appear. Different applications call for different access structures, and not necessarily the usual threshold examples. For example, a company may have four directors (CEO, CTO, CSO and CFO) and access may be granted in the two combinations (CEO and CFO) or (CTO and CSO and CFO). In such a situation it may be more efficient to tailor the protocol to this structure, rather than try to shoe-horn the application into a more standard (i.e. full-threshold) structure. Indeed, while it is possible that a computation can be performed in a full-threshold setting and then the outputs distributed in accordance with the access structure, such a process requires all parties to participate equally in the computation, which may not be feasible in the real world, especially if the computing parties are distributed over a wide network, and susceptible to outages if the total number of parties is large.

Most LSSS-based MPC protocols split the computation into two parts: an *offline phase*, in which parties interact using "expensive" public-key cryptography to lay the groundwork for an *online phase* in which only "cheap" information-theoretic primitives are required. The online phase is where the actual circuit evaluation takes place. For the access structures considered in this work, namely $\mathcal{Q}_2$ structures, the offline phase is almost as fast as the online phase. Thus the goal here is to minimize the cost of communication in both phases.

Realising MPC for different access structures has been well studied: shortly following the advent of Shamir's secret-sharing scheme [9, 35], the first formal MPC – as opposed to 2PC – protocols [5, 16, 26] were constructed, with varying correctness guarantees for different threshold structures. These works were developed by Hirt and Maurer [27], and then Beaver and Wool [3] to general access structures, culminating in Maurer's relatively more recent work [32]. In this latter work it is shown that passively-secure information-theoretic MPC is possible if the access structure is $\mathcal{Q}_2$, and full active security (without abort) is possible if the access structure is $\mathcal{Q}_3$.

In recent work [31], Keller et al. show that by generalising a method of Araki et al [1, 25] communication-efficient computationally-secure MPC with abort can be realised for $\mathcal{Q}_2$ access structures, if replicated secret-sharing is used. The methodology in [1,25,31] uses the explicit properties of replicated secret-sharing so as to authenticate various shares. This enables active security with abort to be achieved relatively cheaply, albeit at the expense in general of the pre-deployment of a large number (depending on the access structure) of symmetric keys to enable the generation of pseudo-random secret sharings (PRSS) in a non-interactive manner. A disadvantage of replicated sharing is the potentially larger (than average) memory footprint needed for each party per share, and consequently there is still a relatively large communication cost involved when the parties need to send shares across the network. In this work we extend this prior work to produce a protocol for *any multiplicative* LSSS which supports the $\mathcal{Q}_2$ access structure. We remark that Cramer et al. [18] showed that any LSSS realising a $\mathcal{Q}_2$ access structure can be made multiplicative by at most doubling the total number of shares.

1.1 Authentication of Shares

Many of practical MPC protocols begin with a basic passively-secure (a.k.a. *semi-honest* or *honest-but-curious*) protocol, in which corrupt parties execute the protocol honestly but try to deduce anything they can about other parties' data from their own data and the communication tapes. Such passively-secure protocols for $\mathcal{Q}_2$ access structures are highly efficient, and are information-theoretically secure. The passively secure protocols are then augmented to obtain active security with abort by using some form of "share authentication"; in this security setting, corrupt parties may deviate arbitrarily from the protocol description but if they do so the honest parties will abort the protocol.

At a high level, modern actively-secure LSSS-based MPC protocols combine:

1. A linear (i.e. additively homomorphic) secret sharing scheme;
2. A passive multiplication protocol; and
3. An authentication protocol.

The communication efficiency of the computation (usually an arithmetic or Boolean circuit) depends heavily on how authentication is performed.

In the full-threshold SPDZ [23] protocol and its successors, e.g. [21,30], authentication is achieved with additively homomorphic MACs (message authentication codes). For each secret that is shared amongst the parties, the parties also share a MAC on that secret. Since the authentication is additively homomorphic and the sharing scheme is linear, this means that the sum (and consequently scalar multiple of) of authenticated shares is authenticated "for free" by performing the addition (or scalar multiplication) on the associated MACs. More work is required for multiplication of secrets, but the general methodology for doing these operations on shared secrets is now generally considered "standard" for MPC in this setting.

One important branch of this authentication methodology contributing significantly to their practical performance is the amortisation of verification costs by batch-checking MACs, a technique developed in [6,23], amongst other works. A different approach to batch verification for authentication of shares, in the case of $\mathcal{Q}_2$ access structures, was introduced by Furakawa et al. [25], in the context of the three-party honest-majority setting, i.e. a $(3, 1)$-threshold access structure. This work extended a passively secure protocol of Araki et al. [1] in the same threshold setting. This approach dispenses with the MACs and instead achieves authentication of shares using a collision-resistant hash function when authenticating an open-to-all operation, and uses redundancy of the underlying secret sharing scheme in an open-to-one operation. Their protocol can be viewed as a bootstrapping of the passively-secure protocol of Beaver and Wool [3], with an optimised sharing procedure (highly tailored to the $(3, 1)$-threshold access structure), to provide a communication-efficient actively-secure protocol (with abort). By using a hash function they sacrifice the information-theoretic security of Beaver-Wool for computational security, and also use computationally-secure share generation operations to improve the offline phase.

The above protocols for replicated sharing in a $(3, 1)$-threshold access structure of [1,25] simultaneously reduce the number of secure communication channels needed *and* the total number of bits sent per multiplication. Recent work [31] has shown that these techniques can be generalised from $(3, 1)$-threshold to *any* $\mathcal{Q}_2$ access structure, using replicated secret-sharing. Both [25] and [31] make use of the fact that replicated sharing provides a trivial method to authenticate a full set of shares; i.e. it somehow offers a form of error-detection.

While replicated secret-sharing offers flexibility in being able to realise *any* access structure, unfortunately it *can* require an exponentially-large number of shares to be held by each party for each shared secret. As threshold access structures illustrate, using a general MSP may enable the same access structure to be realised in a more efficient manner; which motivates our work in this area.

1.2 Our contribution

Until recently most MPC protocols for multiplicative LSSSs for $\mathcal{Q}_2$ access structures have used the multiplicative property in essentially a black-box sense. The two major contributions of this work are as follows:

- Providing a generic way to optimise the passive multiplication subprotocol if we sacrifice information-theoretic security for computational security.
- Showing we can get authentication of shares almost for free, giving us active security with abort.

In Section 6 we also present an actively-secure offline phase (with abort) whose efficiency depends on the precise access structure. This offline phase avoids the exponential blow-up of computation and communication costs of [31].

Many of the previous protocols are optimised for access structures on specific numbers of parties, or use specific secret-sharing schemes. Our optimisation of the passive multiplication is generic in the sense that it only uses the multiplicativity for the multiplication and then the $\mathcal{Q}_2$ nature of the access structure for authentication, with [25] and [31] being special cases of our optimisation.

Our contribution, then, is not so much our full MPC protocol as it is the mechanism for an actively-secure multiplication in the $\mathcal{Q}_2$ setting. Viewing the protocol in this more modular sense allows us to separate the LSSS from the actual multiplication and thus allows us to reduce the search for finding an efficient MPC protocol for a given $\mathcal{Q}_2$ access structure to finding a multiplicative LSSS with a small total number of shares.

To conclude this section, we briefly remark how our work relates to the correspondence between LSSSs and linear codes. Cramer et al. [19] showed how the correspondence between linear secret-sharing schemes and linear codes reveals an efficient method by which qualified parties can *correct* any errors in a set of shares for some secret. The ability to do so requires the access structure to be $\mathcal{Q}_3$, since if this holds then a strongly-multiplicative LSSS realising it allows honest parties to correct any errors introduced by the adversary. This is not a direct connection to error-correction codes since such LSSSs do not necessarily allow unique decoding of the entire share vector: it is only the component of the share vector corresponding to the *secret* that is guaranteed to be correct. In our work we show that if the access structure is $\mathcal{Q}_2$ then any LSSS realising it allows honest parties to agree on whether or not the secret is correct: thus we obtain a form of error-detection. This reveals why the protocols above (viz., [25, 31]) are able to perform the error-detection causing abort. This result seems to be folklore – but we could find no statement or proof in the literature to this effect, and so we prove the required properties here.

2 Preliminaries

2.1 Notation

Let $\mathbb{F}$ denote a finite field; we write $\mathbb{F} = \mathbb{F}_q$ for q some prime power if $\mathbb{F}$ is the field of q elements. We write $r \overset{\$}{\leftarrow} \mathbb{F}$ to mean that r is sampled uniformly at random from $\mathbb{F}$. Vectors are written in bold and are taken to be column vectors. We denote by $\mathbf{0}$ a vector consisting entirely of zeros of appropriate dimension, determined by the context, and similarly by $\mathbf{1}$ a vector consisting entirely of ones. For a vector $\mathbf{x}$ we write the i^{th} component as $\mathbf{x}_i$, whereas $\mathbf{x}^i$ denotes the i^{th} vector from a sequence of vectors. We use the notation $\mathbf{e}^i$ for the i^{th} standard basis vector (defined by $\mathbf{e}^i_j := \delta_{ij}$ where δ_{ij} is the Kronecker delta). We denote by $[n]$ the set $\cup_{i=1}^n \{i\}$, and by $\mathcal{P}$ the complete set of parties, which we take to be $\{P_i\}_{i \in [n]}$. Given some set S, a subset of some larger set S', we write $a \notin S$ to indicate that element a is in $S' \setminus S$; in general, S' will be implicit, according to context. We define the function $\mathit{supp} : \mathbb{F}^m \rightarrow 2^{\mathcal{P}}$ via $\mathbf{s} \mapsto \{i \in [m] : \mathbf{s}_i \neq 0\}$. We use the notation $A \subseteq B$ to mean that A is a (not necessarily proper) subset of B, contrasted with $A \subsetneq B$ where A is a proper subset of B. We write λ and κ for the statistical and computational security parameters respectively.

Given a vector space $V \subseteq \mathbb{F}^d$, we denote by $V^\perp$ the orthogonal complement; that is, $V^\perp = \{\mathbf{w} \in \mathbb{F}^d : \langle \mathbf{v}, \mathbf{w} \rangle = 0\}$, where $\langle \mathbf{v}, \mathbf{w} \rangle = \mathbf{v}^\top \cdot \mathbf{w}$ is the standard inner product. From basic linear algebra, $(V^{\perp})^{\perp} = V$. For a matrix $M \in \mathbb{F}^{m \times d}$, we write $M^{\top}$ for the transpose. If M is a matrix representing a linear map $\mathbb{F}^d \to \mathbb{F}^m$, then $\mathrm{im}(M^{\top}) = \ker(M)^{\perp}$ by the fundamental theorem of linear algebra.

2.2 Access Structures, MSPs, LSSSs and Linear Codes

Access structures: Fix $\mathcal{P} = \{P_i\}_{i \in [n]}$ and let $\mathit{\Gamma} \subseteq 2^{\mathcal{P}}$ be a monotonically increasing set, i.e. $\mathit{\Gamma}$ is closed under taking supersets: if $Q \in \mathit{\Gamma}$ and $Q' \supseteq Q$ then $Q' \in \mathit{\Gamma}$. Similarly, let $\mathit{\Delta} \subseteq 2^{\mathcal{P}}$ be a monotonically decreasing set, i.e. $\mathit{\Delta}$ is closed under taking subsets: if $U \in \mathit{\Delta}$ and $U' \subseteq U$ then $U' \in \mathit{\Delta}$. We call the pair $(\mathit{\Gamma}, \mathit{\Delta})$ a *monotone access structure* if $\mathit{\Gamma} \cap \mathit{\Delta} = \varnothing$. If $\mathit{\Delta} = 2^{\mathcal{P}} \setminus \mathit{\Gamma}$, then we say the access structure is *complete*. In this paper, we will only be concerned with complete monotone access structures and so this is assumed throughout without qualification. The sets in $\mathit{\Gamma}$, usually denoted by Q, are called *qualified*, and the sets in $\mathit{\Delta}$, usually denoted by U, are called *unqualified*. Partial ordering is induced on $\mathit{\Gamma}$ and $\mathit{\Delta}$ by the standard subset relation denoted by "$\subseteq$": we write $\mathit{\Gamma}^-$ for the set of *minimally qualified sets* where minimality is with respect to "$\subseteq$": $\mathit{\Gamma}^- = \{Q \in \mathit{\Gamma} : \text{ if } Q' \in \mathit{\Gamma} \text{ and } Q' \subseteq Q \text{ then } Q' = Q\}$; similarly, $\mathit{\Delta}^+$ denotes the set of *maximally unqualified sets* where maximality is with respect to "$\subseteq$": $\mathit{\Delta}^+ = \{U \in \mathit{\Delta} : \text{ if } U' \in \mathit{\Delta} \text{ and } U \subseteq U' \text{ then } U' = U\}$

An access structure is said to be $\mathcal{Q}_2$ (resp. $\mathcal{Q}_3$) if the union of no two (resp. three) sets in $\mathit{\Gamma}$ is the whole of $\mathcal{P}$. A consequence of this is that in a $\mathcal{Q}_2$ access structure, the complement of a qualified set is unqualified, and *vice versa*.

In an (n, t)-threshold access structure, any set of $t + 1$ parties is qualified, whilst any set of t or fewer parties is unqualified. Thus $\mathit{\Gamma}^-$ contains $\binom{n}{t+1}$ sets in total. The term *full threshold* refers to an $(n, n-1)$-threshold access structure. For an arbitrary complete monotone access structure, the set of minimally qualified sets together with the set of maximally unqualified sets uniquely determine the entire structure. The dual access structure $\mathit{\Gamma}^*$ of an access structure $\mathit{\Gamma}$ is defined by $\mathit{\Gamma}^* := \{Q \in 2^{\mathcal{P}} : 2^{\mathcal{P}} \setminus Q \notin \mathit{\Gamma}\}$. Cramer et al. [19] showed that an access structure $\mathit{\Gamma}$ is $\mathcal{Q}_2$ if and only if $\mathit{\Gamma}^* \subseteq \mathit{\Gamma}$.

Linear Secret Sharing Schemes: An LSSS is a method of sharing secret data amongst parties. It consists of three multi-party algorithms: Input, Open, and ALF (affine linear function), allowing parties to provide secret inputs, reveal (or *open*) secrets, and compute an affine linear function on shared secrets. In a practical sense, this means that the parties can add secrets, multiply by scalars, and add public constants to a shared secret, all by local computations. In this work we consider, as examples, the three most well-known secret-sharing schemes: Shamir; replicated, also known as CNF-based (conjunctive-normal-form-based); and DNF-based (disjunctive-normal-form-based). We assume the reader is familiar with these schemes, but for completeness we give a recap in Appendix A. We will use the term *additive sharing* to mean that a secret s takes the value $s = \sum_{i=1}^{n} s_i$ where s_i is held by P_i and the s_i's are uniformly random subject to the constraint that they sum to s.

An LSSS is called *multiplicative* if the whole set of parties $\mathcal{P}$ can compute an additive sharing of the product of two secrets by performing only local computations. If the product is to be kept as a secret and used further in the computation, it is usually necessary for the parties to engage in one or more rounds of communication to convert the additive sharing into a sharing in the LSSS being used. A secret-sharing scheme is called *strongly multiplicative* if, for any $U \in \Delta$, the parties in $\mathcal{P} \backslash U$ can compute an additive sharing of the product of two secrets by local computations. Such schemes offer robustness, since the adversary, corrupting an unqualified set of parties, cannot prevent the honest parties from reconstructing the desired secret. Cramer et al. [18] showed that any (non-multiplicative) LSSS realising a $\mathcal{Q}_2$ access structure can be converted to a multiplicative LSSS for the same access structure so that each party holds at most twice the number of shares it held originally. There is currently no known construction to convert an arbitrary $\mathcal{Q}_3$ LSSS to a strongly multiplicative LSSS with only polynomial blow-up in the number of shares each party must hold [18, 19].

Monotone Span Programs: Span programs, and monotone span programs specifically, were introduced by Karchmer and Wigderson [29] as a model of computation. It has been shown that MSPs have a close relationship to secret-sharing schemes, as discussed informally below.

Definition 1. *A* Monotone Span Program *(MSP), denoted by $\mathcal{M}$, is a quadruple $(\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ where $\mathbb{F}$ is a field, $M \in \mathbb{F}^{m \times d}$ is a full-rank matrix for some m and $d \leq m$, $\boldsymbol{\varepsilon} \in \mathbb{F}^d$ is an arbitrary non-zero vector called the* target vector*, and $\psi : [m] \twoheadrightarrow \mathcal{P}$ is a surjective "labelling" map of rows to parties. The size of $\mathcal{M}$ is defined to be m, the number of rows of the matrix M.*

Typically, $\boldsymbol{\varepsilon} = \mathbf{e}^1$ or $\boldsymbol{\varepsilon} = \mathbf{1}$, but it can be an arbitrary non-zero vector: changing it simply changes how the vector $\mathbf{x}$ is selected, and corresponds to performing column operations on the columns of M, which does not change the access structure the MSP realises by results of Beimel et al. [4]. Some definitions of MSP do not require that M have full rank, since if this is not the case, one can iteratively remove any columns which are linearly dependent on preceding columns without changing the access structure $\mathcal{M}$ computes. We include this assumption for the sake of simplicity later on.

We say that the row-map ψ defines which rows are "owned" by each party. Given a set $S \subseteq \mathcal{P}$, we denote by M_S the submatrix of M whose rows are indexed by the set $\{i \in [m] : \psi(i) \in S\}$, and similarly $\mathbf{s}_S$ is the subvector of $\mathbf{s}$ whose entries are indexed by the same. Later, we will somewhat abuse notation by denoting again by M_S, where now $S \subseteq [m]$, the submatrix whose rows are indexed by S. Context will determine which matrix we mean since the indexing set is either a set of parties, or a set of row indices. If $\mathbf{s} \in \mathbb{F}^m$, then we call $\mathbf{s}_Q$ a *qualified subvector* of $\mathbf{s}$ if $Q \in \Gamma$, and an *unqualified subvector* otherwise. An MSP $\mathcal{M}$ is said to compute an access structure Γ if for all $Q \in \Gamma$, $\exists\ \boldsymbol{\lambda}^Q \in \mathbb{F}^m$ (i.e. depending on Q) such that $M^\top \cdot \boldsymbol{\lambda}^Q = \boldsymbol{\varepsilon}$ and $\psi(\mathit{supp}(\boldsymbol{\lambda}^Q)) \subseteq Q$. Note that we write $\boldsymbol{\lambda}^Q$ to show that this vector is associated to the set Q; compare with $\boldsymbol{\lambda}^Q_Q$, which is the subvector of $\boldsymbol{\lambda}^Q$ whose co-ordinates are indexed by Q, to be consistent with the notation above. This means that the parties in the set Q "own" rows of the matrix M which can be combined in a public, known linear combination encoded as the vector $\boldsymbol{\lambda}^Q$, to obtain the target vector $\boldsymbol{\varepsilon}$.

