Reducing Communication Channels in MPC

Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood

University of Bristol, Bristol, UK.
M.Keller@bristol.ac.uk,dragos.rotaru@bristol.ac.uk,
nigel@cs.bris.ac.uk,t.wood@bristol.ac.uk

Abstract. In both information theoretic and computationally secure Multi-Party Computation (MPC) protocols the parties are usually assumed to be connected by a complete network of, respectively, secure or authenticated channels. Taking inspiration from a recent, highly efficient, 1-out-of-3 computationally secure MPC protocol of Araki et al, we show how to perform computationally secure MPC for an arbitrary Q^2 access structure over an incomplete network. Our tool is to combine the practical techniques of Araki with the information theoretic approach of Maurer for arbitrary Q^2 structures. We present both passive and actively secure (with abort) variants of our protocol. In all cases we require less communication channels than Maurer's protocol, at the expense of requiring pre-shared secret keys for Pseudo-Random Functions (PRFs). By shedding light on the theoretical underpinnings of the recent protocol of Araki et al. we hope that our work may result in future highly communication-efficient protocols for other access structures.

1 Introduction

The development of secret sharing based secure Multi-Party Computation (MPC) is generally considered to lie in two distinct camps. In the first camp lies the information theoretic protocols arising from the original work of Ben-Or, Goldwasser and Wigderson [BOGW88] and Chaum, Crepeau and Damgård [CCD88]. In this line of work adversarial parties are assumed to be computationally unbounded, and parties in an MPC protocol are assumed to be connected by a *complete network* of *secure channels*. Such a model was originally introduced in the context of threshold adversary structures, i.e. t-out-of-n secret sharing schemes which could tolerate up to t adversaries amongst n parties. To obtain passively secure protocols one requires $t < n/2$, and to obtain actively secure protocols one requires $t < n/3$; such conditions are also sufficient. These threshold structures were extended to arbitrary access/adversary structures by Hirt and Maurer [HM97], in which case the two necessary and sufficient conditions become Q^2 and Q^3 respectively.

Another line of work which considered computationally bounded adversaries started with [GMW87, GL02]. Here the parties are connected by a *complete network* of *authenticated channels*. Here one can obtain actively secure protocols in the threshold case when $t < n/2$ (i.e. honest majority), and one can obtain

active security *with abort* when only one party is honest. Generally speaking, such computationally secure protocols are less efficient than the information theoretic protocols as they usually assume the need for some form of public key cryptography.

In recent years there has been considerable progress in *practical* MPC by marrying the two approaches together. For example, the VIFF [DGKN09], BDOZ [BDOZ11], SPDZ [DPSZ12] and Tiny-OT [NNOB12] protocols are computationally secure and use information theoretic primitives in an online phase, but computationally secure primitives in an offline/pre-processing phase. The offline phase is used to produce so-called Beaver Triples [Bea96], which are then consumed in the online phase. In these protocols, parties are still connected by a *complete network* of *authenticated channels*, and they are usually in the full threshold model (i.e. when only party is assumed to be honest). A key observation in much of the practical MPC work of the last few years is that communication costs are the main bottleneck.

However, recent work has provided a new method to unify information theoretic and computationally secure protocols. In [AFL+16] a very efficient passively secure MPC evaluation of the AES circuit is given in the case of a 1-out-of-3 adversary structure. This is then generalised to an actively secure protocol in [FLNW16]. Both protocols require a pre-processing phase making use of *symmetric key* cryptographic primitives only; thus the pre-processing is much faster than for the full threshold protocols mentioned above. In this paper we generalise both protocols of Araki et al. to all Q^2 access structures, and in the process hopefully shed some light onto the fundamental nature of what initially appear to be very specific constructions for 1-out-of-3 adversary structures in the protocols of Araki et al.

The passively secure protocol of [AFL+16] makes use of a number of optimizations to the basic offline/online hybrid paradigm. Firstly, the offline phase is only used to produce additive sharings of zero. This therefore dispenses with the expensive pre-production of Beaver triples from the other hybrid protocols. Additive sharings of zero can be easily produced using symmetric key primitives and pre-shared secrets. Secondly, the underlying network is *not* assumed to be *complete*: instead of each of the three parties being connected to the other two parties, each party is only connected to *one* other party via a *secure channel*. Thirdly, parties need only transmit one finite field element per multiplication. On the downside, however, each party needs to hold two finite field elements per share, as opposed to using an ideal secret sharing scheme in which each party only holds one finite field element.

The underlying protocol, bar the use of the additive sharings of zero, is highly reminiscent of the Sharemind system [BLW08], which also assumes a 1-out-of-3 adversary structure, since both [AFL+16] and [BLW08] are based on replicated secret sharing, and hence are closely related to the MPC-made-Simple approach of Maurer [Mau06]. Thus for the case of this specific adversary structure the work in [AFL+16] shows that by using cryptography one can obtain optimizations of the MPC-made-Simple approach of Maurer.

The active variant of the protocol of Araki et al. [FLNW16] uses the passively secure protocol (over an incomplete network of *secure* channels) to run an offline phase which produces the Beaver triples. These are then consumed in the online phase, by using the triples to check the passively secure multiplication of actual secrets. The online phase runs over an incomplete network of *authenticated* channels.

The question therefore arises as to whether the approach outlined in [AFL+16], [FLNW16], and [BLW08] is particularly tied to the 1-out-of-3 adversary structure, or whether it generalises to other access/adversary structures. In this paper we show that the basic passively secure protocol will indeed generalise to arbitrary Q^2 access structures. We evaluate how many finite field elements need to be exchanged over how many *secure* channels. Our passively secure protocol is for general Q^2 structures, implemented via replicated secret sharing. When specialised to threshold structures we do not obtain any communication efficiency over using Shamir sharing; but we do obtain a reduction in the required *number of secure channels*.

We then show how to extend this to an actively secure protocol (with abort) for any Q^2 access structure. We take a more traditional approach than [FLNW16] to obtain active security. In particular we utilize our passive protocol as an offline phase, then in the online phase multiplication is performed via standard Beaver multiplication over an incomplete network of *authenticated* channels. We only require a full network of *secure* channels in the active protocol to obtain (verified) private output in the online phase.

Our work may be viewed as optimizing the communication of the MPC-made-Simple approach for all Q^2 access structures, if we allow for some some cryptographic assumptions. This work also initiates the study of how many communication channels are necessary to perform MPC in general.

1.1 Prior Work

As remarked above the MPC protocol, known as BGW [BOGW88], showed that every functionality can be computed with perfect security, assuming the parties are connected by pairwise private communication channels, and either that the adversary acts semi-honestly and corrupts at most half of the parties, or that the adversary acts maliciously and corrupts at most one third. The protocol makes use of Shamir's secret sharing [Sha79], which is a perfect secret sharing scheme for threshold access structures. Addition of secrets requires no communication, but during a multiplication, in order for the polynomial encoding the secret to be uniformly random (which is required for security), every party must send a share to every other party. Thus, in total, $O(n^2)$ field elements need to be transmitted, over a complete secure network. At roughly the same time, Chaum, Crepeau and Damgård devised a different yet closely related scheme offering essentially the same results.

Subsequently, Hirt and Maurer [HM97] showed a generalisation of the techniques to an arbitrary access structure, providing necessary and sufficient conditions on the access structure for the parties to be able to compute any function

securely. The conditions were subsequently called Q^2 and Q^3. The construction of their protocol involves a recursive decomposition of the set of all parties until the base level at which parties are grouped in threes; at each level the parties execute the BGW protocol. Beaver and Wool [BW98] then showed how to improve the communication costs of Hirt and Maurer's protocol, by providing a more direct protocol without needing a recursive definition. Finally Maurer [Mau06] presented a cleaner definition and construction, using replicated secret-sharing, with essentially the same methodology as Beaver and Wool.

The basic technique in the passive case is as follows: Suppose the parties have secrets shared in a Q^2 (i.e. multiplicative) linear secret-sharing scheme. This means that parties can perform local computations, on the shares of two secrets to be multiplied, so that they hold so that they each obtain a summand of the product. The parties then create a sharing in the secret-sharing scheme of this summand and then send the shares to the appropriate parties, according to the access structure. Since the secret-sharing scheme is additive, the parties can then locally sum all shares they receive to the one they generated so that together they obtain a sharing of the product. The local computations that the parties perform require some agreement of which computations each party shall undertake. The main cost is the need to compute a new resharing for each parties partial sum. Leading to a communication cost of $O(n^2)$ over a complete network of secure channels.

The above overview, however, hides some crucial information. Expressing the communication cost as $O(n^2)$ potentially hides a very large constant, depending on the actual access structure and finite field $\mathbb{F}_q$ involved. In the case of threshold structures when $q > n$ one can utilize Shamir secret sharing, which is an ideal secret sharing scheme, and so (in this case) the total communication cost is exactly $n \cdot (n-1)$ field elements; for other access structures, or even the case of threshold structures when $q \leq n$, one needs to use more elaborate secret sharing schemes, or to extend the base field. This has led some authors to consider using algebraic-geometric codes to produce more efficient secret sharing schemes (see, for example, [CDN15][Part II]). Such works try to stay within the information theoretic model, but aim to select secret sharing schemes which are as close to ideal as possible.

In another line of work [HIK07] the authors examine MPC based on Oblivious Transfer (OT) and aim to reduce the total number of pairwise OT channels needed to perform an MPC calculation.

The Sharemind system [BLW08] was the first practical system to make use of an incomplete network of communication. By using replicated secret sharing in the case of a 1-out-of-3 adversary structure, the authors were able to set up a passively secure multiplication protocol which requires only three secure channels, as opposed to the six secure channels which would be required by following the method of Maurer precisely. Finally, [AFL$^+$16] extended this idea by making use of a computational assumption to build a pre-processing phase which allows the evaluation of efficient binary circuits using secret sharing over the field $\mathbb{F}_2$, for a 1-out-of-3 adversary structure, with only three secure channels.

A key point is that the pre-processing (requiring cryptographic assumptions) is so trivial that it can actually be carried out at the same time as the main online phase.

1.2 Our Work

We take the protocol of Maurer [Mau06] for an arbitrary Q^2 access structure and combine it with the pre-processing idea of [AFL+16] to reduce the required number of secure channels and the number of communicated field elements needed. Our actively secure protocol, also requires only a Q^2 access structure; which does not contradict any impossibility results as recall we are assuming a computationally bounded adversary.

The number of channels and total data communication needed for a given access structure depends on what the access structure looks like (or, more precisely, what the maximally unqualified sets of the scheme are). We see our work as using cryptography to optimize the information theoretic protocol of [Mau06], and hence shedding light on the (what appears at first sight ad hoc) construction of [AFL+16].

To provide a concrete basis for our discussion, we provide the following set of maximally unqualified sets for a six party access structure, which we shall use as a running example throughout this paper:

$$\begin{aligned}\mathcal{M} = \Big\{&\{2,5,6\},\{3,5,6\},\{4,5,6\},\\ &\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{1,6\},\\ &\{2,3\},\{2,4\},\{3,4\}\Big\}\end{aligned}$$

Our methodology makes a great deal of usage of the set of complements of these sets which we denote by $\mathcal{B}$; i.e.

$$\begin{aligned}\mathcal{B} = \Big\{&\{1,3,4\},\{1,2,4\},\{1,2,3\},\\ &\{3,4,5,6\},\{2,4,5,6\},\{2,3,5,6\},\{2,3,4,6\},\{2,3,4,5\},\\ &\{1,4,5,6\},\{1,3,5,6\},\{1,2,5,6\}\Big\}\end{aligned}$$

We will recall the definition of Q^2 later, but for now notice that $B_1, B_2 \in \mathcal{B}$ implies that $B_1 \cap B_2 \neq \varnothing$, which suffices to show the structure is Q^2. A multiplication of secrets in the passively secure protocol of Maurer requires all parties to produce one secret sharing, thus sending a total of

$$\sum_{i=1}^{n}\left(\sum_{B\in\mathcal{B},B\ni i}(|B|-1)+\sum_{B\in\mathcal{B},B\not\ni i}(|B|)\right)=\sum_{i=1}^{n}\left(\sum_{B\in\mathcal{B}}|B|-\sum_{B\in\mathcal{B},B\ni i}1\right)$$

finite field elements (when using replicated sharing for this access structure) over $n\cdot(n-1)$ uni-directional secure channels. In our example this translates into

sending $(41-6)+(41-7)+(41-7)+(41-7)+(41-7)+(41-7) = 205$ finite field elements over $6 \cdot 5 = 30$ secure channels. Note that the same finite field element will be sent to multiple parties (every set of parties B obtains a share common to them all), but we count these elements as distinct when analysing communication costs.

In our protocol we partition $\mathcal{B}$ into subsets $\{\mathcal{B}_i\}_{i\in\mathcal{P}}$, such that the sets $\mathcal{B}_i$ form a non-trivial partition $\mathcal{B}$ (i.e. $\mathcal{B}_i \neq \emptyset$), and for all $B \in \mathcal{B}_i$ we have $i \in B$. In our example we set

$$\begin{aligned}
\mathcal{B}_1 &= \{\{1,3,4\}\},\\
\mathcal{B}_2 &= \{\{1,2,4\}\},\\
\mathcal{B}_3 &= \{\{1,2,3\}\},\\
\mathcal{B}_4 &= \{\{2,3,4,5\}\},\\
\mathcal{B}_5 &= \{\{1,2,5,6\},\ \{1,3,5,6\},\ \{1,4,5,6\}\},\\
\mathcal{B}_6 &= \{\{2,3,4,6\},\ \{2,3,5,6\},\ \{2,4,5,6\},\ \{3,4,5,6\}\}.
\end{aligned}$$

We will prove, for access structures in which all players have some part to play in the MPC protocol, that such a partition always exists. We will also discuss the fact that for some access structures, it is not necessary for all parties to be involved in the MPC (barring any input from the said parties). Given this partition our (passively secure) protocol makes use of a set

$$\mathsf{SC} = \bigcup_{i\in\mathcal{P}} \bigcup_{B\in\mathcal{B}_i} \bigcup_{j\in B\setminus\{i\}} \{(i,j)\}$$

of secure channels, where $(i,j) \in \mathsf{SC}$ implies that party i is connected to party j by a uni-directional secure channel (following good practice we assume all channels are uni-directional). For each multiplication we need to send

$$\sum_{B\in\mathcal{B}} (|B|-1)$$

finite field elements. In our example we have

$$\begin{aligned}
\mathsf{SC} = \Big\{&(1,3),(1,4),(2,1),(2,4),(3,1),(3,2),(4,2),\\
&(4,3),(4,5),(5,1),(5,2),(5,3),(5,4),(5,6),\\
&(6,2),(6,3),(6,4),(6,5)\Big\}.
\end{aligned}$$

Thus in this example we need to send 30 finite field elements over 18 (uni-directional) secure channels per multiplication operation, thus giving a saving of 85 percent on the number of finite field elements it is necessary to transmit, and 40 percent on the number of secure channels needed.

We then extend this basic protocol to the case of active security (with abort); again with the objective of minimizing the number of pairwise connections. Our actively secure protocol again follows the paradigm of Araki et al. However, we

need to make a small set of changes to allow for arbitrary Q^2 access structures. Like Araki et al. we use our passively secure multiplication protocol in an offline phase over SC, the set of secure channels, to obtain so-called Beaver triples. These triples are then checked, using the usual trick of sacrificing (see e.g. [BDOZ11]).

The triples are then used in an online phase, but, unlike Araki et al., we use a standard Beaver-like online phase which is executed over an incomplete network of *authenticated* channels. In particular when using replicated secret sharing for an arbitrary Q^2 access structure our set of authenticated channels is given by

$$\mathsf{AC} = \bigcup_{i \in \mathcal{P}} \bigcup_{B \in \mathcal{B}_i} \bigcup_{j \notin B} \{(i,j)\},$$

since publicly opening a secret requires every party to receive every share it doesn't have from at least one other party, which can be done efficiently in the semi-honest protocol using the partition assignment. In our running example this set is given by

$$\begin{aligned}\mathsf{AC} = \Big\{&(1,2),(1,5),(1,6),(2,3),(2,5),(2,6),(3,4),\\ &(3,5),(3,6),(4,1),(4,6),(5,2),(5,3),(5,4),\\ &(6,1),(6,2),(6,3),(6,3),(6,5)\Big\}.\end{aligned}$$

We denote by SC^* the same topology but with secure, instead of authenticated, channels. Each online multiplication in our active online protocol will require a total of

$$\sum_{i \in \mathcal{P}} \sum_{B \in \mathcal{B}_i} (n - |B|) = n \cdot |\mathcal{B}| - \sum_{i \in \mathcal{P}} \sum_{B \in \mathcal{B}_i} |B| = n \cdot |\mathcal{B}| - \sum_{B \in \mathcal{B}} |B|$$

finite field elements need to be sent over these channels. Which in our example equates to $6 \cdot 11 - (3 \cdot 3 + 8 \cdot 4) = 25$ finite field elements, over 19 authentic channels.

Active security is obtained, as in [AFL+16], by hashing players' views and comparing the resulting hashes at the end (which will require a complete set of authentic channels). However, in generalising to arbitrary access structures it is no longer sufficient to hash the view of the opened value: one also needs to hash the shares used to produce this value. This is because each player's view is incomplete, and one cannot rely (as Araki et al. do) on there being at least one honest player receiving data only from honest players, which would otherwise suffice. To make clear what channels are required when, and how many, we provide Table 1

It should be noted that our online phase methodology can actually be executed using other secret sharing schemes, assuming the Beaver triples in the offline phase are produced with respect to this secret sharing scheme. In particular in the threshold case it turns out that we would obtain, using Shamir sharing, an online phase which only requires $n \cdot t$ authenticated channels, as opposed to $n \cdot (n-1)$ authenticated channels using the naïve protocol.

Protocol	Procedure	Channels required
Passive Protocol	Input	SC
	Multiplication	SC
	Output to one	Complete secure
	Output to all	Complete authenticated
Active Offline Protocol	Triple Gen.	SC
	Triple Sac.	AC
	Authentication check	Complete authenticated
Active Online Protocol	Input	SC^* + Complete authenticated
	Multiplication	AC
	Output to one	Complete authenticated + SC^*
	Output to all	Complete authenticated + AC

Table 1. Number of channels needed at each point in the computation. The channels for "Output to one" assumes every party will receive private output. Notice that the active variant of our protocol never needs a complete network of secure channels and that it only requires a complete authenticated network for the hash-comparison stage only.

In this paper we are interested in evaluation of arithmetic circuits over an arbitrary finite field $\mathbb{F}_q$, which could include $q = 2$. We will assume that q is sufficiently large to have a cheating detection probability of $1/q$, but if this is not the case, a simple repetition of the checking procedure will reduce the cheating probability to whatever is required of an application. We do not analyse this aspect in this paper so as to aid the reader in seeing the main concepts more fully. This repetition and its generalisation to balls-and-bins experiments is relatively standard.

2 Preliminaries

In this section we recap on access structures, and in particular Q^2 access structures, and also look at pseudo-random zero sharings with respect to the additive secret sharing scheme. We then describe the traditional MPC protocol of Maurer to evaluate the multiplication gates. In this section we are working over an arbitrary finite field $\mathbb{F}_q$.