Monotone Span Programs induce LSSSs in the following way: Sample $\mathbf{x} \xleftarrow{\$} \mathbb{F}^d$ subject to $\langle \mathbf{x}, \boldsymbol{\varepsilon} \rangle = s$, the secret. Now let $\mathbf{s} = M \cdot \mathbf{x}$ and for each $i \in [m]$, give $\mathbf{s}_i$ (that is, the i^{th} co-ordinate) to party $\psi(i)$. Thus party P_i has the vector $\mathbf{s}_{\{P_i\}}$. We call $\mathbf{x}$ the *randomness vector* since $\mathbf{x}$ is chosen uniformly at random, subject to $\langle \mathbf{x}, \boldsymbol{\varepsilon} \rangle = s$, to generate $\mathbf{s} := M \cdot \mathbf{x}$, the *share vector*. The co-ordinates of $\mathbf{s}$ are precisely the shares of the secret which are distributed to parties according to the mapping ψ. We say that a share vector $\mathbf{s}$ *encodes* a secret s if $\mathbf{s} = M \cdot \mathbf{x}$ for some $\mathbf{x}$ where $\langle \mathbf{x}, \boldsymbol{\varepsilon} \rangle = s$. An MSP is called *ideal* if ψ is injective; since it is surjective by definition, an ideal MSP is an MSP for which ψ is bijective – i.e. each party receives exactly one share.

The associated access structure for an MSP is such that $\boldsymbol{\varepsilon}$ is contained in the linear span of the rows of M owned by any qualified set of parties, and also so that $\boldsymbol{\varepsilon}$ is *not* in the linear span of the rows owned by any unqualified set of parties. It is well known that, given a monotone access structure $(\mathit{\Gamma}, \mathit{\Delta})$, there exists an MSP $\mathcal{M}$ computing it [24, 28, 29].

In more detail: A qualified set of parties $Q \in \mathit{\Gamma}$ can compute the secret from the qualified subvector $\mathbf{s}_Q$ because by construction of M there is a publicly-known recombination vector $\boldsymbol{\lambda}$ associated to this set Q such that $\psi(\mathit{supp}(\boldsymbol{\lambda})) \subseteq Q$ and $M^\top \boldsymbol{\lambda} = \boldsymbol{\varepsilon}$. Note that while $\psi(\mathit{supp}(\boldsymbol{\lambda})) \subseteq Q$, this subset of Q must still be qualified – it just may be the case that not all of the parties' shares are required to reconstruct the secret (for example, if multiple parties hold the same share). Since $\psi(\mathit{supp}(\boldsymbol{\lambda})) \subseteq Q$, we have $\langle \boldsymbol{\lambda}, \mathbf{s} \rangle = \langle \boldsymbol{\lambda}_Q, \mathbf{s}_Q \rangle$, so given $\mathbf{s}_Q$ the parties can compute $\langle \boldsymbol{\lambda}_Q, \mathbf{s}_Q \rangle$, and since $\langle \boldsymbol{\lambda}_Q, \mathbf{s}_Q \rangle = \langle \boldsymbol{\lambda}, \mathbf{s} \rangle = \langle \boldsymbol{\lambda}, M \cdot \mathbf{x} \rangle = \langle M^\top \boldsymbol{\lambda}, \mathbf{x} \rangle = \langle \boldsymbol{\varepsilon}, \mathbf{x} \rangle = s$, they can thus determine the secret.

Conversely, for any unqualified set of parties $U \in \mathit{\Delta}$, again by construction of M we have that $\boldsymbol{\varepsilon} \notin \operatorname{im}(M_U^\top)$, which is equivalent to each of the following three statements:

- $\boldsymbol{\varepsilon} \notin \ker(M_U)^\perp$
- $\exists\ \mathbf{k} \in \ker(M_U)$ such that $\langle \boldsymbol{\varepsilon}, \mathbf{k} \rangle \neq 0$
- $\exists\ \mathbf{k} \in \mathbb{F}^d$ such that $M_U \cdot \mathbf{k} = \mathbf{0}$ with $\langle \boldsymbol{\varepsilon}, \mathbf{k} \rangle = 1$

From the last statement, we can see that for any secret s, for any randomness vector $\mathbf{x} \in \mathbb{F}^d$ encoding it – i.e. where $\langle \mathbf{x}, \boldsymbol{\varepsilon} \rangle = s$ – for any other secret $s' \in \mathbb{F}$ we have $M_U \mathbf{x} = M_U \mathbf{x} + \mathbf{0} = M_U \mathbf{x} + M_U((s' - s) \cdot \mathbf{k}) = M_U(\mathbf{x} + (s' - s) \cdot \mathbf{k})$. Thus if $\mathbf{x}$ encodes the secret s, then the randomness vector $\mathbf{x} + (s' - s) \cdot \mathbf{k}$ encodes s' by linearity of the inner product, but the share vectors held by parties in U are the same. Thus the set of shares received by an unqualified set of parties provides no information about the secret.

In this work we show that for any MSP computing any $\mathcal{Q}_2$ access structure, there exists a matrix N such that for any vector $\mathbf{e} \neq \mathbf{0}$ for which $\psi(\mathit{supp}(\mathbf{e})) \notin \mathit{\Gamma}$, we either have $N \cdot \mathbf{e} \neq \mathbf{0}$, or $N \cdot \mathbf{e} = \mathbf{0}$ and $\langle \mathbf{e}, \boldsymbol{\varepsilon} \rangle = 0$. The matrix N is essentially the parity-check matrix of the code generated by the matrix M of the MSP and turns out to be very useful for efficiently detecting cheating behaviour.

2.3 MPC

Network: We assume secure point-to-point channels. When broadcasting shares but we do not assume broadcast channels: in this context we mean an honest party sends the same element to each other party over the given secure channel.

Security Model: Our protocols are modelled and proved secure in the Universal Composability (UC) framework introduced by Canetti [14] and we assume the reader is familiar with it. We assume static corruptions by the adversary, meaning that the adversary corrupts some set of parties once at the beginning of the protocol. We will usually denote the set of parties the adversary corrupts by $A \subseteq \mathcal{P}$. We assume the adversary is active, meaning that the corrupted parties may execute arbitrary code determined by the adversary, and additionally we allow the protocol to abort prematurely – i.e. the protocols are *actively-secure with abort*. The protocol is secure against a computationally bounded adversary (who successfully cheats by finding a collision of the hash function).

Pre-processing: Many modern MPC protocols split computation into two phases, the *offline* or *pre-processing phase* and the *online phase*. In the offline phase, the parties engage in several rounds of communication to produce data which can then be used in the online phase. The purpose of doing this is that the pre-processing can be done at any time prior to the execution of the online phase, can be made independent of the function to be computed, and may use expensive public-key primitives, in order to allow the online phase to use only fast information-theoretic primitives. In our protocol design, we follow this model, although we only require symmetric-key primitives throughout since the access structure is $\mathcal{Q}_2$.

Hash API

The hash API implemented via the hash function $H : \mathbb{F}^* \to \{0,1\}^\lambda$ consists of the following three algorithms:

- $H.\mathsf{Initialise}()$: the hash function is initialised.
- $H.\mathsf{Update}(\mathbf{s})$: the hash function is updated with the vector $\mathbf{s}$.
- $H.\mathsf{Output}()$: the hash function is evaluated and output provided.

Figure 1. Hash API

Hash authentication: The work of Furakawa et al. [25] is in the three-party honest majority case. A secret is additively split into three parts, and each party is given a different set of two of them. To open a secret, each party sends one field element to one other party. This suffices for all parties to obtain all shares, but does not ensure that the one corrupt party sent the correct share. This is where the hash evaluation comes in: after a secret is opened, all parties update their hash function locally with all three shares (the two they held and the one they received); after possibly many secrets are opened, the parties broadcast (here meaning each party sends to the other two parties over a secure channel) the outputs of their hash evaluations and compare what they received with what they computed themselves. If any hashes differ, they abort. This process ensures that the shares held by all parties are consistent, even though each party need only send one share to one party per opening. If many shares are opened in the execution of the protocol (as is the case in SPDZ-like protocols, since every multiplication requires two secrets to be opened), this significantly reduces communication overhead, at the cost of cryptographic assumptions for the existence of a collision-resistant hash function. This was generalised to any replicated scheme $\mathcal{Q}_2$ LSSS by Keller et al. [31].

In our work, we apply similar techniques to Furukawa et al. and Keller et al. to the problem of opening values to parties, but in a significantly more general case. We achieve this by proving the folklore results that say an LSSS is error-detecting if and only if it is $\mathcal{Q}_2$. Our protocol will use the "standard" hash function API given in Figure 1; in brief, our methods are as follows:

- If single party P_i is required to learn a secret, all the other parties send all of their shares to P_i, and then P_i performs an error-detection check on the shares received, telling all parties to abort if errors are detected.
- If all parties are required to learn a secret, the parties engage in a round of communication in which not all parties need to communicate with each other. The parties reconstruct a view of what they think other parties have received, even if they have not communicated with all other parties. After opening possibly many secrets, each party calls Output on the hash function, broadcasts their output, and checks every other party's hash value against their own; we will see that this process authenticates the secrets.

In the next two sections we outline why the methodologies for the two cases are correct. The proof of security of our protocol can be found in Figure 14 and Figure 15 in Appendix B, but it is best read in the context of the next sections.

3 Opening a Value to One Party

In this section, we show that for an LSSS realising a $\mathcal{Q}_2$ access structure, if the share vector $\mathbf{s}$ for some secret s is modified with an error vector $\mathbf{e}$ with unqualified support then $\mathbf{s}+\mathbf{e}$ is either no longer a valid share vector (i.e. is not in $\text{im}(M)$), or the error vector encodes 0, and so by linearity $\mathbf{s}+\mathbf{e}$ also encodes s. In our MPC protocol, this will provide an efficient method by which a party to whom a secret is opened (by all other parties sending that party all of their shares) can check whether or not the adversary has introduced an error. The procedure of opening to a single party is necessary in order for the parties to provide input and obtain output in an actively-secure manner.

Lemma 1. *For any MSP $\mathcal{M} = (\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ computing a $\mathcal{Q}_2$ access structure Γ, for any vector $\mathbf{s} \in \mathbb{F}^m$,*

$$\psi(supp(\mathbf{s})) \notin \Gamma \implies \begin{cases} \mathbf{s} \notin im(M), \text{ or} \\ \mathbf{s} \in im(M) \text{ and } \mathbf{s} = M\mathbf{x} \text{ for some } \mathbf{x} \in \mathbb{F}^d \text{ where } \langle \mathbf{x}, \boldsymbol{\varepsilon} \rangle = 0. \end{cases}$$

Proof. If $\psi(supp(\mathbf{s})) \notin \Gamma$ then $\mathcal{P} \setminus \psi(supp(\mathbf{s})) \in \Gamma$ since the access structure is $\mathcal{Q}_2$. Thus there is at least one set $Q \in \Gamma$ where $Q \subseteq \mathcal{P} \setminus \psi(supp(\mathbf{s}))$ for which $\mathbf{s}_i = 0$ for all $i \in [m]$ where $\psi(i) \in Q$ (i.e. $\mathbf{s}_Q = \mathbf{0}$), by definition of *supp*.

Recall that for a qualified set Q of parties to reconstruct the secret, they take the appropriate recombination vector $\boldsymbol{\lambda}$ (which has the property that $\psi(supp(\boldsymbol{\lambda})) \subseteq Q$) and compute $s = \langle \boldsymbol{\lambda}, \mathbf{s} \rangle$. For this particular Q and corresponding recombination vector $\boldsymbol{\lambda}$, we have $\langle \boldsymbol{\lambda}, \mathbf{s} \rangle = \langle \boldsymbol{\lambda}_Q, \mathbf{s}_Q \rangle$ since $\psi(supp(\boldsymbol{\lambda})) \subseteq Q$, and $\langle \boldsymbol{\lambda}_Q, \mathbf{s}_Q \rangle = \langle \boldsymbol{\lambda}_Q, \mathbf{0} \rangle = 0$ by the above, so the secret is 0.

If $\mathbf{s} \in \operatorname{im}(M)$ then every set $Q \in \Gamma$ must compute the secret as 0 by the definition of MSP (though note that it is not necessarily the case that $\mathbf{s}_Q = \mathbf{0}$ for all $Q \in \Gamma$). Thus the share vector $\mathbf{s}$ is in $\operatorname{im}(M)$ and encodes the secret $s = 0$.

Otherwise, $\mathbf{s} \notin \operatorname{im}(M)$, and we are done. □

We now show that if the adversary (controlling an unqualified set of parties) adds an error vector $\mathbf{e}$ to a share vector $\mathbf{s}$, the resulting vector $\mathbf{c} := \mathbf{s}+\mathbf{e}$ will either not be a valid share vector, or will encode the same secret as $\mathbf{s}$ (by linearity). Adding in an error $\mathbf{e}$ that does not change the value of the secret can be viewed as the adversary re-randomising the shares he holds for corrupt parties.

Lemma 2. *Let $\mathcal{M} = (\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ be an MSP computing $\mathcal{Q}_2$ 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 secret s, with error $\mathbf{e}$. Then there exists a matrix N such that*

$$\psi(supp(\mathbf{e})) \notin \Gamma \implies \text{either } \mathbf{e} \text{ encodes the error } e = 0, \text{ or } N \cdot \mathbf{c} \neq \mathbf{0}$$

Proof. Let N be any matrix whose rows form a basis of $\ker(M^\top)$ and suppose $\mathbf{e} \in \mathbb{F}^m$. By the fundamental theorem of linear algebra, $\ker(M^\top) = \operatorname{im}(M)^\perp$, so $\mathbf{s} \in \operatorname{im}(M)$ if and only if $N \cdot \mathbf{s} = \mathbf{0}$. Since $\psi(supp(\mathbf{e})) \notin \Gamma$, then by Lemma 1 we have that either $\mathbf{e} \notin \operatorname{im}(M)$, or $\mathbf{e} \in \operatorname{im}(M)$ and $e = 0$.

If $\mathbf{e} \in \operatorname{im}(M)$ then $e = 0$ and we are done, whilst if $\mathbf{e} \notin \operatorname{im}(M)$ then $N \cdot \mathbf{e} \neq \mathbf{0}$. In the latter case, since $\mathbf{s} \in \operatorname{im}(M)$ we have $N \cdot \mathbf{s} = \mathbf{0}$ and hence $N \cdot \mathbf{c} = N \cdot (\mathbf{s}+\mathbf{e}) = N \cdot \mathbf{s} + N \cdot \mathbf{e} = \mathbf{0} + N \cdot \mathbf{e} \neq \mathbf{0}$. □

The matrix N is usually called the *cokernel* of M, and can be viewed as the parity-check matrix of the code defined by generator matrix M. The method to open a secret to a single party P_i is then immediate: all parties send their shares to P_i, who then concatenates the shares into a share vector $\mathbf{s}$ and computes $N \cdot \mathbf{s}$. Since the adversary controls an unqualified set of parties, if $N \cdot \mathbf{s} = 0$ then by Lemma 2 the share vector $\mathbf{s}$ encodes the correct secret. In this case, P_i recalls any recombination vector $\boldsymbol{\lambda}$ and computes the secret as $s = \langle \boldsymbol{\lambda}, \mathbf{s} \rangle$, and otherwise tells the parties to abort.

4 Opening a Value to All Parties

To motivate our procedure for opening to all parties and to show that it is correct, we first discuss the naïve method of opening shares in a semi-honest protocol, then show how to reduce the communication, and then explain how to obtain a version which is actively-secure (with abort).

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

When bootstrapping to active security, we see that the redundancy allows verification of opened secrets: honest parties can check *all* other parties' broadcasted shares for correctness. When reducing communication with the aim of avoiding the redundancy of broadcasting, honest parties must still be able to detect when the adversary sends inconsistent or erroneous shares. In particular, parties *not* receiving shares from the adversary must also be able to detect that cheating has occurred in spite of not directly being sent erroneous shares.

To achieve this, in our protocol each party will receive enough shares from other parties to determine "optimistically" all shares held by all parties – that is, reconstruct the entire share vector – and then all parties will compare their reconstructed share vectors. To amortise the cost of comparison, the parties will actually update a local collision-resistant hash-function each time they reconstruct a new share vector and will then compare the final output of the hash function at the end of the computation, when output is required. This, in essence, is the idea behind the protocols of Furakawa et al. [25] and Keller et al. [31] that are tailored to replicated secret-sharing.

To fix ideas, consider the case of Shamir's scheme; a set of $t + 1$ distinct points determines a unique polynomial of degree at most t that passes through them. This fact not only enables the secret to be computed using $t + 1$ shares, but additionally enables determining the entire polynomial (the coefficients of which are the share vector for the scheme) and consequently all other shares. For some LSSSs it is not the case that *any* qualified set of parties have enough information to reconstruct all shares[3].

To allow the parties to perform reconstruction, each party is assigned a set of shares that it will receive, which we encode as a map $\mathsf{q} : \mathcal{P} \to 2^{[m]}$ defined as follows: for each $P_i \in \mathcal{P}$, define $\mathsf{q}(P_i)$ to be a set $S_i \subseteq [m]$ such that:

- $\ker(M_{S_i}) = \{\mathbf{0}\}$; that is, the kernel of the submatrix M restricted to the rows indexed by S_i, is trivial; and

[3] In Appendix D we provide a formal description of MSPs in which all qualified sets of parties can reconstruct the entire share vector and explain how such MSPs are "good" for our protocol.

- $\psi^{-1}(\{P_i\}) \subseteq S_i$, where ψ^{-1} denotes the preimage of the row-map ψ; that is, each party includes all of their own shares in the set S_i.

These sets are used as follows. Each P_i receives a set of shares, denoted by $\mathbf{s}^i_{\mathsf{q}(P_i)}$, for a given secret. Then in order to reconstruct all shares, P_i tries to find $\mathbf{x}^i$ such that $\mathbf{s}^i_{\mathsf{q}(P_i)} = M_{\mathsf{q}(P_i)} \cdot \mathbf{x}^i$ and then computes $\mathbf{s}^i = M \cdot \mathbf{x}^i$ as the reconstructed share vector, which is then used to update the hash function (locally). Trivially, we can take $\mathsf{q}(P_i) = [m]$ for all $P_i \in \mathcal{P}$, which corresponds to broadcasting all shares; however, better choices of q result in better communication efficiency. In Appendix C, we give a somewhat-optimised algorithm for finding a "good" map q for a given MSP.