2.1 General Secret Sharing

Access Structures: Suppose we have a set of parties $\mathcal{P} = \{1, \ldots, n\}$. Let (Γ, Δ) be an access structure on them: Γ and Δ are subsets of $2^{\mathcal{P}}$ where the sets in Γ are sets of parties qualified to construct the secret, and the sets in Δ the sets of unqualified parties. The access structure is said to be *complete* if $\Gamma \cup \Delta = 2^{\mathcal{P}}$, (i.e. every set of parties is either qualified or unqualified). The structure is said to be *monotone* if Γ is closed under taking supersets (i.e., given a set of qualified parties, adding any other party to this set gives us another set of qualified parties)

and Δ is closed under taking subsets (i.e. any subset of unqualified parties is also unqualified).

A set of parties in Δ is called maximally unqualified if by colluding with any other party not in the set, the parties can together recover the secret. We denote by $\mathcal{M} \subset \Delta$ the set of maximally unqualified parties. A set in Γ is called minimally qualified if it is qualified and every proper subset is unqualified. The set $\mathcal{M}$ and its structure is important for our protocol; however, it will be notationally simpler for us instead to consider the set of complements of maximally unqualified sets, which we denote by $\mathcal{B} = \{M^{\mathrm{C}} \in 2^{\mathcal{P}} : M \in \mathcal{M}\}$.

Given an arbitrary $\mathcal{X} \subset 2^{\mathcal{P}}$, we say that $\mathcal{X}$ is *valid* if the following two conditions hold:

- If $S \subset \mathcal{P}$ then there exists a set $X \in \mathcal{X}$ such that either $X \subseteq S$ or $S \subseteq X$.
- If $X_1, X_2 \in \mathcal{X}$ then $X_1 \not\subset X_2$.

Now suppose we are given a valid set of sets $\mathcal{M}$ on a set of parties $\mathcal{P}$. If we take this to be a set of qualified sets, then because $\mathcal{M}$ is valid, these are maximally unqualified sets: the first property defines which sets in $2^{\mathcal{P}}$ are unqualified (and therefore determines Δ for some access structure), and the second ensures the sets are maximal with respect to set inclusion; thus $\mathcal{M}$ is the set of maximally unqualified sets any access structure whose unqualified sets are given by Δ. Assuming the access structure is complete, $\mathcal{M}$ also entirely determines the qualified sets, $\Gamma (= 2^{\mathcal{P}} \setminus \Delta)$, and thus the access structure (Γ, Δ).

Given a valid set of sets $\mathcal{M}$ over $\mathcal{P}$ and a player $i \in \mathcal{P}$, we define $\mathcal{M}^{(i)}$ as the set of sets over $\mathcal{P} \setminus \{i\}$ which are just the sets of $\mathcal{M}$ after removing player i (if necessary): $\mathcal{M}^{(i)} = \{M \setminus \{i\} : M \in \mathcal{M}\}$. For an access structure (Γ, Δ) given by $\mathcal{M}$ we call a player i *redundant* if the set $\mathcal{M}^{(i)}$ is also valid. A non-redundant access structure is one in which there are no redundant players. A redundant player is one whose shares are not *necessarily* needed to reconstruct the secret, and so one could define an MPC protocol achieving the same (passive) security by ignoring this player entirely in the computation. For example, consider the valid but redundant set $\mathcal{M} = \{\{1\}, \{2\}, \{3, 4\}\}$. We obtain the replicated scheme over this access structure, by computing $\mathcal{B} = \{\{2, 3, 4\}, \{1, 3, 4\}, \{1, 2\}\}$ and splitting a secret s into three shares $s = s_{234} + s_{134} + s_{12}$; then we give player 1 the shares $\{s_{134}, s_{12}\}$, player 2 $\{s_{234}, s_{12}\}$, player 3 $\{s_{234}, s_{134}\}$ and player 4 $\{s_{234}, s_{134}\}$. Everything player 3 can do can also be done by player 4, so we can essentially ignore player 4 in any protocol design and just provide the output to this player at the end.

Non-redundant access structures play a crucial role in our protocol due to the following theorem

Theorem 1. *Given a non-redundant access structure (Γ, Δ) derived from a valid set $\mathcal{M}$, the set of complements $\mathcal{B}$ of elements in $\mathcal{M}$ can be partitioned into sets $\{\mathcal{B}_i\}_{i \in \mathcal{P}}$ such that $\forall i \in \mathcal{P}$ we have $\mathcal{B}_i \neq \varnothing$, and $B \in \mathcal{B}_i$ implies $i \in B$.*

Proof. We give a constructive proof which produces the partition. Suppose we start with a non-redundant valid set $\mathcal{M}$ of maximally unqualified sets. We proceed as follows to define a map from $\mathcal{B}$ to $\mathcal{P}$. We note that if $\mathcal{M}$ consists of

singletons then assigning $\mathcal{B}$ to $\mathcal{P}$ is trivial: since $\mathcal{M}$ is non-redundant, every party appears in a singleton set; then for each $\{i\} \in \mathcal{B}$ we choose any $j \in \mathcal{P}$ with $j \neq i$ and add the set $\{i\}$ to $\mathcal{B}_j$. We therefore proceed inductively, simplifying our structure at each inductive step by removing players until we get down to the case of a set of singletons.

Now assume $\mathcal{M}$ contains a non-singleton set. Pick the largest one M and select a party $i \in M$ which is not contained in any singleton in $\mathcal{M}$. Such an i must exist as otherwise the set $\mathcal{M}$ is invalid (since it fails the second validity property).

Now form the set $\mathcal{M}^{(i)}$ as above. (Recall that $\mathcal{M}^{(i)} = \{M \setminus \{i\} : M \in \mathcal{M}\}$.) The set of sets $\mathcal{M}^{(i)}$ will be invalid, since by definition $\mathcal{M}$ was non-redundant. Let $\mathcal{B}^{(i)}$ denote the associated complements. Note that each set in $\mathcal{B}^{(i)}$ corresponds uniquely to a set in $\mathcal{B}$ (and via the recursion to a set at the top level as well). Thus we can use $\mathcal{B}^{(i)}$ to define an assignment of a subset of the original sets $\mathcal{B}$ to player i.

We now delete sets from $\mathcal{M}^{(i)}$ until we obtain a valid set of sets again. The process for performing this deletion is as follows. We start with $\mathcal{M}' = \mathcal{M}^{(i)}$ and repeat the following process.

- If $\mathcal{M}' = \varnothing$ then we stop.
- As $\mathcal{M}'$ is not valid we must have two sets $M'_1, M'_2 \in \mathcal{M}'$ with $M'_1 \subset M'_2$.
- The value i used to form $\mathcal{M}^{(i)}$ must have been deleted from either M'_1 or M'_2 or both, since $\mathcal{M}$ was valid.
- We must have had that i was deleted from M'_1 only, since otherwise the original set $\mathcal{M}$ would have been invalid.
- We now delete the set M'_2 from $\mathcal{M}'$, find the original associated set B in $\mathcal{B}$, and add this set to $\mathcal{B}_i$. Note this means we never delete singleton sets from $\mathcal{M}'$, as we always delete the bigger set. Also note that the B contains i as M'_2 did not contain i.

We can then recurse, assuming the new set $\mathcal{M}'$ output by this procedure is non-redundant. To prove $\mathcal{M}'$ is non-redundant, we must show that removing any party creates an invalid set of sets. The first property of validity will hold by construction: for $\mathcal{M}$, we have that for every set $S \subseteq \mathcal{P} \setminus \{i\} \subseteq \mathcal{P}$, there exists a set $M \in \mathcal{M}$ such that $S \subseteq M$ or $M \subseteq S$. Then $S \subseteq M$, and so $S \subseteq M \setminus \{i\}$ since $i \notin S$, and alternatively if $M \subseteq S$ we trivially have $M \setminus \{i\} \subseteq S$, and $M \setminus \{i\} \in \mathcal{M}'$ and is non-empty since $M \neq \{i\}$ by construction.

So for $\mathcal{M}'$ to be non-redundant we must show failure of the second property of validity, i.e. that for any $j \in \mathcal{P} \setminus \{i\}$, there exist sets $M'_1, M'_2 \in \mathcal{M}'$ such that $j \in M'_1$ and $M'_1 \setminus \{j\} \subseteq M'_2$.

Fix $j \in \mathcal{P} \setminus \{i\}$. By construction we know that $\{i\} \notin \mathcal{M}$, and that $\mathcal{M}$ is non-redundant. Since $\mathcal{M}$ is non-redundant, there exist sets $M_1, M_2 \in \mathcal{M}$ such that $j \in M_1$ and $M_1 \setminus \{j\} \subset M_2$. Then $\mathcal{M}'$ contains the sets $M'_1 \leftarrow M_1 \setminus \{i\}$ and $M'_2 \leftarrow M_2 \setminus \{i\}$ (which are non-empty since $\{i\} \notin \mathcal{M}$) and we have that $M'_1 \setminus \{j\} \subset M'_2$. Player $j \in \mathcal{P} \setminus \{i\}$ was chosen arbitrarily, so this holds for all $j \in \mathcal{P} \setminus \{i\}$. Thus $\mathcal{M}'$ is non-redundant.

We can therefore perform the recursion until all sets in $\mathcal{B}$ are assigned to sets in $\{\mathcal{B}_i\}_{i\in\mathcal{P}}$. □

Q^ℓ Access Structures: The set Δ, called the adversary structure, is said to be Q^ℓ (for **q**uorum) if no ℓ sets in Δ cover $\mathcal{P}$. A result of Hirt and Maurer [HM00] says that every function can be computed securely in the presence of an adaptive, passive (resp. adaptive, active) computationally unbounded adversary if and only if the adversary structure is Q^2 (resp. Q^3).

It is clear that if Δ is Q^2, then so is any subset. In particular, the set of maximally unqualified sets $\mathcal{M}$ is also Q^2, Hence, for the set of complements $\mathcal{B}$ it holds that if $B_1, B_2 \in \mathcal{B}$ then $B_1 \cap B_2 \neq \varnothing$. A set $\mathcal{B}$ for which this property holds was named a quorum system by Beaver and Wool [BW98].

Let S denote a linear secret sharing scheme which implements the Q^2 access structure (Γ, Δ). We use double square brackets, $[\![v]\!]$ to denote a sharing of the secret v according to this scheme. We let $S_{v,i}$ denote the set of elements which player i holds in representing the value v. Another characterisation of Q^2 is as precisely the set of access structures which can be realised by multiplicative secret sharing schemes, i.e. given two secret shared values $[\![a]\!]$ and $[\![b]\!]$ then the product $a \cdot b$ can be represented a linear combination of the elements in the local Schur products

$$S_{a,i} \otimes S_{b,i} = \{s_a \cdot s_b : s_a \in S_{a,i}, s_b \in S_{b,i}\}.$$

This property allows us to build an MPC protocol secure against passive adversaries for any Q^2 access structure.

2.2 Replicated Secret Sharing:

Given a monotone access structure (Γ, Δ), we will make extensive use of the replicated secret sharing scheme which implements this structure. Let $\mathcal{B}$ be, as above, the set of sets which are complements of maximally unqualified sets. Then to share secret x, we write $x = \sum_{B\in\mathcal{B}} x_B$ and give x_B to player i if $i \in B$. From now on, when writing $[\![x]\!]$ we will mean the secret sharing with respect to this scheme, and in particular the set $S_{x,i}$ above is given by $S_{x,i} = \{x_B : i \in B \text{ and } B \in \mathcal{B}\}$.

The replicated scheme is linear (i.e. the parties can obtain a sharing of any linear function under the same secret sharing scheme by local computations) and is multiplicative if the access structure is Q^2: given secrets x and y, for every pair sets $B_1, B_2 \in \mathcal{B}$ there is some party i in $B_1 \cap B_2$, since the intersection of these sets is non-empty. Then party i can compute the terms $x_{B_1} \cdot y_{B_2}$ and $x_{B_2} \cdot y_{B_1}$ (and also $x_{B_1} \cdot y_{B_1}$ and $x_{B_2} \cdot y_{B_2}$). Thus the parties can together obtain all terms of $x \cdot y = \left(\sum_{B\in\mathcal{B}} x_B\right) \cdot \left(\sum_{B\in\mathcal{B}} y_B\right) = \sum_{B_1,B_2\in\mathcal{B}} x_{B_1} \cdot y_{B_2}$ by local computations. Note that the parties do not, in general, have a correct sharing of the product after these local computations, since each party now holds only one share; the parties must somehow convert this additive share of the product into a sharing within the scheme. Minimising the number of communication channels required after the local computations is the main goal of this paper.

2.3 Pseudo-Random Zero Sharing for Additive Secret Sharing Schemes

At various points we will need to use an additive secret sharing over all players $\mathcal{P} = \{1, \ldots, n\}$. This shares a value $v \in \mathbb{F}_q$ as an additive sum $v = \sum_{i=1}^{n} v_i$ and gives player i the value v_i. We denote such a sharing by $\langle v \rangle$. It is obvious that such a sharing is not Q^2, but it will play a crucial role in our protocols.

Improving on the protocol of [BW98] and [Mau06] requires us to sacrifice the information-theoretic security for a cryptographic assumption. In particular, we require the parties to engage in a pre-processing phase in which they share keys for a pseudo-random function (PRF) in order to generate (non-interactively) pseudo-random zero sharings (PRZS) for the additive secret sharing scheme $\langle v \rangle$, and pseudo-random secret sharings (PRSS) for the replicated scheme $[\![v]\!]$[1]. In particular we wish to implement the functionality given in Figure 1.

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

PRZS(cnt):
- On input of a counter value cnt from each party, if the counter value is the same for all parties and has not been used before, for each $i \in [n-1]$ the functionality samples $w_i \xleftarrow{\$} \mathbb{F}$ uniformly at random, fixes $w_n \leftarrow -\sum_{i \in [n-1]} w_i$ and sends w_i to party i for each $i \in \mathcal{P}$.

PRSS(cnt):
- On input of a counter value cnt from each party, 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 $i \in B$.

Figure 1. The Functionality $\mathcal{F}_{\mathsf{Rand}}$

Pseudo-random secret sharing, and pseudo-random zero sharings in particular, for arbitrary access structures can involve a costly setup phase in general [CDI05]. However, for the simple additive secret sharing scheme it is relatively easy to construct a non-interactive method for producing PRZSs. In our situation each party i shares a secret key $\kappa_{i,j}$ with each party $j \neq i$. The secret keys are assumed to lie in $\{0,1\}^\lambda$, which is the keyspace of a pseudo-random function F with codomain our finite field $\mathbb{F}_q$. The set-up procedure, and the method to generate the PRZS and PRSS is given in Figure 2.

Theorem 2. *Assuming a trusted set-up and that F is a pseudo-random function, the protocol Π_{Rand} securely realises $\mathcal{F}_{\mathsf{Rand}}$ against active adversaries.*

[1] We could produce these using additional interaction, but recall our goal is to reduce communication.

Protocol Π_{Rand}

Set-up: The parties initialise by doing the following:

1. For each $i \in \mathcal{P}$, for each $j \in \mathcal{P} \setminus \{i\}$, party i samples $\kappa_{i,j} \xleftarrow{\$} \{0,1\}^\lambda$.
2. For each $i \in \mathcal{P}$, for each $j \in \mathcal{P} \setminus \{i\}$, party i sends $\kappa_{i,j}$ to party j.
3. For each $B \in \mathcal{B}$, each party $i \in B$ samples $\kappa_{B,i} \xleftarrow{\$} \{0,1\}^\lambda$
4. For each $B \in \mathcal{B}$, each party $i \in B$ sends $\kappa_{B,i}$ to all parties $j \in B \setminus \{i\}$.
5. Each party i sets $\kappa_B \leftarrow \oplus_{i \in B} \kappa_{B,i}$ for each B containing i.

$\mathsf{PRZS}(\mathsf{cnt})$: Party i computes

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

$\mathsf{PRSS}(\mathsf{cnt})$: For each $B \in \mathcal{B}$ containing i, party i computes

$$r_B \leftarrow F_{\kappa_B}(\mathsf{cnt}).$$

Figure 2. Protocol Π_{Rand}

Proof. As there is no interaction after **Set-up**, the protocol is clearly actively secure, if it is correct and passively secure. Correctness follows from basic algebra, and security follows from the fact that F is assumed to be a PRF and from the fact that there is at least one B not held by the adversary (by definition of the access structure. □

2.4 Maurer's MPC-made-Simple Protocol

The information theoretic protocol upon which our protocol is based is the one due to Maurer [Mau06]. Maurer's protocol is itself a variant of the protocol of Beaver and Wool [BW98] but specialised to the case of replicated secret sharing. By means of comparison, we explain Maurer's protocol here: We first assume a Q^2 access structure (Γ, Δ), and we share data values x via the replicated secret sharing $[\![x]\!]$, where $x = \sum_{B \in \mathcal{B}} x_B$. Since this secret sharing scheme is linear, addition of secret shared values comes "for free", i.e. it requires no interaction and parties just need to add their local shares together.

The real difficulty in creating an MPC protocol given a linear secret sharing scheme is in performing secure multiplication of secret shared values, $[\![x]\!]$ and $[\![y]\!]$. With this goal, we begin by following [BW98] and define a *surjective* function $\rho : \mathcal{B}^2 \to \mathcal{P}$ such that $\rho(B_1, B_2) = i$ implies that $i \in B_1 \cap B_2$; the existence of such a function follows from the fact that the access structure is Q^2. Note that party $\rho(B_1, B_2)$ holds a copy of share x_{B_1} and y_{B_2}. We will put player $\rho(B_1, B_2)$ "in charge" of computing the cross term $x_{B_1} \cdot y_{B_2}$ in the following multiplication protocol:

1. Party i computes

$$v_i \leftarrow \sum_{\rho(B_1,B_2)=i} x_{B_1} \cdot y_{B_2}$$

2. Party i creates a sharing $[\![v_i]\!]$ of the value v_i and distributes the different summands securely to the appropriate parties according to the replicated secret sharing scheme.
3. The parties now locally compute

$$[\![z]\!] \leftarrow \sum_{i=1}^{n} [\![v_i]\!].$$

It is clear that each party i, in sharing v_i, needs to generate $m = |\mathcal{B}|$ different finite field elements, each of which is sent to every member of a given set of parties in $\mathcal{B}$. In particular this means that each party has to maintain a secure connection to each other party, assuming a non-redundant access structure. If we let l denote the average size of $B \in \mathcal{B}$, i.e. $l = \sum_{B \in \mathcal{B}} |B|/m$, then it is clear that the total communication required is $n \cdot m \cdot l$ finite field elements.

Our protocol is largely the same, except that the parties do not create a replicated sharing of the partial product v_i. Notice that the v_i form an additive sharing $\langle z \rangle$ of the sum. Our basic idea is first to re-randomize this sum using the PRZS scheme, and then to consider the re-randomized v_i as one share of the product, i.e. z_B indexed by some B containing i, which should then be distributed to all other parties in B. (There are some minor technical caveats but this is the essential idea.) This replaces the creation of n replicated shares and summing them in Maurer's protocol, and means that each party does not need to be connected to each other party by a secure channel. The total number of distinct finite field elements transmitted in a threshold scheme is $O(n \cdot 2^n)$, as opposed to the $O(n^2 \cdot 2^n)$ of Maurer's protocol, as we shall see in the next section. For other Q^2 structures the saving in communication is more significant, as our earlier example demonstrates. Our method directly generalizes the method used by [AFL+16], which concentrated on the case of the finite field $\mathbb{F}_2$ and a 1-out-of-3 adversary structure.

3 Passively Secure MPC Protocol

In this section we outline our optimization of Maurer's protocol. As remarked earlier, our protocol, instead of being in the information theoretic model, uses PRFs to obtain additive sharings of zero non-interactively. We assume throughout that we start with an access structure which does not contain any redundant players. Thus we can define a partition $\{\mathcal{B}_i\}$ of $\mathcal{B}$ such that $\mathcal{B}_i \neq \varnothing$ and $B \in \mathcal{B}_i$ implies $i \in B$. We consider $\mathcal{B}_i$ to be the set of sets for which party i will be "responsible".