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

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

Indeed, the only thing the adversary can do without causing abort – either immediately or later on when hashes are compared – is to change his shares so that his shares combined with the honest parties' shares form a valid share vector. Intuitively, one can think of this as the adversary re-randomising the shares owned only by corrupt parties, which is not possible in Shamir or replicated secret-sharing, but is in DNF-based sharing, and in general is only possible if the LSSS admits non-trivial share vectors with unqualified support.

More formally, we have the following lemma that shows that if all parties *can* reconstruct share vectors and the share vectors are consistent, then the adversary cannot have introduced an error.

Lemma 3. *Let $\mathsf{q} : \mathcal{P} \to 2^{[m]}$ be defined as above and let $\mathbf{s}^i_{\mathsf{q}(i)}$ denote the subvector of shares received by party P_i for a given secret. Suppose it is possible for each party $P_i \in \mathcal{P}$ to find a vector $\mathbf{x}^i$ such that $\mathbf{s}^i_{\mathsf{q}(P_i)} = M_{\mathsf{q}(P_i)}\mathbf{x}^i$; let $\mathbf{s}^i := M \cdot \mathbf{x}^i$ for each $i \in [n]$. If $\mathbf{s}^i = \mathbf{s}^j$ for all honest parties P_i and P_j, then the adversary did not introduce an error on the secret.*

Proof. The existence of q follows from the fact that "at worst" we can take $\mathsf{q}(P_i) = [m]$ for all $P_i \in \mathcal{P}$. There is a unique $\mathbf{x}^i$ solving $\mathbf{s}^i_{\mathsf{q}(P_i)} = M_{\mathsf{q}(P_i)} \cdot \mathbf{x}^i$ (not *a priori* necessarily the same for all parties) because $\ker(M_{\mathsf{q}(P_i)}) = \{\mathbf{0}\}$ for all $P_i \in \mathcal{P}$ by the first requirement in the definition of q.

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

Each honest party uses their own shares in the reconstruction process by the second requirement in the definition of q, so if $\mathbf{s}^i = \mathbf{s}^j$ for all honest parties P_i and P_j, then in particular they all agree on a qualified subvector defined by

Protocol Π_{Opening}

For each $P_i \in \mathcal{P}$, the parties decide on some $\boldsymbol{\lambda}^i$, which is any recombination vector such that $supp(\boldsymbol{\lambda}^i) \subseteq \mathsf{q}(P_i)$. See Section 4 for the definition of q. We denote by H^i the hash function updated locally by P_i which will be initialised as in Figure 5, at the start of the MPC protocol. If at any point a party receives the message Abort, it runs the subprotocol **Abort**.

OpenTo(i):
If $i = 0$, the secret s encoded via share vector $\mathbf{s}$, is to be opened to all, otherwise it is to be opened to only player i.

If $i = 0$ then each $P_j \in \mathcal{P}$ does the following:

1. Retrieve from memory the recombination vector $\boldsymbol{\lambda}^j$.
2. For each $P_\ell \in \mathcal{P}$, for each $k \in \mathsf{q}(P_\ell)$, if $\psi(k) = P_j$ then send $\mathbf{s}_k$ to P_ℓ.
3. For each $k \in \mathsf{q}(P_j)$, wait to receive $\mathbf{s}_k$ from party $\psi(k)$.
4. Concatenate local and received shares into a vector denoted by $\mathbf{s}^j_{\mathsf{q}(P_j)} \in \mathbb{F}^{|\mathsf{q}(P_j)|}$.
5. (Locally) output $s = \langle \boldsymbol{\lambda}^j_{\mathsf{q}(P_j)}, \mathbf{s}^j_{\mathsf{q}(P_j)} \rangle$.
6. Solve $M_{\mathsf{q}(P_j)} \cdot \mathbf{x}^j = \mathbf{s}^j_{\mathsf{q}(P_j)}$ for $\mathbf{x}^j$. If there are no solutions, run **Abort**.
7. Execute $H^j.\mathsf{Update}(M\mathbf{x}^j)$.

If $i \neq 0$, the secret encoded via share vector $\mathbf{s}$ is to be opened to party P_i. The parties do the following:

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

Verify: Each $P_i \in \mathcal{P}$ does the following:

1. Compute $h^i := H^i.\mathsf{Output}()$.
2. Send h^i to all other parties $P_j \in \mathcal{P} \setminus \{P_i\}$ over pair-wise secure channels.
3. Wait for h^j from all other parties $P_j \in \mathcal{P} \setminus \{P_i\}$.
4. If $h^j \neq h^i$ for any j, run **Abort**.

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

Broadcast: When P_i calls this procedure to broadcast a value s,

1. Party P_i sends the secret s to all other players over pair-wise secure channels.
2. When party P_j receives the share, it executes $H^j.\mathsf{Update}(s)$.

Figure 2. Protocol Π_{Opening}

honest shares – i.e. $\mathbf{s}^i_Q = \mathbf{s}^j_Q$ for all honest parties P_i and P_j. Thus some qualified subvector of the share vector is well defined, which uniquely defines the secret by definition of MSP. □

As mentioned in the introduction, our results in the last two sections are somewhat analogous to the result of Cramer et al. [19, Thm 1] which roughly shows that for a strongly multiplicative LSSS implementing a $\mathcal{Q}_3$ access structure, honest parties can always agree on the correct secret (when all parties broadcast their shares). In Figure 2 we present the methods we use to open secret shared data in different situations.

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

The functionality maintains a list Value of secrets that it stores. The set A indexes the corrupt parties (unknown to the honest parties).

Triples: On input (Triple, N_T) from all parties, the functionality does the following:

1. For i from 1 to N_T:
   (a) Sample $a^i, b^i \xleftarrow{\$} \mathbb{F}$ and compute share vectors $\mathbf{a}^i$ and $\mathbf{b}^i$.
   (b) Send $(\mathbf{a}^i_A, \mathbf{b}^i_A)$ to the adversary.
   (c) Receive a subvector of shares $\tilde{\mathbf{c}}^i_A$ from the adversary.
   (d) Compute a vector $\mathbf{c}^i = M \cdot \mathbf{x}^i_c$ such that $\langle \mathbf{x}^i_c, \boldsymbol{\varepsilon} \rangle = a^i \cdot b^i$ and $\mathbf{c}^i_A = \tilde{\mathbf{c}}^i_A$. If no such vector $\mathbf{c}^i$ exists, set an internal flag $\mathtt{Abort}$ to true and continue.
2. Wait for a message OK or Abort from the adversary.
3. If the response is OK and the internal flag $\mathtt{Abort}$ has not been set to true, for each honest $P_i \in \mathcal{P}$, send $(\mathbf{a}^i_{\{P_i\}}, \mathbf{b}^i_{\{P_i\}}, \mathbf{c}^i_{\{P_i\}})_{i=1}^{N_T}$ to each honest party P_i, and otherwise output the message Abort to all honest parties and abort.

Figure 3. Functionality $\mathcal{F}_{\mathsf{Prep}}$

5 MPC Protocol

We are now ready to present our protocol to implement the MPC functionality offering active security with abort as given in Figure 4. We present the online method here, leaving the offline method for Section 6, which, as discussed in Section 6.5, is much more scalable than [31] since the dependence on replicated secret-sharing is removed. The offline method implements the functionality given in Figure 3. Our online protocol, in Figure 5, makes use of the opening protocol $\mathit{\Pi}_{\mathsf{Opening}}$ given in Figure 2 earlier. The majority of our protocol uses standard MPC techniques for secret-sharing. In particular, the equation the parties compute for the multiplication is a standard application of Beaver's circuit randomisation technique [2], albeit for a general LSSS.

Correctness of our input procedure follows from the input method given in the Non-Interactive Pseudo-Random Secret-Sharing protocol of [17]. In particular for party P_i to provide an input s in a secret-shared form $\mathbf{s}$, the parties will first take a secret-sharing $\mathbf{r}$ of a uniformly random secret r – which is some a or b

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

Initialise: On input Init from all parties, the functionality initialises the array $\mathsf{Value}[]$. Accept a message OK or Abort from the adversary; if the message is OK then continue, and otherwise send the message Abort to all parties and abort.

Input: On input $(\mathsf{Input}, \mathsf{id}, x)$ from party P_i and $(\mathsf{Input}, \mathsf{id}, \perp)$ from all other parties, where id is a fresh identifer, the functionality sets $\mathsf{Value}[\mathsf{id}] := x$.

Add: On input $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, if id_3 is a fresh identifier and $\mathsf{Value}[\mathsf{id}_1]$ and $\mathsf{Value}[\mathsf{id}_2]$ have been defined, the functionality sets $\mathsf{Value}[\mathsf{id}_3] := \mathsf{Value}[\mathsf{id}_1] + \mathsf{Value}[\mathsf{id}_2]$.

Multiply: On input $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, if id_3 is a fresh identifier and $\mathsf{Value}[\mathsf{id}_1]$ and $\mathsf{Value}[\mathsf{id}_2]$ have been defined, the functionality waits for a message OK or Abort from the adversary. If the adversary sends the message Abort, send the message Abort to all parties and the adversary and abort, and otherwise set $\mathsf{Value}[\mathsf{id}_3] := \mathsf{Value}[\mathsf{id}_1] \cdot \mathsf{Value}[\mathsf{id}_2]$.

Output: On input $(\mathsf{Output}, \mathsf{id}, i)$ from all parties, if $\mathsf{Value}[\mathsf{id}]$ has been defined, the functionality does the following:

- If $i = 0$, send $\mathsf{Value}[\mathsf{id}]$ to the adversary and wait for a signal OK or Abort in return. If it signals Abort, send the message Abort to all parties and the adversary and abort, and otherwise send $\mathsf{Value}[\mathsf{id}]$ to all parties. If not aborted, wait for another signal OK or Abort. If the adversary signals Abort, send the message Abort to all parties and the adversary and abort.
- If $i \neq 0$ and P_i is corrupt, then the functionality sends $\mathsf{Value}[\mathsf{id}]$ to the adversary and waits for the adversary to signal OK or Abort. If it signals Abort, send the message Abort to all parties and abort.
- If $i \neq 0$ and P_i is honest, the functionality waits for the adversary to signal OK or Abort. If it signals Abort, send the message Abort to all parties and the adversary and abort, and otherwise send $\mathsf{Value}[\mathsf{id}]$ to P_i.

Figure 4. Functionality $\mathcal{F}_{\mathsf{MPC}}$

from a Beaver triple – and open it by calling **OpenTo**(i). Then P_i determines the encoded secret (using any recombination vector) and broadcasts $\epsilon := s - r$. The parties compute the share vector as $\mathbf{s} := \epsilon \cdot \mathbf{u} + \mathbf{r}$ where $\mathbf{u}$ is a pre-agreed sharing of 1, which may be the same vector used to compute all inputs, by which we mean that for $i \in [m]$, party $\psi(i)$ computes $\mathbf{s}_i := \epsilon \cdot \mathbf{u}_i + \mathbf{r}_i$. Since this r is uniformly random by assumption, it hides the input s in the broadcast of ϵ. This is proved formally in our simulation proof.

We have the following proposition, which we prove in Appendix B under the UC framework of Canetti [15]. Here we use $(\Pi_{\mathsf{MPC}} \| \Pi_{\mathsf{Opening}})$ to mean simply that the union of the procedures from both protocols are used.

Proposition 1. *The protocol* $(\Pi_{\mathsf{MPC}} \| \Pi_{\mathsf{Opening}})$ *securely realises* $\mathcal{F}_{\mathsf{MPC}}$ *for a* $\mathcal{Q}_2$ *access structure in the presence of a computationally-bounded active adversary,* *corrupting any unqualified set of parties, in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, assuming the existence of a collision-resistant hash function and point-to-point secure channels.*

We note that since we do not use MAC values, we can also instantiate our protocol over small finite fields[4], or indeed using a LSSS over a ring. The latter will hold as long as the reconstruction vectors can be defined over the said ring. By taking a ring such as $\mathbb{Z}/2^{32}\mathbb{Z}$ we thus obtain a generalisation to an arbitrary $\mathcal{Q}_2$ structure of the Sharemind methodology [12]. Also note that we can extend $\mathcal{F}_{\mathsf{Prep}}$ in a trivial way so as to obtain other forms of pre-processing such shares of bits etc. as in [21].

6 Improved Offline Phase

In this section we give an offline subprotocol. Recall that the offline phase of MPC protocols involves the generation of so-called Beaver triples: triples of share vectors, $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ such that $\mathbf{c}$ encodes the product of the secrets encoded by $\mathbf{a}$ and $\mathbf{b}$. As is relatively standard practice for MPC protocols in the pre-processing model, we provide a semi-honest multiplication procedure, which is then bootstrapped to active security using the standard technique known as *sacrificing* to catch if errors were introduced.

The Beaver triples must be share-vectors with respect to the LSSS used in the online phase. Perhaps the most obvious methods of achieving this are the following:

- **Method 1** Use standard techniques to generate uniformly random shares (e.g. all parties act as the dealer to share a uniform secret, then the parties sum all n share vectors); then use Maurer's protocol to do the passive multiplication, and then sacrifice for active security.
- **Method 2** Use [31] to do all of the pre-processing, and then use the technique by Cramer et al. [17] to convert the replicated shares into shares under any LSSS computing the same access structure by local computations (after an inexpensive one-time set-up phase).
- **Method 3** Generalise the procedure from [31] to convert additive sharings to replicated.

We would expect the second method to be extremely efficient in many situations, since after setting up keys for generating random secrets efficiently, every party only needs to send $\Theta(n)$ field elements to different parties for each passive multiplication. The main problem is that the required number of PRF (pseudo-random function) keys grows linearly with the number of maximally unqualified sets. Indeed, for a $(84, 41)$-threshold access structure using the former method, the parties would need to agree on $\binom{84}{43} > 2^{80}$ keys. We note that the optimisation in [31], requiring PRF keys to generate some shares of a product deterministically, can be instantiated using keys already set up for

[4] If using a small ring/finite field we simply need to modify the sacrificing stage in the triple production process: no changes are needed for the online phase at all.

generating the PRSS shares. The computational cost is asymptotically approximately $O(2^n/\sqrt{n})$ PRF evaluations per party; nevertheless this computational cost reduces the communication cost, so there is a trade-off here.

In this section we propose an optimisation to the plain resharing protocol of Maurer, and use this to improve on pre-processing when the key-agreement becomes prohibitively expensive. For an ideal secret sharing, such as Shamir's, we obtain the same communication costs as in the optimised version of [31], but without additional computational costs. For general secret sharing schemes/access structures, the choice of whether replicated sharing and the protocol in [31], or using the method for general MSPs presented here, depends on the precise nature of the access structure and associated MSP.

Generating random secrets If we dispense with the PRF keys (as in **Method 1** and **Method 3**) then we are required to generate several random secrets for use as Beaver triples. This can done in different ways; for example, here are two different methods:

1. Take the largest maximally unqualified set, say of size t, take its complement, and let the parties in this set each generate a random share vector and distribute the shares. The parties then sum the share vectors. Correctness comes from the fact that this set of $n - t$ parties must contain at least one honest party, so the secret is always uniform. The cost is then the cost of sending $(n - t)$ share vectors per random secret.
2. Apply the "randomness extraction" method of [22] to obtain $n - t$ random sharings from n random sharings.

Generating PRZSs Pseudo-random zero-sharings (PRZSs) are additive sharings of zero, and are used to mask shares before sending them without changing the underlying secret. The functionality is given in Figure 8. We do not provide the protocol here as it is given in [31], 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

$$t_i := \sum_{j \neq i, j \in S} F_{\kappa_{i,j}}(\mathsf{count}) - F_{\kappa_{j,i}}(\mathsf{count}).$$

Our extension by the procedure **Rand** is possible if we assume the existence of random oracles (but we note that this assumption is required for the sacrifice step in any case).

6.1 Method 1: Bootstrap Maurer's Protocol

After obtaining shares of random secrets (as above), the parties do the following:

1. Perform local computations on shares to obtain an additive sharing of the product of the secrets.

2. Each party reshares its summand in the LSSS by acting as the dealer in the sharing protocol.
3. Each party sums the n sharings that result.
4. Use the sacrificing technique to check that the triples were generated correctly.

The chief advantage of this method is that it is computationally relatively lightweight. The local computation cost becomes crucially important when considering large numbers of parties due to the exponential blow-up of the total number of shares in replicated secret-sharing.

We include this method for doing the offline protocol as for very small numbers of parties it is likely to be as good as any other protocol, and for very large numbers of parties it does not need unreasonably many PRF keys.

6.2 Method 2: Use offline protocol of [31]

The basic structure of the [31] triple-generation protocol is:

1. Retrieve two replicated shares from memory and perform the (local) multiplication procedure to get an additive sharing of the product;
2. Convert the additive sharing to a replicated sharing.
3. Use the sacrificing technique to check that the triples were generated correctly.

Recall that an MSP is multiplicative if the parties can perform local computations to obtain an additive sharing of the product of two secrets, and that replicated secret-sharing is always multiplicative if the access structure is $\mathcal{Q}_2$.

The heart of the [31] protocol – the only part requiring communication – is simply an efficient method of turning an additive sharing back into a replicated sharing. This is achieved by finding a map f which maps shares to parties: then each party P_i is set to be in charge of some set of replicated shares S_i defined by f, so to convert from an additive share, P_i additively splits his share into $|S_i|$ pieces and treats them as replicated shares (after blinding with a pseudo-random zero-sharing – see Figure 8) that need to be distributed; these shares are then sent to the players who are supposed to hold them. This obviates the need for each player to reshare their additive share as proposed in **Method 1**.

The obvious way to obtain Beaver triples for any LSSS for any $\mathcal{Q}_2$ access structure, then, would be to use [31] to generate replicated shares and then convert using [17]. Note that unlike with **Method 1** above, the final LSSS *need not be multiplicative*. Curiously, then, this gives an online protocol for any $\mathcal{Q}_2$ access structure using *any* LSSS that realises it.

An optimisation to the plain [31] protocol is for each party to designate some $s^* \in S_i$ as the one which will be communicated, and for all other shares $s \in S_i \setminus \{s^*\}$ to be computed as $s := F_{\kappa_s}(\mathsf{count})$ where κ_s is a key known to all parties who are supposed to hold the share s and F is the chosen PRF keyed with κ_s; then party P_i with share x_i computes the special share as $s^* := x_i - \sum_{s \in S_i \setminus \{s^*\}} F_{\kappa_s}(\mathsf{count})$ and sends this to the parties who are supposed to hold share s^*. This significantly reduces the communication cost and requires no more PRF keys than are required to generate the random secrets for the triples in the first place.

6.3 Method 3: New Offline Protocol

This method principally involves an efficient method of converting an additively-shared secret to a sharing in the LSSS. The starting point is therefore any random secrets shared using a multiplicative LSSS, and sharings of random secrets may be obtained either using standard techniques as described above, or using pre-shared PRF keys.

Since we expect that [31] will outperform other methods when the number of parties is small, we will make the assumption that the secrets are merely shared using some multiplicative LSSS and not necessarily replicated sharings created using PRF evaluations as in **Method 2**.

That said, we note that if the parties start with a multiplicative LSSS realising the access structure, they can perform the local computations to obtain an additive sharing of the product, and then reshare this to any (not necessarily multiplicative) LSSS.

First we provide an algorithm in Figure 6 that, given an MSP provided by the user, transforms the the MSP into a new MSP which realises the same LSSS but has properties amenable to be used in an efficient share-conversion procedure. Then we provide a protocol Π_{Convert} in Figure 7, which shows how to use this MSP to convert additive shares into shares in the LSSS in a communication-efficient manner. One way of viewing our protocol is simply as an efficient computationally-secure resharing method optimised for resharing additive (i.e. full-threshold) sharings.

Correctness of Transformation Algorithm Any MSP can be converted to another equivalent MSP computing the same access structure by performing column operations on the matrix M [4]. This (potentially) changes the target vector, but share vectors in an LSSS are just vectors in the column-space of the generating matrix M, so in fact share vectors under the altered scheme are equally-well viewed as share vectors under the original scheme: one can think of the difference as choosing different randomness vectors to generate the share vector. Note also that column operations do not change the multiplicativity of the LSSS since it merely represents a change of basis for the space of share vectors[5].

In particular, it is always possible for the matrix M with d columns and rank d to be reduced via column operations so that all of the standard basis vectors

[5] In brief, the proof of this fact involves computing the pair-wise tensor of each row with each other row held by a given party and checking that the target vector tensored with itself is in the span of the resulting tensored vectors; the point is that by tensoring the matrices corresponding to the column operations and applying these to the tensored vectors will not change the rank of the span of the tensored rows.

$\{\mathbf{e}^k\}_{i=1}^d$ appear in the rows of M. The map χ is well-defined since we assume the matrix and target vector are non-zero.

Correctness of $\mathit{\Pi}_{\mathsf{Convert}}$ We discuss when the protocol aborts in detail in the proof of security, Appendix B; here we just consider correctness of the conversion procedure.

So far, for consistency with the literature, we have denoted by ψ the map from rows of M to parties in $\mathcal{P}$. For clean notation, we now define $\rho : [m] \to [n]$ by the following: for all $j \in [m]$, $\rho(j) := i$ if and only if $\psi(j) = P_i$. Thus $\psi(j) = P_{\rho(j)}$.

The goal is to reshare the additively-shared x, which we denote by $\langle x \rangle$ where party P_i holds x_i and $\sum_{i=1}^n x_i = x$, under the LSSS. To do this, the parties essentially distribute an additive sharing of a share-vector sharing the sum. In more detail, each party P_i first adds on its share t_i^0 of a PRZS $\langle t^0 \rangle$ – that is, a sharing where P_i holds t_i where $\sum_{i=1}^n t_i = 0$ – onto its share x_i and then samples a set $\{x_{i,k}\}_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}}$ where $K_i := \{\chi(i)\} \cup ([d] \setminus \mathrm{im}(\chi))$ and $S_{\boldsymbol{\varepsilon}}$ is the support of $\boldsymbol{\varepsilon}$, such that $x_i + t_i^0 = \sum_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}} x_{i,k}$. In other words, $x_i + t_i^0$ is reshared into as many pieces as there are unassigned columns, plus one for the column assigned by χ, and except where $\boldsymbol{\varepsilon}_k = 0$. Then for each $k \in K_i \cap S_{\boldsymbol{\varepsilon}}$, P_i chooses its contribution to the k^{th} co-ordinate of a randomness vector $\mathbf{x}$ to be $r_{i,j} := x_{i,k}/\boldsymbol{\varepsilon}_k$. For columns where $\boldsymbol{\varepsilon}_k = 0$ for some $k \in [d]$, each party will sample some randomness $r_{i,k} \leftarrow \mathbb{F}$ as their contribution to the co-ordinate $\mathbf{x}_k$ of the randomness vector $\mathbf{x}$. This is because this part of the randomness vector will be cancelled out upon reconstruction, and as such the parties cannot pack part of their summand x_i in this column. Note that the definition of χ precludes any party from being mapped to a column k where $\boldsymbol{\varepsilon}_k = 0$, so the columns where $\boldsymbol{\varepsilon}_k = 0$ necessarily lie in $[d] \setminus \mathrm{im}(\chi)$.

Defining $\mathbf{x}$ in this way means that overall (i.e. after all parties have done the above) we have

- $\mathbf{x}_k = \sum_{i \in \chi^{-1}(\{k\})} x_{i,k}/\boldsymbol{\varepsilon}_k$ for all $k \in \mathrm{im}(\chi)$;
- $\mathbf{x}_k = \sum_{i=1}^n x_{i,k}/\boldsymbol{\varepsilon}_k$ for $k \in [d] \setminus \mathrm{im}(\chi)$ where $\boldsymbol{\varepsilon}_k \neq 0$; and
- $\mathbf{x}_k = \sum_{i=1}^n r_{i,k}$ for $k \in [d] \setminus \mathrm{im}(\chi)$ where $\boldsymbol{\varepsilon}_k = 0$.

Hence the following:

$$\begin{aligned}
\langle \mathbf{x}, \boldsymbol{\varepsilon} \rangle &= \sum_{k \in \mathrm{im}(\chi)} \mathbf{x}_k \cdot \boldsymbol{\varepsilon}_k + \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \cap S_{\boldsymbol{\varepsilon}}} \mathbf{x}_k \cdot \boldsymbol{\varepsilon}_k + \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \setminus S_{\boldsymbol{\varepsilon}}} \mathbf{x}_k \cdot \boldsymbol{\varepsilon}_k \\
&= \sum_{k \in \mathrm{im}(\chi)} \sum_{i \in \chi^{-1}(\{k\})} (x_{i,k}/\boldsymbol{\varepsilon}_k) \cdot \boldsymbol{\varepsilon}_k \\
&\qquad + \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \cap S_{\boldsymbol{\varepsilon}}} \sum_{i=1}^n (x_{i,k}/\boldsymbol{\varepsilon}_k) \cdot \boldsymbol{\varepsilon}_k + \sum_{i=1}^n \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \setminus S_{\boldsymbol{\varepsilon}}} r_{i,k} \cdot 0 \\
&= \sum_{i=1}^n (x_{i,\chi(i)}/\boldsymbol{\varepsilon}_{\chi(i)}) \cdot \boldsymbol{\varepsilon}_{\chi(i)} + \sum_{i=1}^n \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \cap S_{\boldsymbol{\varepsilon}}} (x_{i,k}/\boldsymbol{\varepsilon}_k) \cdot \boldsymbol{\varepsilon}_k
\end{aligned}$$

$$= \sum_{i=1}^{n} \left(x_{i,\chi(i)} + \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \cap S_{\boldsymbol{\varepsilon}}} x_{i,k} \right) = \sum_{i=1}^{n} \sum_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}} x_{i,k} = \sum_{i=1}^{n} x_i = x,$$

i.e. the share vector where $\mathbf{x}$ is the randomness vector shares the secret $x = \sum_{i=1}^{n} x_i$.

We will now discuss in more detail how the shares of the share vector with randomness vector defined as above are distributed amongst the parties. Each share is then computed in one of two different ways.

- For each row of the matrix which is not a standard basis vector, the parties each compute

  $$a_i^j := \left(\sum_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}} M_j[k] \cdot r_{i,k} \right) + t_i^j$$

  where $\langle t_j \rangle$ is an n-party PRZS and in the protocol and above we defined $r_{i,k} := x_{i,k}/\boldsymbol{\varepsilon}_k$ for $k \in K_i \cap S_{\boldsymbol{\varepsilon}}$ and sampled $r_{i,k} \xleftarrow{\$} \mathbb{F}$ for $k \in K_i \setminus S_{\boldsymbol{\varepsilon}}$; the recipient simply sums all incoming share and adds its own contribution computed in the same way. Thus the party obtains:

  $$\begin{aligned}
  \sum_{i=1}^{n} a_i^j &= \sum_{i=1}^{n} \left(\left(\sum_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}} M_j[k] \cdot x_{i,k}/\boldsymbol{\varepsilon}_k \right) + \left(\sum_{k \in K_i \setminus S_{\boldsymbol{\varepsilon}}} M_j[k] \cdot r_{i,k} \right) + t_i^j \right) \\
  &= \sum_{i=1}^{n} \left(\left(\sum_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}} M_j[k] \cdot x_{i,k}/\boldsymbol{\varepsilon}_k \right) + \left(\sum_{k \in K_i \setminus S_{\boldsymbol{\varepsilon}}} M_j[k] \cdot r_{i,k} \right) \right) + \sum_{i=1}^{n} t_i^j \\
  &= \sum_{k \in \mathrm{im}(\chi)} \left(\sum_{i \in \chi^{-1}(\{k\})} M_j[k] \cdot x_{i,k}/\boldsymbol{\varepsilon}_k \right) \\
  &\qquad + \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \cap S_{\boldsymbol{\varepsilon}}} \left(\sum_{i=1}^{n} M_j[k] \cdot x_{i,k}/\boldsymbol{\varepsilon}_k \right) \\
  &\qquad\qquad + \sum_{k \in ([d] \setminus \mathrm{im}(\chi)) \setminus S_{\boldsymbol{\varepsilon}}} \left(\sum_{i=1}^{n} M_j[k] \cdot r_{i,k} \right) \\
  &= \sum_{k=1}^{d} M_j[k] \cdot \mathbf{x}_k = \langle M_j, \mathbf{x} \rangle.
  \end{aligned}$$

- For each row j which is a standard basis vector, the parties do the following: let k be the column index of the non-zero entry of the row; then the parties in P_i with $k \in K_i$ retrieve a PRZS amongst them and send their contribution $r_{i,k}$ to party $\psi(j)$:

  $$a_i^j := M_j[k] \cdot r_{i,k} + t_i^j$$

  where here $M_j[k] = 1$. Note that if it holds for some party P_i that $K_i \supseteq K_j$ for some $j \neq i$ then P_i will potentially be able to compute the value of $x_j + t_j^0$;

this is one reason for adding the n-party PRZS. The recipient again simply sums all values received.

The result of the protocol is that the parties have converted an additive sharing into a sharing in the LSSS with which they are working. The cost of doing so

6.4 Full Pre-processing Protocol

In Figure 9 we provide the full pre-processing protocol when using **Method 3**, which shows how Π_{Convert} interfaces with our more specialised opening protocol Π_{Opening}. For the sake of simplicity of exposition, we will assume sharings of random secrets are still generated using $\mathcal{F}_{\mathsf{Rand}}$ and are therefore replicated sharings, but we emphasise that the purpose of our protocol is to avoid using replicated secret-sharing in practice. (It makes almost no difference to the protocol description, since the only property of replicated sharing that is used is multiplicativity.) When it is necessary to produce random sharings for large numbers of parties, one of the methods outlined at the beginning of this section may be used. In our comparison of the costs in Section 6.5, we assume replicated secret-sharing is *not* used.

At a high level, Π_{Prep} generates Beaver triples in an actively-secure manner assuming black-box access to a functionality generating additive sharings of zero and replicated sharings of random secrets. Active security is obtained by doing a multiplication passively and then employing the standard technique of *sacrificing*. Extending the protocol to produce other forms of pre-processing (such as *shared bits* – see [21] for details) is trivial and therefore omitted. Recall we use $\Pi_A || \Pi_B$ to denote a protocol comprising the union of the procedures of Π_A and Π_B.

Proposition 2. *For a $\mathcal{Q}_2$ access structure, the protocol $(\Pi_{\mathsf{Prep}} || \Pi_{\mathsf{Opening}})$ (Figure 9) securely implements $\mathcal{F}_{\mathsf{Prep}}$ (Figure 3) in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model.*

6.5 Costs

To make some comparisons with the other protocols, we now consider the asymptotic costs involved in Maurer's protocol [32], the protocol of Keller et al. [31], and our protocol. The communication costs are summarised in Table 1. Note that we do *not* assume PRFs are used to generate random secrets for our protocol in our cost analysis, though our description of Π_{Prep} does to ease exposition: instead, we assume in Maurer's protocol and our own that each random secret is generated by all parties sampling a random secret and dealing shares as in the sharing protocol, and finally all parties summing the shares locally.

Notably, our protocol is as good or better than Maurer's protocol in all of the most expensive subroutines run in the pre-processing, and indeed the cost of setting up PRF keys and of performing authentication in our protocol is negligible considering the number of triples one typically needs to evaluate a reasonable circuit.

Our protocol is also considerably cheaper than [31] in these methods where even compared to the optimised version we replace a large binomial term with a linear term in the sacrifice step. This proves the importance of removing the dependence on replicated secret-sharing.

We note that while we have an $O(n^2)$ cost associated with generating random secrets in our protocol compared to [31], which arises from the fact that in our case the parties do not agree on $\binom{n}{t}$ PRF keys, our protocol is much more scalable. Indeed, the key set-up of [31] begins to become prohibitively expensive even with only 20 parties, where potentially $\binom{20}{9} > 2^{17}$ PRF keys need to be agreed upon in the honest majority setting.

An additional goal of [31] was to reduce the number of communication channels required to perform MPC. In Table 2 we compare the number of channels used in each protocol at different points. Our protocol uses a smaller number of channels than Maurer's protocol [32] and uses the same as [31] in the threshold case for most subroutines.

Protocol	[32]	[31]	Optimised [31]	Ours
Key set-up	0	$\kappa n\left((n-1)+\binom{n-1}{t}\right)$	$\kappa n\left((n-1)+\binom{n-1}{t}\right)$	$\kappa n(n-1)$
Random secret gen.	$n(n-1)$	0	0	$n(n-1)$
Passive mult.	$n(n-1)$	$\binom{n}{t}\cdot(n-t-1)$	$n(n-t-1)$	$n(n-t-1)$
Sacrifice	$3n(n-1)$	$3\binom{n}{t}\cdot t$	$3\binom{n}{t}\cdot t$	$3nt$
Authentication	0	$256n(n-1)$	$256n(n-1)$	$256n(n-1)$

Table 1. A comparison of communication cost given in number of bits. The rows **Key set-up** and **Authentication** are the number of bits, the rest are the number of field elements. We assume the hash function is SHA256 to give 128 bits of security against collision by the birthday bound; κ is the computational security parameter and is the key-length of the PRF. In [32]'s and our protocol we use Shamir's secret-sharing whereas [31] (necessarily) uses the standard replicated MSP. Note that while our protocol comparatively little key set-up, the sharings of random secrets used in Beaver triples are more expensive to generate in terms of communication.

Protocol	[32]	[31]	Optimised [31]	Ours
Key set-up	0	$n(n-1)$	$n(n-1)$	$n(n-1)$
Random secret gen.	$n(n-1)$	0	0	$n(n-1)$
Passive mult.	$n(n-1)$	$n(n-t-1)$	$n(n-t-1)$	$n(n-t-1)$
Sacrifice	$n(n-1)$	nt	nt	nt
Authentication	0	$n(n-1)$	$n(n-1)$	$n(n-1)$

Table 2. A comparison of the number of uni-directional channels required for each subroutine in our protocol Π_{Prep}

Acknowledgements

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 No. N66001-15-C-4070, and by EPSRC via grant EP/N021940/1.