As in Maurer's MPC-Made-Simple protocol, we assume a Q^2 access structure (Γ, Δ) and share data values x via the replicated secret sharing $[\![x]\!]$, so that $x = \sum_{B \in \mathcal{B}} x_B$. We also retain the assignment which tells player $i = \rho(B_1, B_2)$ to compute the product $x_{B_1} \cdot y_{B_2}$. However, our basic multiplication procedure is given by:

1. Party i computes
$$v_i \leftarrow \sum_{\rho(B_1,B_2)=i} x_{B_1} \cdot y_{B_2}$$
We think of v_i as an additive sharing $\langle v \rangle$ of the product.
2. The parties obtain an additive sharing of zero $\langle t \rangle$ using the PRZS from earlier, thus party i holds t_i such that $\sum_{i=1}^{n} t_i = 0$.
3. Party i samples z_B for $B \in \mathcal{B}_i$ such that $\sum_{B \in \mathcal{B}_i} z_B = v_i + t_i$.
4. Party i sends, for all $B \in B_i$, the value z_B to party j for all $j \in B$.

Notice that the parties do not need to perform local computations after the communication as in Maurer's protocol, and that the total number of elements transmitted is $\sum_{B \in \mathcal{B}}(|B| - 1)$.

The key observation for security is that the PRZS masks the Schur product terms, so after choosing the z_B's and sending these to the appropriate parties, not even qualified sets of parties can learn any information about these terms, despite the secret being reconstructible by qualified sets of parties.

Passively Secure MPC Functionality $\mathcal{F}_{\mathsf{PMPC}}$

Input: On input (Input, x_i) by party i, the functionality stores (id, x_i) in memory.

Add: On input $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$, the functionality retrieves (id_1, x) and (id_2, y) and stores $(\mathsf{id}_3, x + y)$.

Multiply: On input $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$, the functionality retrieves (id_1, x) and (id_2, y) and stores $(\mathsf{id}_3, x \cdot y)$.

Output: On input $(\mathsf{Output}, \mathsf{id}, i)$ from all parties, the functionality retrieves (id, x) and returns x to all parties if $i \neq 0$, and to player i only otherwise.

Figure 3. Passively Secure MPC Functionality $\mathcal{F}_{\mathsf{PMPC}}$

Given this informal description we now give a full description of our MPC protocol, which is the analogue of [AFL+16] for arbitrary Q^2 access structures and abitrary finite fields; see Figure 4 for details. One can think of the passively secure protocol as being in the pre-processing model in which the offline phase simply involves some key agreement. The online phase is then a standard MPC protocol in which parties can compute an arithmetic circuit on their combined (secret) inputs, using the multiplication procedure described above, so as to implement the functionality in Figure 3. That the protocol securely implements this functionality is given by the following theorem, whose proof is given in Appendix A.

Theorem 3. *Suppose we have a non-redundant Q^2 access structure with a set $\mathcal{B}$ defined as above. Then the protocol Π_{PMPC} securely realises the functionality $\mathcal{F}_{\mathsf{PMPC}}$ against passive adversaries in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model.*

The protocol requires at most $\sum_{B\in\mathcal{B}}(|B|-1)$ *field elements of communication, over* $|\mathsf{SC}|$ *secure channels, per multiplication gate, and the same number to perform the input procedure. In the output procedure we require that the parties be connected by a complete network of bilateral secure channels (i.e.* $n\cdot(n-1)$ *uni-directional channels) if all players are to receive distinct private outputs, and instead a complete network of authenticated channels if only public output is required.*

Note, the above theorem is given for non-redundant access structures. To apply the protocol in the case of redundant access structures, we simply remove redundant players from the computation phase and only require interaction with them in the input and output phases. To avoid explaining this (trivial) extra complication we specialise to the case of non-redundant access structures.

Protocol Π_{PMPC}

The set $\mathcal{B}_i$ denotes the set of the partition $\mathcal{B}=\{\mathcal{B}_i\}_{i\in\mathcal{P}}$ containing sets associated to party i (though note that it is a -usually strict- subset of the sets containing i) and constructed via proof of Theorem 1.

Input: For party i to provide input x,

1. The parties call $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}$ so that each player $j\in\mathcal{P}$ obtains t_j such that $\sum_{i\in\mathcal{P}} t_j = 0$.
2. Party i samples $\{u_B\}_{B\in\mathcal{B}_i}\leftarrow\mathbb{F}$ such that $\sum_{B\in\mathcal{B}_i} u_B = x + z_i$.
3. For each $j\in\mathcal{P}\setminus\{i\}$, party j samples $\{u_B\}_{B\in\mathcal{B}_j}\leftarrow\mathbb{F}$ such that $\sum_{B\in\mathcal{B}_j} u_B = z_j$.
4. For all $j\in\mathcal{P}$, for each $B\in\mathcal{B}_j$, party j sends u_B securely to party k for all $k\in B$.

Add:

1. For each $B\in\mathcal{B}$, each party $i\in B$ locally computes x_B+y_B so that collectively the parties obtain $[\![x+y]\!]$.

Multiply:

1. For each $i\in\mathcal{P}$, party i computes $v_i\leftarrow\sum_{\rho(B_1,B_2)=i} x_{B_1}\cdot y_{B_2}$.
2. The parties call $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}$ so that each player $i\in\mathcal{P}$ obtains t_i such that $\sum_{i\in\mathcal{P}} t_i = 0$.
3. For each $i\in\mathcal{P}$, party i samples $\{u_B\}_{B\in\mathcal{B}_i}\leftarrow\mathbb{F}$ such that $\sum_{B\in\mathcal{B}_i} u_B = v_i + t_i$.
4. For each $i\in\mathcal{P}$, party i sends, for all $B\in\mathcal{B}_i$, the value u_B securely to party j for all $j\in B\setminus\{i\}$.

Output$([\![x]\!],i)$:

1. If $i\neq 0$, each player j securely sends x_B to i if $i\notin B$, for all $B\in\mathcal{B}_j$. If $i=0$, each player j instead sends to *all* players i for which $i\notin B$. In the latter case the communication need not be done securely.
2. Player i (or all players if $i=0$) computes $x\leftarrow\sum_{B\in\mathcal{B}} x_B$.

Figure 4. Protocol Π_{PMPC}

We end this section by examining our protocol in the inefficient (for us) but interesting case of threshold access structures. For an (n, t)-threshold scheme, each $B \in \mathcal{B}$ has size $n-t$, and there are $\binom{n}{t}$ sets in total, so the total number of elements transmitted is exactly $\binom{n}{t} \cdot (n-t-1)$. If t is expressed as a constant fraction of n, then $\binom{n}{t}$ is (asymptotically) exponential in n, so this has complexity $O(n \cdot 2^n)$. This compares favourably with Maurer's protocol which has complexity $O(n^2 \cdot 2^n)$, because in that protocol the parties effectively make n shares and add them together instead of creating one between them as in the new protocol. It was observed by Beaver and Wool [BW98] that the real overhead in communication depends on the size of $\mathcal{B}$, which grows exponentially in n for threshold schemes if the threshold is expressed as a constant fraction of n, and therefore it is desirable to construct schemes which are oblivious to this parameter. In the threshold case, both Maurer's protocol and ours are highly inefficient in terms of the number of finite field elements transmitted when compared to Shamir sharing.

Note, however, that the main goal of this paper is to reduce the number of communication channels required to perform the computation. For an (n, t)-threshold access structure, Maurer's protocol (and the standard Shamir-based protocol) requires $n \cdot (n-1)$ uni-directional secure channels since every party sends to and receives from every other party; for our passive protocol, we will still have every party connected to every other party, but for every set in $\mathcal{B}$ a party is in, it will either receive or send, but not both. Thus the number of secure channels is exactly half, $\frac{1}{2} \cdot n \cdot (n-1)$.

For non-threshold Q^2 access structures, since the previous best protocol was that of Maurer, we obtain also a more efficient protocol in terms of number of finite field elements transmitted, and comes at the expense of our (limited) use of cryptographically secure PRFs to set up the correlated randomness.

4 Maliciously Secure MPC Protocol

In this section we show how to realise an actively secure variant of $\mathcal{F}_{\mathsf{PMPC}}$ in the standard SPDZ-like fashion – with a pre-processing phase and an online phase, for our Q^2 access structures. Again, we take inspiration from the technique used in a restricted setting in [FLNW16]. In particular, we show how to achieve malicious security *without* using MACs, and how the required communication channels can be reduced for general access structures.

The offline phase uses our passively secure protocol for multiplication to produce so-called *Beaver triples*. These are then checked for correctness using a sacrificing step. The online phase then proceeds by the standard Beaver methodology of opening values to players. Note this is unlike the method in [FLNW16] where the online multiplication protocol uses the passively secure multiplication protocol, and then checks this is correct using a Beaver triple. The traditional method is conceptually easier, and means that the online protocol (bar outputting of data privately to one party) may be executed over authenticated, as opposed to secure, channels.

Furthermore, we reduce the number of authenticated channels required for multiplication by replacing the traditional Beaver "broadcast" phase with an "opening agreement" phase which requires fewer authenticated channels. This agreement on an open value is possible because we have a Q^2 access structure and replicated secret sharing. This means that every share must be held by at least one honest party. Checking of consistency is then done by hashing all the publicly opened shares (not just the opened secrets they reconstruct to), and then parties comparing their hashes at various points in the protocol. In [FLNW16] a similar method is used to maintain consistency by just hashing the opened values. This, however, only works for limited access structures and communication topologies (though note that MPC based on multiplicative secret sharing inherently requires that the access structure be Q^2).