References

1. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16: 23rd Conference on Computer and Communications Security. pp. 805–817. ACM Press, Vienna, Austria (Oct 24–28, 2016)
2. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) Advances in Cryptology – CRYPTO'91. Lecture Notes in Computer Science, vol. 576, pp. 420–432. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 11–15, 1992)
3. Beaver, D., Wool, A.: Quorum-based secure multi-party computation. In: Nyberg, K. (ed.) Advances in Cryptology – EUROCRYPT'98. Lecture Notes in Computer Science, vol. 1403, pp. 375–390. Springer, Heidelberg, Germany, Espoo, Finland (May 31 – Jun 4, 1998)
4. Beimel, A., Gál, A., Paterson, M.: Lower bounds for monotone span programs. In: 36th Annual Symposium on Foundations of Computer Science. pp. 674–681. IEEE Computer Society Press, Milwaukee, Wisconsin (Oct 23–25, 1995)
5. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: 20th Annual ACM Symposium on Theory of Computing. pp. 1–10. ACM Press, Chicago, IL, USA (May 2–4, 1988)
6. Ben-Sasson, E., Fehr, S., Ostrovsky, R.: Near-linear unconditionally-secure multiparty computation with a dishonest minority. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 663–680. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012)
7. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. Lecture Notes in Computer Science, vol. 6632, pp. 169–188. Springer, Heidelberg, Germany, Tallinn, Estonia (May 15–19, 2011)
8. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 326–343. Springer, Heidelberg, Germany, Kaoshiung, Taiwan, R.O.C. (Dec 7–11, 2014)
9. Blakley, G.R.: Safeguarding cryptographic keys. Proceedings of AFIPS 1979 National Computer Conference 48, 313–317 (1979)
10. Blakley, G.R., Meadows, C.: Security of ramp schemes. In: Blakley, G.R., Chaum, D. (eds.) Advances in Cryptology – CRYPTO'84. Lecture Notes in Computer Science, vol. 196, pp. 242–268. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 1984)

11. Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., Talviste, R.: Students and taxes: a privacy-preserving social study using secure computation. Cryptology ePrint Archive, Report 2015/1159 (2015), http://eprint.iacr.org/2015/1159
12. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) ESORICS 2008: 13th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer, Heidelberg, Germany, Málaga, Spain (Oct 6–8, 2008)
13. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009: 13th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 5628, pp. 325–343. Springer, Heidelberg, Germany, Accra Beach, Barbados (Feb 23–26, 2009)
14. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology 13(1), 143–202 (2000)
15. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000), http://eprint.iacr.org/2000/067
16. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: 20th Annual ACM Symposium on Theory of Computing. pp. 11–19. ACM Press, Chicago, IL, USA (May 2–4, 1988)
17. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC 2005: 2nd Theory of Cryptography Conference. Lecture Notes in Computer Science, vol. 3378, pp. 342–362. Springer, Heidelberg, Germany, Cambridge, MA, USA (Feb 10–12, 2005)
18. Cramer, R., Damgård, I., Maurer, U.M.: General secure multi-party computation from any linear secret-sharing scheme. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000. Lecture Notes in Computer Science, vol. 1807, pp. 316–334. Springer, Heidelberg, Germany, Bruges, Belgium (May 14–18, 2000)
19. Cramer, R., Daza, V., Gracia, I., Urroz, J.J., Leander, G., Martí-Farré, J., Padró, C.: On codes, matroids and secure multi-party computation from linear secret sharing schemes. In: Shoup, V. (ed.) Advances in Cryptology – CRYPTO 2005. Lecture Notes in Computer Science, vol. 3621, pp. 327–343. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 14–18, 2005)
20. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography. Lecture Notes in Computer Science, vol. 5443, pp. 160–179. Springer, Heidelberg, Germany, Irvine, CA, USA (Mar 18–20, 2009)
21. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013: 18th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 8134, pp. 1–18. Springer, Heidelberg, Germany, Egham, UK (Sep 9–13, 2013)
22. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) Advances in Cryptology – CRYPTO 2007. Lecture Notes in Computer Science, vol. 4622, pp. 572–590. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2007)

23. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012)
24. van Dijk, M.: Secret Key Sharing and Secret Key Generation. Ph.D. thesis, Eindhoven University of Technology (1997)
25. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Coron, J., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017, Part II. Lecture Notes in Computer Science, vol. 10211, pp. 225–255. Springer, Heidelberg, Germany, Paris, France (Apr 30 – May 4, 2017)
26. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th Annual ACM Symposium on Theory of Computing. pp. 218–229. ACM Press, New York City, NY, USA (May 25–27, 1987)
27. Hirt, M., Maurer, U.M.: Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In: Burns, J.E., Attiya, H. (eds.) 16th ACM Symposium Annual on Principles of Distributed Computing. pp. 25–34. Association for Computing Machinery, Santa Barbara, CA, USA (Aug 21–24, 1997)
28. Ito, M., Saito, A., Nishizeki, T.: Secret sharing schemes realizing general access structure. In: Proc. IEEE Global Telecommunication Conf. (Globecom'87). pp. 99–102 (1987)
29. Karchmer, M., Wigderson, A.: On span programs. In: Proceedings of Structures in Complexity Theory. pp. 102–111 (1993)
30. Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16: 23rd Conference on Computer and Communications Security. pp. 830–842. ACM Press, Vienna, Austria (Oct 24–28, 2016)
31. Keller, M., Rotaru, D., P. Smart, N., Wood, T.: Reducing communication channels in MPC. In: International Conference on Security in Communication Networks, 11th International Conference, SCN 2018, Amalfi, Italy, September 57, 2018, Proceedings. pp. 181–199 (01 2018)
32. Maurer, U.M.: Secure multi-party computation made simple. Discrete Applied Mathematics 154(2), 370–381 (2006), http://dx.doi.org/10.1016/j.dam.2005.03.020
33. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of nonlinear functions in the presence of glitches. Journal of Cryptology 24(2), 292–321 (Apr 2011)
34. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Gennaro, R., Robshaw, M.J.B. (eds.) Advances in Cryptology – CRYPTO 2015, Part I. Lecture Notes in Computer Science, vol. 9215, pp. 764–783. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 2015)
35. Shamir, A.: How to share a secret. Communications of the Association for Computing Machinery 22(11), 612–613 (Nov 1979)
36. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th Annual Symposium on Foundations of Computer Science. pp. 162–167. IEEE Computer Society Press, Toronto, Ontario, Canada (Oct 27–29, 1986)

Protocol $\mathit{\Pi}_{\mathsf{MPC}}$

Note that this protocol calls on procedures from $\mathit{\Pi}_{\mathsf{Opening}}$ in Figure 2. If a party never receives an expected message from the adversary, we assume the receiving party signals Abort to all other parties and aborts.

Initialise: The parties do the following:

1. Each $P_i \in \mathcal{P}$ executes $H^i.\mathsf{Initialise}()$.
2. The parties call $\mathcal{F}_{\mathsf{Prep}}$ with input (Triple, N_T) get N_T triples.
3. The parties agree on a public sharing of the secret 1, denoted by $\mathbf{u}$.
4. Each party has one random secret opened to them for every input they will provide to the protocol: the parties do the following:
   (a) Retrieve from memory a sharing $\mathbf{r}$ of a uniformly random secret r, obtained first or second random secret from a Beaver triple. (The secret used may neither be used again for input nor used in a multiplication.)
   (b) Run **OpenTo**(i) on $\mathbf{r}$ so that P_i obtains r.

Input: For party P_i to input secret s,

1. Party P_i retrieves a secret r from memory, corresponding to a share vector $\mathbf{r}$ established during **Initialise** for inputs, and all parties $P_j \in \mathcal{P}$ retrieve their shares $\mathbf{r}_{\{P_j\}}$.
2. Party P_i executes **Broadcast** to open $\epsilon := s - r$.
3. Each party $P_j \in \mathcal{P}$ computes $\mathbf{s}_{\{P_j\}} := \epsilon \cdot \mathbf{u}_{\{P_j\}} + \mathbf{r}_{\{P_j\}}$.

Add: To add secrets s and s', with corresponding share vectors $\mathbf{s}$ and $\mathbf{s}'$, for each $P_i \in \mathcal{P}$ party P_i computes $\mathbf{s}_{\{P_i\}} + \mathbf{s}'_{\{P_i\}}$.

Multiply: To multiply secrets s and s', with corresponding share vectors $\mathbf{s}$ and $\mathbf{s}'$, each $P_i \in \mathcal{P}$ does the following:

1. Retrieve from memory the shares $(\mathbf{a}_{\{P_i\}}, \mathbf{b}_{\{P_i\}}, \mathbf{c}_{\{P_i\}})$ of a triple $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ obtained in **Initialise**.
2. Compute $\mathbf{s}_{\{P_i\}} - \mathbf{a}_{\{P_i\}}$ and $\mathbf{s}'_{\{P_i\}} - \mathbf{b}_{\{P_i\}}$.
3. Run **OpenTo**(0) on $\mathbf{s} - \mathbf{a}$ and $\mathbf{s}' - \mathbf{b}$ to obtain (publicly) $s - a$ and $s' - b$.
4. If the parties have not aborted, compute the following as the share of the product $\mathbf{c}_{\{P_i\}} + (s-a) \cdot \mathbf{s}'_{\{P_i\}} + (s'-b) \cdot \mathbf{s}_{\{P_i\}} - (s-a) \cdot (s'-b) \cdot \mathbf{u}_{\{P_i\}}$.

OutputTo(i): If $i = 0$, the secret s, encoded via share vector $\mathbf{s}$, is to be output to all parties, so the parties do the following:

1. Run **Verify**.
2. If the parties have not aborted, run **OpenTo**(0) on s.
3. If the parties have not aborted, run **Verify** again.
4. If the parties have not aborted, all parties (locally) output s.

If $P_i \in \mathcal{P}$, the secret s encoded via share vector $\mathbf{s}$ is to be output to party P_i, so the parties do the following:

1. Run **Verify**.
2. If the parties have not aborted, run **OpenTo**(i) on s.
3. If P_i has not aborted it (locally) outputs s.

Figure 5. Protocol $\mathit{\Pi}_{\mathsf{MPC}}$

Algorithm for computing a "good" MSP

The input is the multiplicative MSP $\mathcal{M} = (\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$, where we assume M has d columns and rank d. The output is a map $\chi : [n] \to [d]$ of party indices to columns, and a new MSP.

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

Figure 6. Algorithm for computing a "good" MSP

Subprotocol 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 current MSP $(\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ (output by the conversion algorithm).

1. The parties call $\mathcal{F}_{\mathsf{Rand}}$ with the command $(\mathsf{PRZS}, \mathsf{count}, \mathcal{P})$ to obtain a PRZS amongst them, denoted hereafter by $\langle t^0 \rangle$.
2. Each P_i splits $x_i + t_i^0$ as $x_i + t_i^0 = \sum_{k \in K_i \cap S_{\boldsymbol{\varepsilon}}} x_{i,k}$ where $K_i := (\{(\chi(P_i)\} \cup ([d] \setminus \mathrm{im}(\chi)))$ and $S_{\boldsymbol{\varepsilon}}$ is the support of $\boldsymbol{\varepsilon}$.
3. Each P_i sets $r_{i,k} := x_{i,k} / \boldsymbol{\varepsilon}_k$ for each $k \in K_i \cap S_{\boldsymbol{\varepsilon}}$.
4. Each P_i samples $r_{i,k} \leftarrow \mathbb{F}$ for each $k \in K_i \setminus S_{\boldsymbol{\varepsilon}}$.
5. For each row j which is *not* a standard basis vector, the parties do the following
   (a) The parties call $\mathcal{F}_{\mathsf{Rand}}$ with the command $(\mathsf{PRZS}, \mathsf{count}, \mathcal{P})$ to obtain a PRZS amongst them, denoted hereafter by $\langle t^j \rangle$.
   (b) Each P_i computes
   $$a_i^j := \left(\sum_{k=1}^{d} M_j[k] \cdot r_{i,k} \right) + t_i^j,$$
   where $M_j[k]$ denotes the k^{th} element of row j.
   (c) Party P_i sends a_i^j to party $\psi(j)$.
   (d) Party $\psi(j)$ computes $\mathbf{s}_j := \sum_{i=1}^{n} a_i^j$.
6. For each row j which *is* a standard basis vector: let k be the column in which the vector is non-zero; then the parties do the following:
   (a) Let $X_i = \{P_i \in \mathcal{P} : K_i \ni k\}$; then parties in X_i call $\mathcal{F}_{\mathsf{Rand}}$ with the command $(\mathsf{PRZS}, \mathsf{count}, X_i)$ to obtain a PRZS $\langle t^j \rangle$.
   (b) Each party $P_i \in X_i$ computes
   $$a_i^j := M_j[k] \cdot r_{i,k} + t_i^j,$$
   and then sends a_i^j to party $\psi(j)$ (or retains it if $\psi(j) = P_i$). (Note that we always have $M_j[k] = 1$ in this case.)
   (c) Party $\psi(j)$ sets $\mathbf{s}_j := \sum_{i : P_i \in X_i} a_i^j$

Figure 7. Subprotocol Π_{Convert} converting additive shares to shares in the LSSS

The Functionality $\mathcal{F}_{\mathsf{Rand}}$

This functionality outputs the same random field element to all parties, pseudo-random additive sharings of zero and pseudo-random replicated shares of random field elements, depending on what the parties input. The set $\mathcal{B}$ is the set of complements of sets in Δ^+, the set of maximally unqualified sets, which is used in replicated secret-sharing: see Figure 11 for more details.

PRZS:

- On input $(\mathsf{PRZS}, \mathsf{count}, S)$ from all parties in a set $S \subseteq \mathcal{P}$, 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 $t_i \xleftarrow{\$} \mathbb{F}$ uniformly at random, fixes $t_{i^*} := -\sum_{i \in S \setminus \{P_{i^*}\}} t_i$ and sends t_i to party P_i for each $i \in S \setminus \{P_{i^*}\}$ and t_{i^*} to P_{i^*}.

PRSS:

- On input $(\mathsf{PRSS}, \mathsf{count})$ from all parties, if the counter value is the same for all parties and has not been used before, the functionality samples a set $\{r_B\}_{B \in \mathcal{B}} \xleftarrow{\$} \mathbb{F}$ and for each $B \in \mathcal{B}$ sends r_B to all $i \in B$.

Rand

- On input $(\mathsf{Rand}, \mathsf{count})$ from all parties, if the counter value is the same for all parties and has not been used before, the functionality samples $r \xleftarrow{\$} \mathbb{F}$ and sends it to all parties.

Figure 8. The Functionality $\mathcal{F}_{\mathsf{Rand}}$

Protocol Π_{Prep}

Pre-processing for any LSSS computing a $\mathcal{Q}_2$ access structure. Note that it takes *replicated* shares from $\mathcal{F}_{\mathsf{Rand}}$, computes the additive sharing of the product, then uses Π_{Convert} to convert this to the *specific LSSS* sharing. Consequently, the LSSS need not be multiplicative. It takes in a parameter N_T for the number of triples, and then the parties do the following:

1. Initialise a global counter count and agree on some sharing $\mathbf{u}$ of the value 1.
2. For i from 1 to N_T, do the following:
   (a) Call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\mathsf{PRSS}, \mathsf{count})$ four times, incrementing count after each call, to obtain secrets a^i, b^i, x^i and y^i shared as replicated shares; i.e. $\mathbf{a}^i$, $\mathbf{b}^i$, $\mathbf{x}^i$ and $\mathbf{y}^i$.
   (b) Each party performs local computations on its shares so that together they obtain an additive sharing of the product of a^i and b^i, and then do similarly with x^i and y^i.
   (c) Run the subprotocol Π_{Convert} to obtain a sharing $\mathbf{c}^i$ of the secret $a^i \cdot b^i$ and a sharing $\mathbf{z}^i$ of the secret $x^i \cdot y^i$ (both sharings in the LSSS).
   (d) Using the method of [17], locally convert $\mathbf{a}^i$ and $\mathbf{b}^i$ from replicated shares to shares in this LSSS.
3. Now sacrifice: for i from 1 to N_T, do the following:
   (a) Call $\mathcal{F}_{\mathsf{Rand}}$ with input $(\mathsf{Rand}, \mathsf{count})$ to obtain r^i, increment count, and run $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ on $r^i\mathbf{x}^i - \mathbf{a}^i$ and $\mathbf{y}^i - \mathbf{b}^i$.
   (b) Locally compute
   $$\mathbf{t}^i := r^i\mathbf{z}^i - (y^i - b^i)\mathbf{a}^i - (r^i x^i - a^i)\mathbf{b}^i - \mathbf{c}^i - (r^i x^i - a^i)(y^i - b^i)\mathbf{u}.$$
   (c) Run $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ on $\mathbf{t}^i$ to obtain t^i.
4. Run $\Pi_{\mathsf{Opening}}.\mathbf{Verify}$.
5. If $t^i = 0$ for all i, locally output the triples $(\mathbf{a}^i, \mathbf{b}^i, \mathbf{c}^i)_{i=1}^{N_T}$.

Figure 9. Protocol Π_{Prep}

A Standard Linear Secret-Sharing Schemes

In order to aid the reader, in this appendix we outline some examples of standard Linear Secret Sharing Schemes that we refer to in the main text. For each, we point out how the schemes relate to the notions in our paper in terms of error-detection. The three schemes we select are Shamir, Replicated and DNF-based sharing; the latter is sometimes referred to as ISN sharing. Replicated and DNF-based sharing can be derived from the conjunctive and disjunctive normal forms of the Boolean formulae describing their access structures, respectively. Every $\mathsf{OR}(A, B)$ in the formulae denotes that the parties in A and B get the current share; and every $\mathsf{AND}(A, B)$ denotes that the current share is additively shared between A and B. Proofs of correctness are omitted: the interested reader may refer to [28, 35].

A.1 Shamir Sharing

We now give examples of Shamir sharing to make ideas more concrete.

Shamir Sharing $\mathit{\Pi}_{\mathsf{Shamir}}$

The access structure is (n, t)-threshold.

Input: For party P_i to provide input s, it does the following:

1. Sample an irreducible polynomial $f \xleftarrow{\$} \mathbb{F}[X]$ of degree t subject to $f(0) = s$.
2. For each $P_j \in \mathcal{P} \setminus \{P_i\}$, give party P_i the value $f(i)$.

Open: For a qualified set of parties Q to open a secret s,

1. Each party $P_i \in Q$ broadcasts their share $f(i)$ to all other parties in Q.
2. Each party uses Lagrange interpolation to compute $f(0)$.

ALF: To compute an affine function $L(s^1, \dots, s^k) + a$ where L is a linear function, a a public constant, and $\{s^1, \dots, s^k\}$ a set of secrets, each party computes $L(f_{s^1}(i), \dots, f_{s^k}(i)) + a$.

Figure 10. Shamir Sharing $\mathit{\Pi}_{\mathsf{Shamir}}$

Shamir, $(3, 1)$-threshold Consider Shamir's secret-sharing scheme for a $(3, 1)$-threshold access structure and for simplicity assume that P_i receives $f(i)$. The MSP matrix is a Vandermonde matrix. We write it below with the row-map ψ on the left:

$$M = \begin{array}{c} P_1 \\ P_2 \\ P_3 \end{array} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix}$$

The target vector in this case is $\boldsymbol{\varepsilon} = \mathbf{e}^1 = (1, 0)^\top \in \mathbb{F}^2$. Let N be the cokernel of M, for some choice of basis; one possible choice is:

$$N = \begin{pmatrix} 1 & -2 & 1 \end{pmatrix}.$$

There are no share vectors with unqualified support, since such a share vector corresponds to a polynomial of degree t with $t+1$ zeros, which cannot exist. Hence, by linearity, the adversary cannot change the share vector so that the resulting vector is still a valid share vector. This is consistent with our lemmata since the access structure is $\mathcal{Q}_2$.

Shamir, $(4, 1)$-threshold Now we do the same for $(4, 1)$-threshold. We have

$$M = \begin{array}{c} P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix}$$

and a possible parity-check matrix

$$N = \begin{pmatrix} 1 & 0 & -3 & 2 \\ 0 & 1 & -2 & 1 \end{pmatrix}.$$

A.2 Replicated Sharing

Again we give some basic examples.

Replicated Sharing $\Pi_{\mathsf{Replicated}}$

Let $\mathcal{B} := \{B \in 2^{\mathcal{P}} : \mathcal{P} \setminus B \in \Delta^+\}$.

Input: For party P_i to provide input s, it does the following:

1. Sample a set $\{s_B\}_{B \in \mathcal{B}} \xleftarrow{\$} \mathbb{F}$ subject to $\sum_{B \in \mathcal{B}} s_B = s$.
2. For each $B \in \mathcal{B}$, give s_B to all $i \in B$.

Open: For a qualified set of parties Q to open a secret s,

1. A qualified set Q of parties together has all shares, i.e. the set $\{s_B\}_{B \in \mathcal{B}}$. For each $B \in \mathcal{B}$, if $P_j \in Q$ and $P_j \notin B$ then all parties $P_i \in B$ send s_B to party P_j.
2. Each party computes $s = \sum_{B \in \mathcal{B}} s_B$.

ALF: To compute an affine linear function $L(s^1, \ldots, s^k) + a$ where L is a linear function, a a public constant, and $\{s^1, \ldots, s^k\}$ a set of secrets, each $P_i \in \mathcal{P}$, for each $B \in \mathcal{B}$ where $B \ni P_i$, computes the output as $s_B := L(s^1_B, \ldots, s^k_B)$, and then for some $B \in \mathcal{B}$, every $P_i \in B$ sets the share s_B as $s_B := L(s^1_B, \ldots, s^k_B) + a$.

Figure 11. Replicated Sharing $\Pi_{\mathsf{Replicated}}$

Replicated, (3, 1)-threshold Consider the replicated secret-sharing for the (3, 1)-threshold access structure. One choice of MSP matrix is:

$$M = \begin{array}{c} P_1 \\ P_2 \\ P_2 \\ P_3 \\ P_3 \\ P_1 \end{array} \begin{pmatrix} 1 & -1 & -1 \\ 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

where the numbers to the left of the matrix indicate which party owns each row (i.e. they indicate the map ψ). The target vector is $\boldsymbol{\varepsilon} = \mathbf{e}^1 = (1, 0, 0)^\top \in \mathbb{F}^3$. Let N be the cokernel of M, for some choice of basis; one possible choice is:

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

As with Shamir's secret-sharing, there are no share vectors with unqualified support: if $N \cdot \mathbf{t} = \mathbf{0}$, then $\mathbf{t}$ is of the form $(\mathbf{t}_1, \mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_2, \mathbf{t}_3, \mathbf{t}_3)^\top$ where $\mathbf{t}_i \in \mathbb{F}$; however, the image under ψ of the support of an error with this form (for which the encoded secret is non-zero) is necessarily qualified (e.g. $\mathbf{t} = (0, 0, 0, 0, 1, 1)^\top$ has support $\{1, 3\}$, which is qualified). Thus the adversary cannot change the share vector so that the new shares encode the same secret, let alone changing the share so that it shares a different secret, without the resulting vector not being in $\text{im}(M)$.

A.3 DNF-based Sharing

Here we choose a little more exotic an example. Consider the $\mathcal{Q}_2$ access structure given by (where we are just writing party indices, for clarity):

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

$$\mathit{\Delta}^+ = \{\{1\}, \{2\}, \{3\}\}.$$

One choice of MSP matrix is:

$$M = \begin{array}{c} P_4 \\ P_1 \\ P_2 \\ P_1 \\ P_3 \\ P_2 \\ P_3 \end{array} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

DNF Sharing $\varPi_{\mathsf{DNF}}$

Input: For party P_i to provide input s, it does the following:

1. For each $Q \in \varGamma^-$, sample $\{s_Q^j\}_{P_j \in Q} \xleftarrow{\$} \mathbb{F}$ subject to $\sum_{P_j \in Q} s_Q^j = s$.
2. For each $Q \in \varGamma^-$, for each $P_j \in Q$, give s_Q^j to P_j.

Open: For a qualified set of parties Q to open a secret s,

1. A qualified set Q of parties has all shares in the set $\{s_Q^i\}_{P_i \in Q}$. Each party $P_i \in Q$ broadcasts their share s_Q^i to all other parties in Q.
2. Each party in Q computes $s = \sum_{P_i \in Q} s_Q^i$.

ALF: For notational simplicity, we describe how to compute the affine linear function $L(x, y) = \alpha x + \beta y + \gamma$ where $\alpha, \beta \in \mathbb{F}$ define a linear function on secrets x and y, and $\gamma \in \mathbb{F}$ is a public constant.
Each $P_i \in \mathcal{P}$, for each $Q \in \varGamma^+$ where $Q \ni P_i$, computes $z_Q^i := \alpha x_Q^i + \beta y_Q^i$. The public constant γ must be added to each of the $|\varGamma^-|$ sharings, so for each $Q \in \varGamma^-$ the parties decide on some $P_i \in Q$ to modify z_Q^i to be $z_Q^i := z_Q^i + \gamma$.

Figure 12. DNF Sharing $\varPi_{\mathsf{DNF}}$

The target vector in this case is $\boldsymbol{\varepsilon} = \mathbf{e}^1 = (1, 0, 0, 0)^\top \in \mathbb{F}^4$. Let N be the cokernel of M, for some choice of basis; one possible choice is:

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

As with the $\mathcal{Q}_2$ Shamir and Replicated secret sharing examples, there are no share vectors with unqualified support. This is because any share vector must contain the secret in the first component, which automatically makes its support qualified since $\psi(1) = \{P_4\}$, which is qualified.

B Proofs

B.1 Proof of Proposition 1

Following standard proofs in the UC model, we provide a simulator in Figure 13, Figure 14 and Figure 15 and will argue that no environment, which provides input and output to all parties, and additionally decides on all action and views all internal behaviour of the adversary, can distinguish between the world in which the simulator provides a transcript running the adversary as a subroutine and interacting with $\mathcal{F}_{\mathsf{MPC}}$ given in Figure 4, and a hybrid world in which the honest parties and adversary instead interact in a protocol execution given in Figure 2 and Figure 5 where in this hybrid world the existence of $\mathcal{F}_{\mathsf{Prep}}$ given in Figure 3 is assumed. Before we give the formal indistinguishability argument, we make a couple of remarks on the proof and simulation.

Correctness of Simulation The majority of the simulation merely involves running the protocol honestly with the adversary, extracting inputs, and faking inputs of honest parties.

Initialise: Note that $\mathcal{F}_{\mathsf{MPC}}$ must even allow the adversary to abort during the initialisation since the $\mathcal{F}_{\mathsf{Prep}}$ protocol allows the adversary to abort. Agreement of the sharing of 1 can be done in different ways and is not particularly interesting for our protocol; for example, one party can generate a random sharing of 1 and broadcast it; the parties can then update the hash function with the sharing they observed to ensure (after running Π_{MPC}.**Verify** later on) that all parties receive the same vector. It is for this reason that the details are omitted from the protocol description.

When opening the random mask $\mathbf{r}$ which will be an input mask, the functionality must allow a message from the adversary to abort or continue because in the simulation the adversary can cause an abort during Π_{Opening}.**OpenTo**(i). Recall, however, that the adversary can either cause an abort, or the party providing input necessarily computes the correct mask r, by Lemma 1; in particular, then, the adversary cannot cause the (honest) party to use an incorrect value instead of r. Note also that if the party providing input is corrupt and sends different values of ϵ to different honest players (emulated by the simulator), the flag `Abort` is *not* set as true because the protocol is supposed to continue; however, the hashes are computed by the simulator exactly as they would be in the real world, and hence the protocol will abort if the adversary behaves in this manner.

LSSSs admitting non-trivial sharings of 0 We highlight this part of the proof since it is important for understanding why simulation is possible even though for some LSSSs the adversary can cause different parties to reconstruct different share vectors which share the same secret.

Note that if the LSSS admits non-trivial sharings of the secret 0, the adversary could open a secret to different honest parties by making them reconstruct *different* share vectors, even though the secret must be the same because of the inherent error-detection in the MSP. However, the definition of q in Section 4 disallows this since it requires that $\ker(M_{\mathsf{q}(P_i)}) = \{\mathbf{0}\}$ for each i. In such cases, the size of the set $\mathsf{q}(P_i)$ is likely to be large, if not the whole set of shares.

Indistinguishability

Proof. Following standard practice, we provide a sequence of hybrid executions between the real-world and ideal-world executions, prove that each hybrid is indistinguishable from the next, and conclude that the first and last executions are indistinguishable.

Hybrid 0: This is the hybrid world in which the adversary and simulator run the protocol and the simulator additionally responds to all calls to $\mathcal{F}_{\mathsf{Prep}}$. Here the simulator uses the actual inputs of honest parties. Using knowledge of the mask from internally running $\mathcal{F}_{\mathsf{Rand}}$, the simulator extracts input of the adversary from the broadcast and passes it to $\mathcal{F}_{\mathsf{MPC}}$, which interacts with honest parties with their inputs from the environment. Whatever the behaviour of the adversary, the simulator continues the protocol honestly, and sends the message to $\mathcal{F}_{\mathsf{MPC}}$ to abort if an honest party or the adversary in the protocol would signal abort.

Hybrid 1: Same as **Hybrid 0**, but the simulator additionally stores errors for each sharing, as induced by the adversary during calls to Π_{Opening}.**Broadcast** and Π_{Opening}.**OpenTo**(). Then, during calls to Π_{MPC}.**Add**, the simulator add errors in the obvious way, and in Π_{MPC}.**Multiply** replace opening of vectors $\mathbf{s}-\mathbf{a}$ and $\mathbf{s}'-\mathbf{b}$ in **Multiply** with share vectors $\boldsymbol{\rho}+\boldsymbol{\delta}^{(s)}$ and $\boldsymbol{\sigma}+\boldsymbol{\delta}^{(s')}$ sharing uniform secrets, but which agree with the shares held by the adversary on $\mathbf{s}_A - \mathbf{a}_A$ and $\mathbf{s}'_A - \mathbf{b}_A$ respectively. See the appropriate procedures in $\mathcal{S}_{\mathsf{MPC}}$ in Figure 13 for details.

Hybrid 2: Same as **Hybrid 1** but the simulator replaces the broadcast of $\epsilon = s-r$ for honest parties' actual inputs s, and r from $\mathcal{F}_{\mathsf{Prep}}$, with uniformly randomly sampled inputs instead (or equivalently sample and broadcast a uniform ϵ). Note that in the previous hybrid we removed the dependence on the inputs of honest parties of the transcript during Π_{MPC}.**Multiply**. The only remaining place to remove the dependence is during output. To do this, the simulator stores all share vectors output from $\mathcal{F}_{\mathsf{Prep}}$, and executes the entire remainder of Π_{MPC} (after the alterations in **Input**) just as honest parties would, except for output, in which the simulator generates new shares derived from the output of the functionality $\mathcal{F}_{\mathsf{MPC}}$ and the internally-run $\mathcal{F}_{\mathsf{Prep}}$ and passes these to the adversary during the call to **OutputTo** (detailed in Figure 15). The simulator sends the message to $\mathcal{F}_{\mathsf{MPC}}$ to abort if an honest party or the adversary in the protocol would signal abort.

Hybrid 0 → ***Hybrid 1*** **Hybrid 0** produces the same transcript for the adversary as would be generated in the protocol execution since at this point we give the simulator the actual inputs of honest parties, and the simulator uses the honest parties' inputs to perform the protocol honestly. Observe that any errors induced on share vectors by acting maliciously in Π_{Opening}.**Broadcast** or Π_{Opening}.**OpenTo** will (possibly) be observed by the adversary in the procedure Π_{MPC}.**Multiply**, since the honest parties will not necessarily have aborted by this point; of course, the protocol will still abort in Π_{MPC}.**OutputTo**, but the simulation of the protocol must continue until this is called, since honest parties in the real (hybrid) world would also continue. Moreover, note that Π_{MPC}.**Multiply** is the *only* place errors are (possibly) observable by the adversary. Therefore, to replace the secrets opened in Π_{MPC}.**Multiply** with uniform secrets, which we want to do as then they are trivially independent of the honest parties' inputs, the simulator must keep track of the errors introduced by the adversary throughout and ensure those errors are observed during the calls to

$\mathit{\Pi}_{\mathsf{Opening}}$.**OpenTo** of $\mathit{\Pi}_{\mathsf{MPC}}$.**Multiply**. By "observed", we mean that the same sets of honest and/or corrupt parties will fail to reconstruct and consequently request that the protocol abort. (Note that this set of parties is somewhat governed by the function q since this defines how shares are reconstructed (before verification), but we are not claiming any secrecy on this function.) Since the simulator is able to extract these errors and maintain them through the computation (see Figure 14, Figure 15), the adversary will observe the same the same set of parties failing or succeeding in $\mathit{\Pi}_{\mathsf{Opening}}$.**OpenTo** as if the protocol were followed honestly. Note that the errors $\boldsymbol{\delta}$ computed by the simulator will be (subvectors of) sharings of zero if the adversary behaves honestly with the inputs provided by the simulator.

More concretely, consider **Input** for corrupt parties' inputs: the simulator needs to maintain a record of the corrupt shares by computing $\mathbf{s}_{\{P_j\}} := \epsilon^j \cdot \mathbf{u}_{\{P_j\}} + \mathbf{r}_{\{P_j\}}$ for each honest P_j, where ϵ^j is what the adversary sent to honest P_j, not necessarily the same to all honest parties since we only assume pair-wise secure channels. Observe that if the adversary behaves dishonestly and sends a different ϵ to the simulator for some honest party, then the hashes will differ in **Verify** later on; otherwise, the adversary behaves honestly and the simulator is able to extract the well-defined input of the adversary as $s := \epsilon + r$ and forward it to the functionality $\mathcal{F}_{\mathsf{MPC}}$ because the mask r was provided by $\mathcal{F}_{\mathsf{Prep}}$, which the simulator ran internally, and forwarded it to the adversary during **Initialise**.

Note also that maintaining a list of these errors allows the simulator to evaluate the hash function used in **Verify** identically to how honest parties would in the protocol.

Finally, note that the replacement secrets the simulator uses during multiplication were sampled uniformly, so the contribution to the distributions is the same between the hybrids because:

1. The secrets a and b in the triple are never revealed, so the secrets $s - a$ and $s' - b$ are computationally indistinguishable from uniform since a and b are pseudo-randomly generated;
2. The errors induced by the adversary during calls to $\mathit{\Pi}_{\mathsf{Opening}}$.**Broadcast** and $\mathit{\Pi}_{\mathsf{Opening}}$.**OpenTo** were carried through.

Hybrid 1* → *Hybrid 2 Since $\mathbf{r}$ is generated from pre-processing and the encoded secret r is uniform and unknown to the adversary, the broadcasted value ϵ masks the input of honest parties computationally since r is only generated as a pseudo-random secret. Indeed, note that every call to $\mathit{\Pi}_{\mathsf{Opening}}$.**OpenTo** is of sharings encoding uniformly random secrets, and that when $\mathit{\Pi}_{\mathsf{Opening}}$.**Broadcast** is called when honest parties are to provide inputs, the broadcasted value is also uniform. Thus the transcript reveals nothing about the inputs of honest parties, and in particular (computationally) hides the fact that the simulator samples honest parties' inputs during $\mathcal{S}_{\mathsf{MPC}}$.**Input** in **Hybrid 2**.

Moreover, the simulator is able to create a convincing output vector which shares the secret output by the functionality because a set of unqualified parties learns *no* information about the secret from the shares they hold. Note that some generalised forms of secret-sharing do not offer such guarantees: see, for example, ramp schemes [10] in which some sets of parties are neither unqualified nor qualified and are allowed to learn *something* about the secret. If the set of shares revealed something about the secret (say, for example, that the sampled secret lay in a strict subset of the domain) then if the functionality later outputs a secret not in this domain, the simulator cannot necessarily provide a set of shares which convincingly opens to the actual input (and not the sampled secret). Since this does not happen, the inputs and outputs of all parties are the same, the transcript is independent of the input, and the parties abort with the same distributions in both worlds.

Since each pair of hybrids is indistinguishable (computationally or better), by the triangle inequality applied to the success probabilities adversary in distinguishing between each pair, the $\mathcal{F}_{\mathsf{Prep}}$-hybrid world and the ideal world are computationally indistinguishable. □

B.2 Proof of Proposition 2

We will first briefly justify the correctness of the protocol in the sense that it will always abort if the adversary cheats. Note that our protocol allows any party to alter the encoded secrets simply by changing their beginning summand x_i; sacrificing is used to ensure that this does not happen. Let the error the adversary introduces be $\boldsymbol{\delta}$.

1. If $N\boldsymbol{\delta} \neq \mathbf{0}$ then the adversary has introduced errors so that the resulting vector is *not* a valid sharing of any secret in the LSSS. An error of this form will cause honest parties construct different but *valid* share vectors when opening shares during sacrifice. Such an error is moreover *not* detected during the sacrifice stage, during $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$, if and only if the adversary manages to cancel the error in the other triple used during sacrifice: i.e. if the adversary manages to introduce errors $\boldsymbol{\delta}^{a,b}$ and $\boldsymbol{\delta}^{x,y}$ on $\mathbf{c}$ and $\mathbf{z}$, and errors when broadcasting for the sacrifice check so that the vector defined by
$$r \cdot \boldsymbol{\delta}^{x,y}_{\{P_i\}} - (\rho - \rho^i) \cdot \mathbf{b}_{\{P_i\}} - (\sigma^i - \sigma) \cdot \mathbf{a}_{\{P_i\}} - \boldsymbol{\delta}^{a,b}_{\{P_i\}} - (\rho^i \cdot \sigma^i - \rho \cdot \sigma) \cdot \mathbf{u}_{\{P_i\}}$$
for each $i \in [n]$ is a valid sharing of zero, where ρ^i and σ^i are the values reconstructed by P_i in opening $\boldsymbol{\rho}$ and $\boldsymbol{\sigma}$. Since ρ and σ are uniform, the protocol will abort during Π_{Opening} except with probability $1/q$ where $q = |\mathbb{F}|$.
2. If $N\boldsymbol{\delta} = \mathbf{0}$ then the adversary has introduced errors so that the resulting vector is a valid sharing of some secret which is different from the secret encoded by the original additive sharing. In this case, the protocol will abort unless the adversary manages to compute something to add to $\mathbf{c}$ and $\mathbf{z}$ such that in the sacrifice step it holds that
$$0 = r \cdot (x \cdot y + \delta^z) + a' \cdot b' - (a' \cdot b' + \delta^c) - r \cdot x \cdot y.$$
In other words, the adversary must add on errors to $\mathbf{c}$ and $\mathbf{z}$ encoding secrets δ^c and δ^z such that $r\delta^z - \delta^c = 0$, which again only happens with probability

$1/q$ since r is uniformly random (since it is derived from ρ and σ which are uniform.

Thus in either case, the protocol will abort with overwhelming probability.

We now turn to proving security. Our simulator is given in Figure 17 and Figure 18. We will discuss correctness of the simulation, and then prove that the simulator successfully provides a transcript to adversary so that no environment, who provides all inputs, sees all outputs of honest parties, and additionally controls all internal behaviour of the adversary, can distinguish between the ideal world in which the simulator acts with $\mathcal{F}_{\mathsf{Prep}}$, and the hybrid world in which the honest parties interact with the adversary as in the protocol and additionally have access to $\mathcal{F}_{\mathsf{Rand}}$.