To explain our method we use the standard hash API of Init, Update and $\mathsf{Finalise}$; so that to execute $h = H(m_1 \| m_2 \| \dots \| m_t)$ we actually execute the statements

$$H \leftarrow \mathsf{Init}(),\ \ H.\mathsf{Update}(m_1),\ \ H.\mathsf{Update}(m_2), \dots \\ \dots, H.\mathsf{Update}(m_t),\ \ h \leftarrow H.\mathsf{Finalise}().$$

The hash function is used to check views of various opened value as in the subprotocols defined in Figure 5. This protocol requires a complete network of authenticated channels to implement **CompareView**, and a set SC^* (defined in the introduction) of secure channels to implement $\mathbf{Reveal}([\![x]\!], i)$ for all $i \neq 0$.

Note that in $\mathbf{Reveal}([\![x]\!], i \neq 0)$ the reconstructed secret is guaranteed to be the correct value because, since the adversary structure is Q^2, each value x_B will be received from at least one honest party. Hence, if an adversary deviates then this is detected by the receiving party. A similar checking is obtained in $\mathbf{Reveal}([\![x]\!], 0)$, i.e. when a secret is opened to everyone, but instead via the hash function, since two honest parties will differ in their views if the adversary tries to deviate from the protocol.

We can now define our pre-processing functionality $\mathcal{F}_{\mathsf{Triple}}$ in Figure 6, and the protocol Π_{Triple} in Figure 7 which will implement this, which itself uses the opening subprotocols defined in Figure 5. We let $\mathcal{A}$ denote the set of corrupted players, and recall we assume (for simplicity) that our finite field $\mathbb{F}_q$ is chosen for a suitably large value of q, so that $1/q$ is negligible. That the protocol implements the functionality is given by the following theorem, whose proof we give in Appendix B.

Theorem 4. *The protocol Π_{Triple} securely realises $\mathcal{F}_{\mathsf{Triple}}$ in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model against static, malicious adversaries, assuming the hash function H is collision resistant, and the finite field size q is sufficiently large. The protocol requires $|\mathsf{SC}|$ secure channels to execute the passively secure multiplication protocol, which is at the core of the **Triple Generation** step, and $|\mathsf{AC}|$ authenticated channels to execute **Triple Sacrifice**. The steps in which the parties verify the hash values in **Opening Check** requires a full network of authenticated channels (over which only a single hash value per channel is sent).*

Subprotocol Π_{Open}

The set $\mathcal{P}$ is the set of parties, and the set $\mathcal{A} \subset \mathcal{P}$ the set of corrupt parties in $\mathcal{P}$. Recall that there is a partition $\mathcal{B} = \{\mathcal{B}_i\}_{i \in \mathcal{P}}$

Init: $H_i = \mathsf{Init}()$ for all players i.

Broadcast(i, ϵ): Party i sends ϵ to all players over authenticated channels. Each player $j \in \mathcal{P}$ then executes $H_j.\mathsf{Update}(\epsilon)$.

Reveal$([\![x]\!], i)$: If $i = 0$, the secret is opened to everyone, and otherwise is output only to party i.

- To open the secret $[\![x]\!]$ to party i,
  1. For each share x_B where $i \notin B$, every party $j \in B$ sends x_B to party i over a *secure channel*.
  2. If any two incoming shares differ, party i sets Abort to true.
  3. Otherwise player i computes the secret as $x \leftarrow \sum x_B$.
- To open the secret $[\![x]\!]$ to all parties,
  1. For every $j \in \mathcal{P}$, for every $B \in \mathcal{B}_j$, party j sends x_B to all $k \notin B$ over authenticated channels.
  2. Every party now holds x_B for all $B \in \mathcal{B}$ and so can recover the secret as $x \leftarrow \sum x_B$.
  3. Each party j calls $H_j.\mathsf{Update}(x_B)$ for all $B \in \mathcal{B}$.

CompareView:

1. Each party i executes $h_i \leftarrow H_i.\mathsf{Finalise}()$ and broadcasts the result.
2. The parties compare all the hashes they see; if any two hashes any party sees differ, they set Abort to true.
3. Each party i reinitialises the hash function, $H_i \leftarrow \mathsf{Init}()$.

Figure 5. Subprotocol Π_{Open}

Note that since there is at least one set which contains no corrupt parties, the functionality is able to sample shares indexed by such a $B \in \mathcal{B}$ with $B \cap \mathcal{A} = \varnothing$ so that the triple generated is correct regardless of what shares the adversary sent the functionality.

4.1 Actively Secure Protocol

We can now present our actively secure (with abort) online protocol, see Figure 9, which implements the functionality given in Figure 8, and uses the opening protocols defined in Figure 5. Note that a more elaborate input methodology is required to ensure actively secure input of values, compared to the passively secure protocol. The following theorem shows that the protocol implements the functionality, the proof we give in Appendix B.

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

This functionality will generate Beaver triples in batches of n_T at a time. Again recall that there is a partition $\mathcal{B} = \{\mathcal{B}_i\}_{i\in\mathcal{P}}$; we denote by $\mathcal{A}$ the indexing set of corrupt parties.

Triple: On input $\{\mathsf{Triple}, n_T\}$ the functionality proceeds as follows. If at any point the adversary signals abort then the functionality returns $\perp$ and terminates.

1. For $i = 1, \ldots, n_T$, the functionality does the following:
   (a) The functionality samples $\{a_B^{(i)}, b_B^{(i)}\}_{B\in\mathcal{B}} \leftarrow \mathbb{F}$ and sets
   $$c^{(i)} \leftarrow \left(\sum_{B\in\mathcal{B}} a_B^{(i)}\right) \cdot \left(\sum_{B\in\mathcal{B}} b_B^{(i)}\right).$$
   (b) The functionality sends the adversary the shares
   $$(\{a_B^{(i)}, b_B^{(i)} : B \in \mathcal{B} \text{ s.t. } k \in B\})_{k\in\mathcal{A}}.$$
   [These are all shares which the adversary has access to in sharing $a^{(i)}$ and $b^{(i)}$.]
   (c) The functionality uniformly samples a set $\bigcup_{j\notin\mathcal{A}}\{c_B^{(i)} : B \in \mathcal{B}_j \text{ s.t. } B\cap\mathcal{A} \neq \varnothing\} \leftarrow \mathbb{F}$ and sends $(\bigcup_{j\notin\mathcal{A}}\{c_B^{(i)} : B \in \mathcal{B}_j \text{ s.t. } k \in B\})_{k\in\mathcal{A}}$ to the adversary.
   (d) The functionality waits for the adversary to return a set of shares $(\bigcup_{k\in\mathcal{A}}\{c_{B,j}^{(i)} : B \in \mathcal{B}_k \text{ s.t. } j \in B\})_{j\notin\mathcal{A}}$.
   (e) If $c_{B,j_1} \neq c_{B,j_2}$ for any $j_1 \neq j_2$ then the flag Abort is set to be true.
   [These are all shares which the adversary controls during multiplication.]
2. If Abort is false the functionality samples $\{c_B^{(i)} : B \in \mathcal{B} \text{ s.t. } B\cap\mathcal{A} = \varnothing\}$ so that
   $$\left(\sum_{B\in\mathcal{B}} a_B^{(i)}\right) \cdot \left(\sum_{B\in\mathcal{B}} b_B^{(i)}\right) = \left(\sum_{B\in\mathcal{B}} c_B^{(i)}\right).$$
3. The functionality waits for the adversary to signal Abort or $\mathsf{Deliver}$. If it signals $\mathsf{Deliver}$, then for each $B \in \mathcal{B}$, $(a_B^{(i)}, b_B^{(i)}, c_B^{(i)})_{i=1}^{n_T}$ are sent to all honest players j for which $j \in B$. If the adversary instead signals Abort, or the Abort flag is true, then, for all $B \in \mathcal{B}$, the functionality sends $(a_B^{(i)}, b_B^{(i)}, c_B^{(i)})_{i=1}^{n_T}$ to the adversary and $\perp$ to all honest players.

Figure 6. Functionality $\mathcal{F}_{\mathsf{Triple}}$

Theorem 5. *The protocol Π_{Online} securely realises the functionality $\mathcal{F}_{\mathsf{AMPC}}$ against static, malicious adversaries for any non-redundant Q^2 access structure in the $\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Triple}}$-hybrid model, assuming H is collision resistant, and the finite field size q is sufficiently large.*

The protocol uses $|\mathsf{SC}|$ secure channels in the offline phase (i.e. for Π_{Triple}), and requires $|\mathsf{AC}|$ authenticated channels in the online phase for the multiplication operation. Inputting values requires $|\mathsf{SC}^*|$ secure channels and a complete network of authenticated channels, and output requires a complete network of authenticated channels for comparing views, and then to output a value privately

Protocol Π_{Triple}

This procedure runs in three phases: triple generation, triple sacrifice and opening checking. We assume that we start with an unset value for the flag Abort.

Triple Generation:

1. For $i = 1, \ldots, 2 \cdot n_T$ do
   (a) The parties call $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ twice to obtain $[\![a^{(i)}]\!]$ and $[\![b^{(i)}]\!]$
   (b) For each $j \in \mathcal{P}$, party j computes $v_j = \sum_{\rho(B_1,B_2)=j} a^{(i)}_{B_1} \cdot b^{(i)}_{B_2}$.
   (c) The parties call $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}$ so that player j obtains t_j such that $\sum_{l \in \mathcal{P}} t_l = 0$.
   (d) For each $j \in \mathcal{P}$, party j samples $c^{(i)}_B$ for $B \in \mathcal{B}_j$ such that $\sum_{B \in \mathcal{B}_j} c^{(i)}_B = v_j + t_j$.
   (e) For each $j \in \mathcal{P}$, party j sends, for all $B \in \mathcal{B}_j$, the value $c^{(i)}_B$ securely to party k for all $k \in B$.
2. The parties then run **Triple Sacrifice**.