Correctness of Simulation Since we are in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid world, the simulator is required to respond to all calls of the adversary to generate PRSSs and PRZSs and public random values r.

Throughout the simulation, the simulator keeps track of errors introduced by the adversary on honest parties' shares by storing a vector $\boldsymbol{\delta}$ along with share vector; these vectors (potentially) change after each round of communication, which is the only time the adversary can influence shares of honest parties. More specifically, the simulator must keep track of errors from Π_{Convert} so that the correct distribution of honest parties aborts during **OpenTo**() or **Verify**.

Because all of the raw data used to construct the Beaver triples comes from either $\mathcal{F}_{\mathsf{Rand}}$ or $\mathcal{F}_{\mathsf{Prep}}$, the simulator can always compute what the adversary would compute were he to follow the protocol, as the simulator runs $\mathcal{F}_{\mathsf{Rand}}$ internally and interfaces between the adversary and $\mathcal{F}_{\mathsf{Prep}}$; using this information, he can exactly determine the errors the adversary introduces when sending information to (emulated) honest parties; the simulator can then use this information to provide a transcript indistinguishable between the hybrid world and the ideal world, even without learning the shares received by honest parties, as we shall prove.

Note that if the LSSS admits share vectors with unqualified support, the adversary can potentially create a set of shares different from those generated by the multiplication protocol but so that the sacrifice check passes; this can be viewed as re-randomising shares, as was discussed in Section 4. However, by Lemma 2, any re-randomisation cannot affect the value of the *secrets* encoded. In particular, this means that the shares the adversary holds at the end may not be the same as the shares given to the functionality by the simulator, who simply passes on exactly what the adversary *would* compute were he to follow the protocol; however, if the adversary follows the protocol then the distributions of the final share vectors held by all parties are the same in both real (hybrid) and ideal worlds, since by linearity,

$$\{\mathbf{c} : \mathbf{c} \text{ encodes the secret } c\} = \{\mathbf{c} + \mathbf{e} : \mathbf{c} \text{ encodes the secret } c\}$$

for any $\mathbf{e}$ with $\psi(supp(\mathbf{e})) \subseteq A$, since $\mathbf{e}$ must encode 0, and the functionality uniformly selects the share vector $\mathbf{c}$ to which to lift the subvector $\mathbf{c}_A$, subject to the constraint that it must share $a \cdot b$.

Indistinguishability

Proof. We will show directly that our simulator provides the correct view, instead of hopping between a series of hybrids.

The simulator creates a view for the adversary as if it were running the protocol, and responds to all calls to $\mathcal{F}_{\mathsf{Rand}}$. Note that there is no input from the adversary to the functionality, so there is no need for the simulator to extract any inputs from the adversary. The simulator runs $\mathcal{F}_{\mathsf{Rand}}$ honestly internally for calls to PRZS. For the calls to the simulator for $\mathbf{x}$ and $\mathbf{y}$, the simulator runs $\mathcal{F}_{\mathsf{Rand}}$ internally honestly and sends the adversary the appropriate output. The secrets x and y are never revealed to the adversary and the simulator can compute them since it stored all outputs of $\mathcal{F}_{\mathsf{Rand}}$. Thus for these two calls by the adversary to $\mathcal{F}_{\mathsf{Rand}}$, the protocol transcript produced in the real world and the transcript the simulator produces are identically distributed.

When the adversary requests a PRSS for $\mathbf{a}$ and $\mathbf{b}$, the simulator only receives a subset of shares for corrupt parties – namely $\mathbf{a}_A$ and $\mathbf{b}_A$. Thus the simulator cannot know secrets a and b sampled by $\mathcal{F}_{\mathsf{Prep}}$, and so it samples random secrets a' and b' and creates share vectors consistent with these secrets and the outputs of $\mathcal{F}_{\mathsf{Prep}}$.

However, later in the protocol the simulator replaces share vectors which should encode the secrets $r \cdot \mathbf{x} - \mathbf{a}'$ and $\mathbf{y} - \mathbf{b}'$ with share vectors encoding uniform secrets but which also agree on shares held by corrupt parties; the simulator can do this as it knows the values of $\mathbf{a}'_A$, $\mathbf{b}'_A$, $\mathbf{x}_A$ and $\mathbf{y}_A$ from the above.

Note that by definition of MSP, it is always possible to create a share vector for any secret given only some unqualified subvector. This fact is used repeatedly in our simulator to replace the shares of honest parties. Note that choosing r in the way defined in Figure 17, it is indistinguishable from uniform as σ and ρ were sampled uniformly. This removes the dependence of the transcript during Π_{Opening} on the secrets a' and b' the simulator initially sampled.

Observe that if $\rho^i = \rho$ and $\sigma^i = \sigma$ for all i, then since the simulator simply runs the protocol honestly and stores errors introduced, the sacrifice will fail, with the same set of parties initially calling for abort, with the same distributions in both worlds.

We still require that the transcript reveal nothing about the secrets the simulator sampled, a' and b'. Consider what can be learnt from the protocol transcript during Π_{Convert}:

1. For any row which is not a standard basis vector, the owner of the row receives shares from all other parties and learns the value of $\langle M_j, \mathbf{x} \rangle$. Since the summands a_i^j contain PRZS masks, the recipient can learn nothing more than what they can learn from their own shares and the value $\langle M_j, \mathbf{x} \rangle$.
2. For any row which is a standard basis vector, the owner can learn the sum of reshares assigned to the column whose index is the index of the non-

zero entry of the row. Here the n-party PRZS added to $\langle x \rangle$ prevents the recipient from learning anything about any of the shares x_i of the secret before conversion, and hence nothing from the original secrets a and b used to create $\langle x \rangle$ which otherwise might be learnable since x_i is a sum of a subset of shares of $\mathbf{a}$ and $\mathbf{b}$.

Note that for any column in which $\boldsymbol{\varepsilon}_k = 0$, the parties randomise the sharing by choosing a random scalar multiple of this column to the final share vector. This is possible because the fact that $\boldsymbol{\varepsilon}_k = 0$ means that any amount of the secret value shared via share vectors in the linear span of column $M[k]$ are cancelled out in recombination. Randomising in this way is necessary because the secrets corresponding to these columns are essentially used to mask the secret. (Consider an MSP for Shamir, as in Appendix A: all entries of the target vector but the first are zero; if one were always to assume one column of the latter columns of the MSP matrix contributed nothing to the final share vector, the resulting access structure would have threshold one less than before – i.e. the access structure would change).

This shows that the transcript during Π_{Convert} does not reveal information to the adversary about the values of the secrets being multiplied.

Now consider values opened in Π_{Opening}. In the real (hybrid) world, x and y are uniform, and hence the broadcasted values $r \cdot x - a$ and $y - b$ reveal nothing about a and b; thus the simulation in which ρ and σ encode uniform secrets is contributes the same distribution of elements here. Then, observe that in $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ on $\mathbf{t}$ the shares of honest parties are independent of a' and b', since even $\boldsymbol{\delta}^{a,b}$ is computed by the adversary based only on knowledge of $\mathbf{a}_A$ and $\mathbf{b}_A$. Thus the transcript is independent of a', b' and $a' \cdot b'$, and so the environment is not able to detect that a different triple has been generated from the one in the ideal world (with high probability) since the simulator did not receive shares for honest parties from $\mathcal{F}_{\mathsf{Prep}}$ and thus could not compute the secrets. This shows that the transcript during Π_{Opening} does not reveal information on a' and b', and hence hides the fact that they are not the same as in the ideal world.

Thus, since the adversary and environment cannot learn information on the encoded secrets from the transcript, the simulator can provide a convincing transcript by replacing the share vectors which are multiplied and opened with share vectors encoding uniform secrets, subject to the constraint that the share vectors be consistent with whatever was previously seen by the adversary.

Since the triples are produced independently of each other, the combined distribution of triples offers no information to the environment.

Since the hybrids are computationally indistinguishable, the hybrid world and the ideal world are also computationally indistinguishable. □

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

The set $A \subseteq \mathcal{P}$ is the set of corrupt parties; let $I_A \subseteq [n]$ denote the corresponding indexing set. We write $i \notin I_A$, and $P_i \notin A$, if P_i is honest. See the protocol description in Figure 2 for the definition of $\boldsymbol{\lambda}^i$ and Section 4 for the definition of q. We denote by H^i the hash function updated locally by P_i. Error vectors $\boldsymbol{\delta}$ are computed in $\mathcal{S}_{\mathsf{MPC}}$.**Input**, $\mathcal{S}_{\mathsf{MPC}}$.**Multiply** and $\mathcal{S}_{\mathsf{MPC}}$.**Add**. In the protocol, if an honest party never receives an expected message from the adversary, it signals abort; in the simulation, if this happens then simulator sets the internal flag `Abort` to true.

OpenTo(i): By the time this is called in the simulation, the simulator has obtained a secret s, knows the complete share vector $\mathbf{s}$, and has the corresponding the error vector $\boldsymbol{\delta}$.

- To open a secret to all parties, the simulator does the following:
  1. Retrieve from memory the recombination vectors $\boldsymbol{\lambda}^j$, for $j \in [n]$.
  2. For each $j \in I_A$ and $k \in \mathsf{q}(P_j)$, if $\psi(k) \notin A$ then send $\mathbf{s}_k + \boldsymbol{\delta}_k$ to the adversary.
  3. For each $j \notin I_A$ and $k \in \mathsf{q}(P_j)$, if $\psi(k) \in A$ then wait for $\mathbf{s}_k$ from adversary.
  4. For each $j \notin I_A$, concatenate local and received shares into a vector $\mathbf{s}^j_{\mathsf{q}(P_j)} \in \mathbb{F}^{|\mathsf{q}(P_j)|}$.
  5. For each $j \notin I_A$, solve $M_{\mathsf{q}(P_j)}\mathbf{x}^j = \mathbf{s}^j_{\mathsf{q}(P_j)}$ for $\mathbf{x}^j$, if it exists. If for any j there is no such $\mathbf{x}^j$, send the message Abort to the adversary to abort the protocol and set the internal flag `Abort` to true.
  6. If the protocol has not aborted, for all $j \notin I_A$ execute $H^j.\mathsf{Update}(M\mathbf{x}^j)$.
- To open a secret to P_i, the simulator does the following:
  1. If P_i is corrupt,
     (a) For each $P_j \notin A$, send $\mathbf{s}_{\{P_j\}}$ to the adversary.
     (b) If the adversary signals Abort then send the message Abort to the adversary to abort the protocol and set the internal flag `Abort` to true.
  2. If P_i is honest,
     (a) For each $P_j \in A$, for each $i \in I_A$, wait for some subvector $\mathbf{s}'_{\{P_j\}}$ from the adversary.
     (b) Let $\mathbf{s}'$ be the vector $\mathbf{s}$ modified so that $\mathbf{s}_{\{P_j\}} := \mathbf{s}'_{\{P_j\}}$ for all $j \notin I_A$. If $N\mathbf{s}' \neq \mathbf{0}$, then run send the message Abort to the adversary to abort the protocol and set the internal flag `Abort` to true.

Verify The simulator runs the procedure as honest parties would:

1. Compute $h^i := H^i.\mathsf{Output}()$ for all $i \notin I_A$ and send to the adversary.
2. Wait for $\{h^{j,i}\}_{j \in I_A, i \notin I_A}$ from the adversary, where $h^{j,i}$ denotes the output of the hash function sent by the adversary for corrupt P_i to the simulator in place of honest P_j.
3. For each $i \notin I_A$, if $h^{j,i} \neq h^i$ for any $j \in I_A$, send the message Abort to the adversary to abort the protocol and set the internal flag `Abort` to true.

Broadcast: The simulator runs the procedure as honest parties would: when P_i needs to run this procedure to broadcast some ϵ, the simulator does the following:

- If the broadcasting party P_i is honest, then for each $j \notin I_A$ set $\epsilon^j := \epsilon$ and then send $\{\epsilon^j\}_{j \notin I_A}$ to the adversary on behalf of P_i.
- If the broadcasting party P_i is corrupt, wait for the adversary to send $\{\epsilon^j\}_{j \notin I_A}$.

For each $j \notin I_A$, update the local hash function $H^j.\mathsf{Update}(\epsilon^j)$.

Figure 13. Simulator $\mathcal{S}_{\mathsf{Opening}}$

Simulator $\mathcal{S}_{\mathsf{MPC}}$ Part 1 of 3 (Commands Part 1 of 3)

The simulator stores all share vectors with their identifiers, along with error vectors (see below). The set I_A denotes the indexing set of corrupt parties. The simulator always performs Π_{Opening} exactly as honest parties would in the protocol, but we define our simulator $\mathcal{S}_{\mathsf{Opening}}$ because we must keep track of, and incorporate into any revealed shares or secrets, all errors the adversary introduces during the protocol.

Initialise: The simulator does the following:

1. Set the internal flag `Abort` to false and for each $i \notin I_A$, initialise a local hash function H^i by running $H^i.\mathsf{Initialise}()$.
2. Initialise an internal copy of $\mathcal{F}_{\mathsf{Prep}}$. When the adversary sends a message (Triple, N_T), execute $\mathcal{F}_{\mathsf{Prep}}.\mathbf{Triples}$ honestly with the adversary and store all information sent from the adversary and output from the functionality. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, send the message Abort to the adversary, and send Init and then Abort to $\mathcal{F}_{\mathsf{MPC}}$ and abort, and otherwise continue. For each share vector $\mathbf{s}$ output from $\mathcal{F}_{\mathsf{Prep}}$, determine the encoded secret s by computing $\langle \mathbf{s}, \boldsymbol{\lambda} \rangle$ using any recombination vector $\boldsymbol{\lambda}$, and store it.
3. Agree with the adversary on the sharing of 1, denoted by $\mathbf{u}$.
4. For generating input masks, the simulator honestly executes the protocol:
   (a) For providing input to corrupt or honest P_i, retrieve a sharing $\mathbf{r}$ and corresponding secret r from memory.
   (b) Run $\mathcal{S}_{\mathsf{Opening}}.\mathbf{OpenTo}(i)$ with the adversary on $\mathbf{r}$.
5. Send the message Init to $\mathcal{F}_{\mathsf{MPC}}$, followed by a message Abort if the internal flag `Abort` is set to true, or OK otherwise.

Input: For P_i to give input, the simulator does the following:

1. Do the following:
   - If P_i is honest, sample $\epsilon \xleftarrow{\$} \mathbb{F}$ and do the following:
     (a) Run $\mathcal{S}_{\mathsf{Opening}}.\mathbf{Broadcast}$ with the adversary, with P_i broadcasting ϵ.
     (b) Define an error vector to be used later by $\boldsymbol{\delta}_{\{P_j\}} := \mathbf{0}$ for all $j \in [n]$ and store the share vector $\mathbf{s} := \epsilon \cdot \mathbf{u} + \mathbf{r}$.
     (c) Using the fresh identifier id, send $(\mathsf{Input}, \mathsf{id}, \perp)_{P_j \in A}$ to $\mathcal{F}_{\mathsf{MPC}}$.
   - If P_i is corrupt, do the following:
     (a) Run $\mathcal{S}_{\mathsf{Opening}}.\mathbf{Broadcast}$ with the adversary (where the adversary provides input) so that the simulator obtains a set $\{\epsilon_j\}_{j \notin I_A}$. Choose any $j \notin I_A$ and set $\epsilon := \epsilon^j$.
     (b) Define an error vector to be used later by $\boldsymbol{\delta}_{\{P_j\}} := (\epsilon^j - \epsilon) \cdot \mathbf{u}_{\{P_j\}}$ for all $j \in [n]$ and also store the share vector $\mathbf{s} := \epsilon \cdot \mathbf{u} + \mathbf{r}$.
     (c) If the internal flag `Abort` has not been set to true, using the fresh identifier id, send $(\mathsf{Input}, \mathsf{id}, \epsilon + r)$ and $(\mathsf{Input}, \mathsf{id}, \perp)_{j \in I_A \setminus \{i\}}$ to the functionality $\mathcal{F}_{\mathsf{MPC}}$, and then additionally send OK; otherwise sample some ϵ, send $(\mathsf{Input}, \mathsf{id}, \epsilon)$ and $(\mathsf{Input}, \mathsf{id}, \perp)_{j \in I_A \setminus \{i\}}$ to $\mathcal{F}_{\mathsf{MPC}}$ and then send the message Abort.

Figure 14. Simulator $\mathcal{S}_{\mathsf{MPC}}$ Part 1 of 3 (Commands Part 1 of 3)

Simulator $\mathcal{S}_{\mathsf{MPC}}$ Part 2 of 3 (Commands Part 2 of 3)

Add: When adding secrets shared via $\mathbf{s}$ and $\mathbf{s}'$ with identifiers id_s and $\mathsf{id}_{s'}$, the simulator does the following:

1. Retrieve $\mathbf{s}$ and $\mathbf{s}'$ from memory along with error vectors $\boldsymbol{\delta}^{(s)}$ and $\boldsymbol{\delta}^{(s')}$.
2. Compute the sum as $\mathbf{s} + \mathbf{s}'$ and a new error vector $\boldsymbol{\delta}^{(s)} + \boldsymbol{\delta}^{(s')}$, and store these with the new identifier id.
3. Send the command $(\mathsf{Add}, \mathsf{id}_s, \mathsf{id}_{s'}, \mathsf{id})$ to $\mathcal{F}_{\mathsf{MPC}}$.