Triple Sacrifice:

1. Call $\Pi_{\mathsf{Open}}.\mathsf{Init}$.
2. For $i = 1, \ldots, n_T$ do
   (a) The parties run $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ to obtain $[\![r^{(i)}]\!]$.
   (b) The parties run $\Pi_{\mathsf{Open}}.\mathbf{Reveal}([\![r^{(i)}]\!], 0)$ to obtain $r^{(i)}$.
   (c) The parties run $\Pi_{\mathsf{Open}}.\mathbf{Reveal}(\cdot, 0)$ on the values
   $$[\![b^{(i)}]\!] - [\![b^{(i+n_T)}]\!] \text{ and } r^{(i)} \cdot [\![a^{(i)}]\!] - [\![a^{(i+n_T)}]\!]$$
   and set the outputs to be $\sigma^{(i)}$ and $\rho^{(i)}$ respectively.
   (d) If the flag Abort has not been set to true, the parties locally compute value
   $$[\![z]\!] \leftarrow r^{(i)} \cdot [\![c^{(i)}]\!] - \sigma^{(i)} \cdot [\![a^{(i+n_T)}]\!] - \rho^{(i)} \cdot [\![b^{(i+n_T)}]\!] - [\![c^{(i+n_T)}]\!] - \sigma^{(i)} \cdot \rho^{(i)}.$$
   (e) The parties then run $\Pi_{\mathsf{Open}}.\mathbf{CompareView}$, and if Abort is not set to true then they perform $\Pi_{\mathsf{Open}}.\mathbf{Reveal}([\![z]\!], 0)$. If the returned value is not zero, the parties set the flag Abort to true.
3. The parties then run **Opening Check**.

Opening Check:

1. The parties run $\Pi_{\mathsf{Open}}.\mathbf{CompareView}$.
2. If the flag Abort is not set to true then the parties output their shares of $([\![a^{(i)}]\!], [\![b^{(i)}]\!], [\![c^{(i)}]\!])$, for $i = 1, \ldots, n_T$, and otherwise broadcast all shares of all their triples and output $\perp$.

Figure 7. Protocol Π_{Triple}

requires $|\mathsf{SC}^*|$ *secure channels, or* $|\mathsf{AC}|$ *authenticated channels if only public output is required.*

Actively Secure MPC Functionality $\mathcal{F}_{\mathsf{AMPC}}$

Init: The functionality receives a command Init from all parties to initialise.

Input: On input (Input, x_i) by party i, the functionality stores (id, x_i) in memory.

Add: On input $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$, the functionality retrieves (id_1, x) and (id_2, y) and stores $(\mathsf{id}_3, x + y)$.

Multiply: On input $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$, the functionality retrieves (id_1, x) and (id_2, y) and stores $(\mathsf{id}_3, x \cdot y)$.

Output: On input $(\mathbf{Output}, \mathsf{id}, i)$ from all parties, the functionality retrieves (id, x). It then does one of two things:

- If $i = 0$ it outputs x to the adversary. If the adversary returns Abort, then $\perp$ is returned to all players. If the adversary instead returns OK, x is passed to all players.
- If $i \neq 0$ then the functionality waits for input from the adversary. If the adversary returns Abort then $\perp$ is returned to all players. If the adversary instead returns OK, x is passed to player i.

Figure 8. Actively Secure MPC Functionality $\mathcal{F}_{\mathsf{AMPC}}$

4.2 Extension to Shamir Sharing

Our online protocol also works in the case of Shamir sharing, and here we can also reduce the required number of authenticated channels. Each party need only receive t shares (via authenticated channels) in order to reconstruct the sharing polynomial. From this polynomial they can then reconstruct the supposed shares of all other parties. By hashing all these shares, and then eventually comparing the hash values, the honest parties can ensure that the supposed opened values are all consistent and valid. Thus in the case of Shamir sharing our method of using hash values to impose honest behavious on malicious parties can result in a reduction of uni-directional authenticated channels from $n \cdot (n-1)$ down to $n \cdot t$.

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 grants EP/M012824 and EP/N021940/1.

The Protocol Π_{Online}

Recall the set $\mathcal{P}$ is the set of parties, and the set $A \subset \mathcal{P}$ the set of corrupt parties in $\mathcal{P}$. Recall that there is a partition $\mathcal{B} = \{\mathcal{B}_i\}_{i \in \mathcal{P}}$.

Init:
1. $H_i = \mathsf{Init}()$ for all players i.
2. The parties call $\mathcal{F}_{\mathsf{Triple}}$ to produce n_T triples, where n_T is the number of multiplication gates in the circuit. If $\mathcal{F}_{\mathsf{Triple}}$ aborts, then the parties abort the protocol.

[n_T can be a crude upper bound, if the number of triples runs out then $\mathcal{F}_{\mathsf{Triple}}$ can be called again.]

Input: For party i to provide input x,
1. The parties call $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ to obtain a sharing $[\![r]\!]$.
2. The parties call $\Pi_{\mathsf{Open}}.\mathbf{Reveal}([\![r]\!], i)$ to open r to player i.
3. The players execute $\Pi_{\mathsf{Open}}.\mathbf{Broadcast}(i, \epsilon)$ where $\epsilon = x - r$.
4. The parties locally compute[a] $[\![x]\!] = [\![r]\!] + \epsilon$.

Add:
1. For each $B \in \mathcal{B}$, each party $i \in B$ locally computes $x_B + y_B$ so that collectively the parties obtain $[\![x + y]\!]$.

Multiply: On input $([\![x]\!], [\![y]\!])$, the perform the following:
1. Take one unused multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from the pre-processing.
2. Compute $[\![\varepsilon]\!] \leftarrow [\![x]\!] - [\![a]\!]$ and $[\![\delta]\!] \leftarrow [\![y]\!] - [\![b]\!]$.
3. The parties run $\Pi_{\mathsf{Open}}.\mathbf{Reveal}(\cdot, 0)$ on $[\![\varepsilon]\!]$ and $[\![\delta]\!]$ to obtain ϵ and δ.
4. The parties set $[\![z]\!] \leftarrow [\![c]\!] + \varepsilon \cdot [\![b]\!] + \delta \cdot [\![a]\!] + \varepsilon \cdot \delta$.

Output$([\![x]\!], i)$:
1. The parties perform $\Pi_{\mathsf{Open}}.\mathbf{CompareView}$.
2. If Abort is true then the parties output $\perp$ and stop.
3. If $i \neq 0$ then the parties call $\Pi_{\mathsf{Open}}.\mathbf{Reveal}([\![x]\!], i)$. If Abort is not set to true, the party outputs x.
4. If $i = 0$ then the parties call $\Pi_{\mathsf{Open}}.\mathbf{Reveal}([\![x]\!], 0)$ to open x, and then $\Pi_{\mathsf{Open}}.\mathbf{CompareView}$. If Abort is true then the parties output $\perp$ and stop, otherwise they output x.

[a] This computation is done by the parties agreeing on some B and then adding ϵ to r_B, the rest of the shares being left as they are.

Figure 9. The Protocol Π_{Online}

References

AFL+16. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of*

the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pages 805–817. ACM, 2016.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

Bea96. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th Annual ACM Symposium on Theory of Computing*, pages 479–488, Philadephia, PA, USA, May 22–24, 1996. ACM Press.

BLW08. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008: 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.

BOGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

BW98. Donald Beaver and Avishai Wool. Quorum-based secure multi-party computation. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 375–390, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.

CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

CDN15. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

DGKN09. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

FLNW16. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. *IACR Cryptology ePrint Archive*, 2016:944, 2016.

GL02. Shafi Goldwasser and Yehuda Lindell. Secure computation without agreement. In Dahlia Malkhi, editor, *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, volume 2508 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2002.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

HIK07. Danny Harnik, Yuval Ishai, and Eyal Kushilevitz. How many oblivious transfers are needed for secure multiparty computation? In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 284–302, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

HM97. Martin Hirt and Ueli M. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In James E. Burns and Hagit Attiya, editors, *16th ACM Symposium Annual on Principles of Distributed Computing*, pages 25–34, Santa Barbara, CA, USA, August 21–24, 1997. Association for Computing Machinery.

HM00. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.

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

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

A Proof of Theorem 3

Proof. We prove security in the universal composability (UC) framework. In the UC model, we model all the possible operations of the world outside the single execution of the protocol by a probabilistic polynomial-time algorithm $\mathcal{Z}$ called the environment, which provides all inputs and sees all outputs of honest parties, and arbitrarily interacts with the adversary $\mathcal{A}$. We create a probabilistic polynomial-time algorithm $\mathcal{S}$, called the simulator whose job it is on one hand to interact with the adversary via the protocol, and on the other hand to interact with the ideal world via the functionality.

We prove the theorem if we show that the view of the environment when the adversary interacts with the real-world protocol is indistinguishable from the view when the adversary instead interacts with the ideal-world functionality via the simulator.

Simulator $\mathcal{S}_{\mathsf{PMPC}}$: Part 1/2

During simulation, whenever an input is provided, an output is given, or a multiplication is performed, the simulator is sent or already knows what shares the adversary holds for these secrets, and using this it maintains a list of shares it has seen. When the simulator and adversary compute a linear function on any of these secrets, the simulator must locally compute the same linear function on these shares in order to generate shares for the output which are consistent with the values the adversary has seen before.

For clarity, we use the variable k for corrupt parties, and the variable j for (simulated) honest parties.

Set-up: On receiving a command from the adversary to run $\mathcal{F}_{\mathsf{Rand}}.\mathsf{Set\text{-}up}$, the simulator simply initialises the functionality (internally).