Multiply: When multiplying secrets shared via $\mathbf{s}$ and $\mathbf{s}'$ with identifiers id_s and $\mathsf{id}_{s'}$, the simulator does the following:

- Retrieve the share vectors for the triple $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, and the corresponding identifiers.
- Retrieve from memory the share vectors $\mathbf{s}$ and $\mathbf{s}'$ and corresponding error vectors $\boldsymbol{\delta}^{(s)}$ and $\boldsymbol{\delta}^{(s')}$.
- Sample two share vectors $\boldsymbol{\rho}$ and $\boldsymbol{\sigma}$ of uniform secrets ρ and σ such that $\boldsymbol{\rho}_A = \mathbf{s}_A - \mathbf{a}_A$ and $\boldsymbol{\sigma}_A = \mathbf{s}'_A - \mathbf{b}_A$.
- Execute $\mathcal{S}_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ on $\boldsymbol{\rho} + \boldsymbol{\delta}^{(s)}$ and $\boldsymbol{\sigma} + \boldsymbol{\delta}^{(s')}$ to obtain for honest parties $\{\rho^i\}_{i \notin I_A}$ and $\{\sigma^i\}_{i \notin I_A}$.
- If the internal flag `Abort` has been set to true, send the command $(\mathsf{Multiply}, \mathsf{id}_s, \mathsf{id}_{s'}, \mathsf{id})$ to $\mathcal{F}_{\mathsf{MPC}}$, and then send the message Abort; otherwise, do the following:
  - Compute $\{\rho^i\}_{i \in I_A}$ and $\{\sigma^i\}_{i \in I_A}$ based on knowledge of $\boldsymbol{\rho}_A$ and $\boldsymbol{\sigma}_A$.
  - Store the new share vector as

    $$\mathbf{s}'' := \mathbf{c}_{\{P_i\}} + \rho \cdot \mathbf{s}'_{\{P_i\}} + \sigma \cdot \mathbf{s}_{\{P_i\}} - \rho \cdot \sigma \cdot \mathbf{u}_{\{P_i\}}$$

    and also store the new error on the product defined by

    $$\boldsymbol{\delta}^{(s'')}_{\{P_i\}} := (\rho^i - \rho) \cdot \mathbf{s}'_{\{P_i\}} + \rho^i \cdot \boldsymbol{\delta}^{(s')}_{\{P_i\}} + (\sigma^i - \sigma) \cdot \mathbf{s}_{\{P_i\}} + \sigma^i \cdot \boldsymbol{\delta}^{(s)}_{\{P_i\}} - (\rho^i \cdot \sigma^i - \rho \cdot \sigma) \cdot \mathbf{u}_{\{P_i\}}$$

    for all $i \in [n]$.
  - Send the command $(\mathsf{Multiply}, \mathsf{id}_s, \mathsf{id}_{s'}, \mathsf{id})$ to $\mathcal{F}_{\mathsf{MPC}}$, and then send the message Abort if the internal flag `Abort` was set to true, and otherwise send the message OK.

Figure 15. Simulator $\mathcal{S}_{\mathsf{MPC}}$ Part 2 of 3 (Commands Part 2 of 3)

Simulator $\mathcal{S}_{\mathsf{MPC}}$ Part 3 of 3 (Commands Part 3 of 3)

OutputTo(i): When the adversary requests to open a sharing $\mathbf{s}$ encoding secret s, the simulator does the following:

1. Send the command $(\mathsf{Output}, \mathsf{id}, 0)$ to $\mathcal{F}_{\mathsf{MPC}}$.
2. Retrieve the identifier id used for this secret and the error vector $\boldsymbol{\delta}^{(s)}$.
3. Run $\mathcal{S}_{\mathsf{Opening}}$.**Verify**.
4. – If $i = 0$,
    - (a) Receive the secret s from $\mathcal{F}_{\mathsf{MPC}}$.
    - (b) Run $\mathcal{S}_{\mathsf{Opening}}$.**OpenTo**$(0)$ to open $\mathbf{s} + \boldsymbol{\delta}^{(s)}$.
    - (c) If the internal flag `Abort` has been set to true, send the message Abort to $\mathcal{F}_{\mathsf{MPC}}$ and otherwise run $\mathcal{S}_{\mathsf{Opening}}$.**Verify**.
    - (d) If the internal flag `Abort` has been set to true, send Abort to $\mathcal{F}_{\mathsf{MPC}}$, and otherwise send OK.

   – If $P_i \in \mathcal{P}$,
    - If P_i is corrupt,
        - (a) Send the command $(\mathsf{Output}, \mathsf{id}, i)$ to $\mathcal{F}_{\mathsf{MPC}}$ and receive back the secret s.
        - (b) Retrieve from memory the share vector $\mathbf{s}$ corresponding to this secret along with its error vector $\boldsymbol{\delta}^{(s)}$ and sample a new share vector $\mathbf{t}$ that encodes the secret s and it holds that $\mathbf{s}_A = \mathbf{t}_A$.
        - (c) Run $\mathcal{S}_{\mathsf{Opening}}$.**OpenTo**$(i)$ on $\mathbf{t} + \boldsymbol{\delta}^{(s)}$.
        - (d) If the internal flag `Abort` has been set to true, send Abort to $\mathcal{F}_{\mathsf{MPC}}$, and otherwise send OK.
    - If P_i is honest,
        - (a) Run $\mathcal{S}_{\mathsf{Opening}}$.**OpenTo**$(i)$ with the adversary.
        - (b) If the internal flag `Abort` has been set to true, send Abort to $\mathcal{F}_{\mathsf{MPC}}$, and otherwise send OK.

Figure 16. Simulator $\mathcal{S}_{\mathsf{MPC}}$ Part 3 of 3 (Commands Part 3 of 3)

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

We omit indices for the N_T triples for the sake of clarity and because all triples are generated independently. The simulator does the following:

1. Initialise count.
2. For i from 1 to N_T, do the following:
   - When the adversary sends a message PRSS to the simulator for $\mathcal{F}_{\mathsf{Rand}}$, invoke $\mathcal{F}_{\mathsf{Prep}}$ and forward to the adversary the vectors $\mathbf{a}_A$ and $\mathbf{b}_A$ it receives. Create share vectors $\mathbf{a}'$ and $\mathbf{b}'$ such that they share uniform secrets a' and b', respectively, and such that $\mathbf{a}'_A = \mathbf{a}_A$ and $\mathbf{b}'_A = \mathbf{b}_A$.
   - When the adversary asks for shares of $\mathbf{x}$ and $\mathbf{y}$, run $\mathcal{F}_{\mathsf{Rand}}$ internally and give the appropriate outputs to the adversary.
   - As in the protocol, perform the local computations to obtain additive sharings of the products of a' with b' and x with y. Using knowledge of the shares given to the adversary from $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{Prep}}$, the simulator can do this for all parties, including corrupt.
   - Now run $\mathcal{S}_{\mathsf{Convert}}$ with the adversary; the simulator obtains share vectors $\mathbf{c}$, $\tilde{\mathbf{c}}$ and an error vector $\boldsymbol{\delta}^{a,b}$, where $\mathbf{c}$ is computed by emulating the whole Π_{Convert} protocol honestly using knowledge of $\mathbf{a}'_A$ and $\mathbf{b}'_A$.
   - Do similarly for $\mathbf{x}$ and $\mathbf{y}$ to obtain share vectors $\mathbf{z}$ and $\tilde{\mathbf{z}}$ and error vector $\boldsymbol{\delta}^{x,y}$.
   - Send to $\mathcal{F}_{\mathsf{Prep}}$ the share (sub)-vector $\mathbf{c}_A$ computed as in the execution of Π_{Convert} on the additive sharing of the product of $\mathbf{a}'$ and $\mathbf{b}'$.
   - Run $\mathcal{F}_{\mathsf{Rand}}$ honestly internally to obtain some r and send it to the adversary.
   - Create sharings $\boldsymbol{\rho}$ and $\boldsymbol{\sigma}$ of uniform secrets ρ and σ subject to the requirement that $\boldsymbol{\rho}_A = r \cdot \mathbf{x}_A - \mathbf{a}'_A$ and $\boldsymbol{\sigma}_A = \mathbf{y}_A - \mathbf{b}'_A$, which is possible as A is unqualified.
   - Execute $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ as in the protocol (aborting if necessary and sending the message Abort to $\mathcal{F}_{\mathsf{Prep}}$), but opening $\boldsymbol{\rho}$ and $\boldsymbol{\sigma}$ instead of $r\cdot\mathbf{x}-\mathbf{a}'$ and $\mathbf{y} - \mathbf{b}'$. Denote by ρ^i and σ^i respectively the values each party $i \notin I_A$ opened the secrets to.
   - Now set

     $$\mathbf{t}_{\{P_i\}} := r \cdot (\mathbf{z}_{\{P_i\}} + \boldsymbol{\delta}^{x,y}_{\{P_i\}}) - \rho^i \cdot (\mathbf{y}_{\{P_i\}} - \boldsymbol{\sigma}_{\{P_i\}}) - \sigma^i \cdot (r \cdot \mathbf{x}_{\{P_i\}} - \boldsymbol{\rho}_{\{P_i\}})$$
     $$-(\mathbf{c}_{\{P_i\}} + \boldsymbol{\delta}^{a,b}_{\{P_i\}}) - \rho^i \cdot \sigma^i \cdot \mathbf{u}_{\{P_i\}}$$

     and run $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ on $\mathbf{t}$. If the procedure $\Pi_{\mathsf{Opening}}.\mathbf{OpenTo}(0)$ aborts or one of the (emulated) honest parties computes an opening different from 0, then the simulator sends the message Abort to $\mathcal{F}_{\mathsf{Prep}}$ and the adversary.
3. Execute $\Pi_{\mathsf{Opening}}.\mathbf{Verify}$ as in the protocol, sending the message Abort to $\mathcal{F}_{\mathsf{Prep}}$ if an honest party would abort and otherwise sending OK.

Figure 17. Simulator $\mathcal{S}_{\mathsf{Prep}}$

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

When this part of the simulation is called, the simulator has one summand x_i of the secret product for each party (including summands the corrupt parties are supposed to be holding, since they were derived from secrets known to the simulator).

1. Following the protocol, the simulator samples a set of shares $\{x_{i,k} : i \notin A,\ k \in K_i\}$ such that $\sum_{k \in K_i} x_{i,k} = x_i$ for each $i \notin I_A$.
2. Following the protocol, for each $j \in [m]$, the simulator does the following:
3. If j is *not* a standard basis vector, the simulator does the following:
   (a) When the adversary sends the command $(\mathsf{PRZS}, \mathsf{count}, \mathcal{P})$ to the simulator, run $\mathcal{F}_{\mathsf{Rand}}$ honestly to obtain some PRZS $\langle t^j \rangle$, record all outputs, and forward appropriate outputs to the adversary.
   (b) If $\psi(j) \notin A$ then wait to receive a set $\{\tilde{a}_i^j : i \in I_A\}$ of shares from the adversary, and additionally compute the shares $\{a_i^j : i \notin I_A\}$ by following the protocol for honest parties. Compute $\tilde{\mathbf{s}}_j := \sum_{i \in I_A}^{n} \tilde{a}_i^j + \sum_{i \notin I_A} a_i^j$ and $\mathbf{s}_j := \sum_{i=1}^{n} a_i^j$.
   (c) If $\psi(j) \in A$ then execute the protocol honestly to compute the shares $\{a_i^j : i \notin I_A\}$, and send them to the adversary. Additionally, compute the shares an honest adversary would compute $\{a_i^j : i \in I_A\}$ using knowledge of outputs of $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{Rand}}$ and then set $\mathbf{s}_j := \tilde{\mathbf{s}}_j := \sum_{i=1}^{n} a_i^j$.
4. If j *is* a standard basis vector $\mathbf{e}^k$ for some k, the simulator does the following:
   (a) When the adversary sends the command $(\mathsf{PRZS}, \mathsf{count}, X)$ to the simulator, run $\mathcal{F}_{\mathsf{Rand}}$ honestly to obtain some PRZS $\langle t^j \rangle$, record all outputs, and forward appropriate outputs to the adversary.
   (b) If $\psi(j) \notin A$ then wait to receive a set $\{\tilde{a}_i^j : i \in X \cap I_A\}$ of shares from the adversary. Compute $\tilde{\mathbf{s}}_j := \sum_{i \in X \cap I_A} \tilde{a}_i^j + \sum_{i \in X \setminus I_A} a_i^j$ and $\mathbf{s}_j := \sum_{i \in X} a_i^j$.
   (c) If $\psi(j) \in A$ then, execute the protocol honestly and send $\{a_i^j : i \in X \setminus I_A\}$ to the adversary. Additionally, compute the shares an honest adversary would compute $\{a_i^j : i \in I_A\}$ using knowledge of outputs of $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{Rand}}$ and then set $\mathbf{s}_j := \tilde{\mathbf{s}}_j := \sum_{i=1}^{n} a_i^j$.
5. The simulator computes an error

$$\boldsymbol{\delta} := \mathbf{s} - \tilde{\mathbf{s}}$$

(and note that $\boldsymbol{\delta}_{\mathcal{P} \setminus A} = \mathbf{0}$).

Figure 18. Simulator $\mathcal{S}_{\mathsf{Convert}}$

C Algorithm for finding a (crudely optimised) map q

See Section 4 for the definition of q. In this section we provide an algorithm to find a map q such that $|supp(\mathsf{q}(i))|$ is “quite small”, for all i. We have a choice as to whether to minimise the number of implied uni- or bi-directional channels. In the algorithm below, by choosing the qualified sets as described we crudely optimise the number of uni-directional channels.

For the algorithm, we assume the MSP matrix M has linearly independent columns, since if not then we can take a linearly independent subset to obtain a new LSSS which realises the same access structure [4]. We also assume that the map ψ is increasing so that each party owns a contiguous submatrix of M. If it is not, the rows can be interchanged so that this is true.

```
Algorithm for determining q

1. For each $P_i \in \mathcal{P}$,
   (a) Set $A := M_{\{P_i\}}$, $S := \varnothing$, $k := 1$, and ExhaustedQ := false.
   (b) Set $Q \in \Gamma^-$ to be a smallest minimally qualified set containing $P_i$.
   (c) While rank$(A) <$ NoOfColumns$(M)$ and ExhaustedQ = false Do
        i. While $\psi(k) \notin Q$ and $k \leq$ NoOfRows$(M)$ do $k := k + 1$.
       ii. If $k \leq$ NoOfRows$(M)$ and the row $M_k$ of $M$ is linearly independent of
           the rows of $A$ then
            A. Insert $M_k$ into $A$.
            B. $S := S \cup \{k\}$.
      iii. End if.
       iv. $k := k + 1$.
        v. If $k >$ NoOfRows$(M)$ then ExhaustedQ := true.
   (d) End While.
   (e) $k := 1$.
   (f) While rank$(A) <$ NoOfColumns$(M)$ and $k \leq$ NoOfRows$(M)$ do
        i. While $\psi(k) \in Q$ and $k \leq$ NoOfRows$(M)$ do $k := k + 1$.
       ii. If $k \leq$ NoOfRows$(M)$ and the row $M_k$ of $M$ is linearly independent of
           the rows of $A$ then
            A. Insert $M_k$ into $A$.
            B. $S := S \cup \{k\}$.
      iii. End if.
       iv. $k := k + 1$.
   (g) End While.
   (h) Set $\mathsf{q}(P_i) := S$.
2. End For.
3. Output q.
```

Figure 19. Algorithm for determining q

D Share-reconstructable

In this section, we briefly define "share-reconstructable" MSPs as those in which share vectors can be reconstructed entirely from any qualified subvector. They are of interest to us because their use offers particularly good communication efficiency in our protocol, as $\mathsf{q}(P_i)$ is "as small as possible" for each $P_i \in \mathcal{P}$, which makes our opening protocol very efficient. If an MSP is used in our protocol which is both share-reconstructable *and* ideal (such as Shamir's scheme), then the communication cost in our protocol obtains its minimum for that access structure.

Definition 2. *Let $\mathcal{M} = (\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ be an MSP, where* $\ker(M) = \{\mathbf{0}\}$*, computing a $\mathcal{Q}_2$ access structure Γ. $\mathcal{M}$ is called* share-reconstructable *if for every any $\mathbf{s} \in im(M)$, for every $Q \in \Gamma$, $\mathbf{s}_Q$ uniquely determines the vector* $\mathbf{s}$*.*

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

Lemma 4. *Let $\mathcal{M} = (\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ be an MSP, where* $\ker(M) = \{\mathbf{0}\}$*, computing a $\mathcal{Q}_2$ access structure Γ. Then $\mathcal{M}$ is share-reconstructable if and only if for every $Q \in \Gamma$ it holds that* $\ker(M_Q) = \{\mathbf{0}\}$*.*

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

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

Example 1. Shamir's secret-sharing scheme is an example since any t rows of the Vandermonde matrix are linearly independent.

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

$$\begin{array}{c} P_1 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{pmatrix} 1\ 0\ 0 \\ 0\ 1\ 0 \\ 1\ 0\ 1 \\ 0\ 1\ 1 \\ 0\ 0\ 1 \end{pmatrix}$$

where $\boldsymbol{\varepsilon} = (1, 1, 1)$. The reader may check that $\ker(M_Q) = \{\mathbf{0}\}$ for every $Q \in \mathit{\Gamma}^-$, so given $\mathbf{s}_Q$ there is always unique solution to $M_Q\mathbf{x} = \mathbf{s}_Q$ for $\mathbf{x}$, and hence $\mathbf{s}$ can be determined from $\mathbf{s}_Q$ by finding $\mathbf{x}$ and computing $\mathbf{s} = M\mathbf{x}$.

We know by Lemma 1 that for any MSP computing a $\mathcal{Q}_2$ access structure, share vectors all have qualified support unless they encode the secret 0; to allow a unique construction of each share vector from qualified subsets, share-reconstructable MSPs are those for which the secret 0 is only encoded via the zero vector or with share vectors with qualified support.

Lemma 5. *Let $\mathcal{M} = (\mathbb{F}, M, \boldsymbol{\varepsilon}, \psi)$ be an MSP, where $\ker(M) = \{\mathbf{0}\}$, computing a $\mathcal{Q}_2$ access structure $\mathit{\Gamma}$. Then $\mathcal{M}$ is share reconstructable if and only if*

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

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

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

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

Conversely, suppose there exists some $\mathbf{s} \in \mathrm{im}(M)$ such that $supp(\mathbf{s}) \notin \mathit{\Gamma} \cup \{\varnothing\}$, i.e., $\mathbf{s} \neq \mathbf{0}$ and $supp(\mathbf{s})$ is unqualified. Let $\mathbf{x}$ be such that $\mathbf{s} = M\mathbf{x}$, which exists since $\mathbf{s} \in \mathrm{im}(M)$. Then $Q := \mathcal{P} \setminus \psi(supp(\mathbf{s}))$ is qualified since $\mathit{\Gamma}$ is $\mathcal{Q}_2$. Since $\mathbf{s}_Q = \mathbf{0}$, we have $M_Q\mathbf{x} = \mathbf{s}_Q = \mathbf{0}$, so $\mathbf{x} \in \ker(M_Q)$. If $\mathbf{x} = \mathbf{0}$ then $\mathbf{s} = M\mathbf{x} = \mathbf{0}$; but $supp(\mathbf{s}) \neq \varnothing$, so this is not the case, and so $\ker(M_Q) \neq \{\mathbf{0}\}$, and thus $\mathcal{M}$ is not share-reconstructable by Lemma 4. □

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

Theorem 1. *For every $\mathcal{Q}_2$ access structure there exists a share-reconstructable MSP computing it.*

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