Input: When party i is to provide input,

- The simulator receives the command $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}(\mathsf{cnt})$ from the adversary, which it executes honestly (internally).
- The simulator stores all the outputs $\{t_i\}_{i\in\mathcal{P}}$ from $\mathcal{F}_{\mathsf{Rand}}$ and sends the appropriate shares to the adversary, namely $\{t_i\}_{i\in\mathcal{A}}$.
- The simulator samples shares $\bigcup_{j\notin\mathcal{A}}\{x_B\}_{B\in\mathcal{B}_j} \leftarrow \mathbb{F}$ and sends

$$\left(\bigcup_{j\notin\mathcal{A}}\{x_B : B\in\mathcal{B}_j \text{ s.t. } k\in B\}\right)_{k\in\mathcal{A}}$$

  to the adversary. Note that there is at least one set $B\in\mathcal{B}$ such that $B\cap\mathcal{A}=\varnothing$; the simulator sets $x_B\leftarrow\perp$ for any such set(s). The simulator updates its list of stored values with these shares.
- (The simulator then waits for the adversary to send shares

$$\left(\bigcup_{k\in\mathcal{A}}\{\tilde{x}_B : B\in\mathcal{B}_k \text{ s.t. } j\in B\}\right)_{j\notin\mathcal{A}}$$

  to the simulator.)
- If i is corrupt, then since every $B\in\mathcal{B}$ contains an honest party, the adversary sends the entire set $\{x_B\}_{B\in\mathcal{B}_i}$ to the simulator. The simulator is therefore able to compute $x=-t_i+\sum_{B\in\mathcal{B}_i}x_B$, thus extracting the input x which it sends to the functionality $\mathcal{F}_{\mathsf{MPC}}$ as input for this party.

Figure 10. Simulator $\mathcal{S}_{\mathsf{PMPC}}$: Part 1/2

The simulator has access to the ideal-world functionality, and essentially extracts the inputs from the data it is sent by the adversary and then, acting as an "ideal-world" adversary, passes it on to the functionality, and finally returns to the adversary whatever the functionality returns to the simulator in the ideal world. Our proof is in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model, so the simulator is required to

respond to all calls the adversary makes to this functionality. The simulation is given in Figure 10 and Figure 11.

Now we argue that the simulation creates a view for the adversary which is indistinguishable from the view it has in the ideal world. The methods **Set-up** and **Add** require no simulation. The uniformly randomly sampled shares generated in **Input** and **Multiply** are computationally indistinguishable from the shares generated in a real-world execution of the protocol since in both cases the shares are masked by PRZSs from $\mathcal{F}_{\mathsf{Rand}}$.

Thus the only difficulty is in **Output**. Here the simulator must ensure that the outputs the adversary sees are consistent with what has already been revealed, which we now discuss in detail.

When an honest party provides input or the parties perform a multiplication of secrets, the simulator samples shares for honest parties uniformly at random, except at least one share held by only honest parties which it does not (and need not) fix. Such a share exists since otherwise the adversary holds every share. When the simulator is required to produce output, two possible cases occur. Either the output is a linear function of previously seen sharings; in which case the simulator can compute all of the shares (by computing the linear function) and hence return the valid sharing. Otherwise the output is not a linear function, in which case the simulator obtains the desired output from the functionality and samples the shares of the honest players so that they sum (with the adversaries shares) to the correct value.

In either case this means that the neither the sets of shares revealed in our simulation nor the values to which they reconstruct provide the environment with any information with which to distinguish between a real execution of the protocol and the ideal world.

□

Simulator $\mathcal{S}_{\mathsf{PMPC}}$: Part 2/2

Add: The simulator sends the command $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to the functionality $\mathcal{F}_{\mathsf{MPC}}$.

Multiply: To multiply a secret,

- The simulator receives the command $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}(\mathsf{cnt})$ from the adversary, which it executes honestly (internally).
- The simulator stores all the outputs t_i from $\mathcal{F}_{\mathsf{Rand}}$ and sends $\{t_i\}_{i\in\mathcal{A}}$ to the adversary.
- The simulator samples shares $\bigcup_{j\notin\mathcal{A}}\{z_B\}_{B\in\mathcal{B}_j} \leftarrow \mathbb{F}$ and sends

$$\left(\bigcup_{j\notin\mathcal{A}}\{z_B : B\in\mathcal{B}_j \text{ s.t. } k\in B\}\right)_{k\in\mathcal{A}}$$

  to the adversary. The simulator updates its list of stored values with these shares.
- (The simulator then waits for the adversary to send shares

$$\left(\bigcup_{k\in\mathcal{A}}\{\tilde{z}_B : B\in\mathcal{B}_k \text{ s.t. } j\in B\}\right)_{j\notin\mathcal{A}}$$

  to the simulator.)
- Finally, the simulator sends the command $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to the functionality $\mathcal{F}_{\mathsf{MPC}}$.

Output: On receiving the command to output shares, with input i,

- The simulator sends the same command to the functionality $\mathcal{F}_{\mathsf{MPC}}$ and receives an output value x.
- Using the list of shares it stored throughout, the simulator generates a set of shares which are consistent with the shares the adversary has seen before and which also sum to the secret x. This is either done by using a linear function of existing shares output by the simulator, or by sampling new shares (which it can always do because there is some $B\in\mathcal{B}$ such that $B\cap\mathcal{A}=\varnothing$), depending on whether x is a linear combination of previously output values.
- If $i\neq 0$ and $i\in\mathcal{A}$, the simulator sends the set

$$\bigcup_{j\notin\mathcal{A}}\{x_B : B\in\mathcal{B}_j \text{ s.t. } i\notin B\}$$

  to the adversary, and if $i=0$ it sends the tuple

$$\left(\bigcup_{j\notin\mathcal{A}}\{x_B : B\in\mathcal{B}_j \text{ s.t. } k\notin B\}\right)_{k\in\mathcal{A}}.$$

Figure 11. Simulator $\mathcal{S}_{\mathsf{PMPC}}$: Part 2/2

B Proof of Theorems 4 and 5

Proof (Of Theorem 4). We are in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model, so the simulator is required to respond to all calls made by the adversary to $\mathcal{F}_{\mathsf{Rand}}$. We describe the simulator after briefly discussing our notation. In the description of the simulator, for a secret sharing of a value c, we let the shares of c held by honest party j, for $B \in \mathcal{B}$ with $j \in B$, be denoted by $c_{B,j}$. This is to model the fact that the adversary can send different values for the same share to different honest parties, even though they are supposed to be the same. If $c_{B,j} = c_{B,j'}$ for all j, j' we write the share simply as c_B. Errors in honest parties' shares can either come from this inconsistency, or from the fact that the adversary has given *all* honest parties the wrong value of c_B for a set B for which it is responsible. The simulation can be found in Figure 12, Figure 13, Figure 14 and Figure 15.

The functionality is designed so that the adversary can choose its shares, and subsequently can cause an abort, or can allow honest parties to receive outputs. In the latter case, the outputs received by honest parties are necessarily a valid triple with the shares the adversary chose and sent to the functionality. Similarly, in the protocol, the parties either generate a correct triple, or they abort – they cannot generate an incorrect triple without aborting. A technicality requires that the functionality send shares it generated for honest parties to the adversary in the case of an abort. We will now show that the simulator provides an interface between the real-world adversary and the ideal-world functionality so that no environment can determine which of these worlds it is working in.

During **Triple Generation** (Figure 12), when the adversary calls $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$, the simulator just passes on what it receives from the functionality for the secrets a and b. The simulator can do this because $\mathcal{F}_{\mathsf{Rand}}$ is actively secure, so the uniformly sampled shares from the functionality are indistinguishable from an output of $\mathcal{F}_{\mathsf{Rand}}$.

Next, the simulator runs $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}$, just as the real parties would in the real world, so the outputs are identically distributed. The shares of the product $c^{(i)}$ which the simulator samples uniformly at random are indistinguishable from shares of the masked Schur product because the PRZSs (computationally) hide the sums. More formally, for any (finite) indexing set I, for any $i^* \in I$,

$$\begin{aligned}
&\left\{(a_i)_{i\in I\setminus\{i^*\}} : a_i \leftarrow \mathbb{F}_q \text{ uniformly}\right\}_q \\
&\qquad\qquad \equiv \left\{(z_i)_{i\in I\setminus\{i^*\}} : z_i \leftarrow \mathbb{F}_q \text{ uniformly s.t. } \sum_{i\in I} z_i = 0\right\}_q \\
&\qquad\qquad \approx_C \left\{(z_i)_{i\in I\setminus\{i^*\}} : \{z_i\}_{i\in I} \leftarrow \mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}\right\}_q .
\end{aligned}$$

If there are any discrepancies between any individual shares of $c^{(i)}$ which are supposed to be sent to different honest parties, the protocol will abort, because $c^{(i)}$ forms part of a share which is opened publicly later: while the value $c^{(i)}$ is never publicly opened, the public value $[\![z^{(i)}]\!]$ in **Triple Sacrifice** is a linear

Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 1/4 (Triple Generation)

For clarity, we use the variable k for corrupt parties, and the variable j for (simulated) honest parties.

Triple Generation:

- The simulator does the following $2 \cdot n_T$ times, indexed by i:
  - The simulator invokes the functionality $\mathcal{F}_{\mathsf{Triple}}$ and receives

$$(\{a_B^{(i)}, b_B^{(i)} : B \in \mathcal{B} \text{ s.t. } k \in B\})_{k \in \mathcal{A}}.$$

    This is a tuple of sets of shares, where the set indexed by $k \in \mathcal{A}$ is the set of shares received by corrupt party k.
  - When the adversary makes a call to $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ for $a^{(i)}$ and $b^{(i)}$, if $i \leq n_T$ the simulator just returns the shares it received from $\mathcal{F}_{\mathsf{Triple}}$, and if $i > n_T$ the simulator executes $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ honestly and returns the appropriate shares to the adversary.
  - When the adversary makes a call to $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRZS}$ for a secret-shared zero, the simulator executes this internally to obtain $\{t_l^{(i)}\}_{l \in \mathcal{P}}$ with $\sum_{l \in \mathcal{P}} t_l^{(i)} = 0$, which the simulator stores. The simulator sends $(t_k^{(i)})_{k \in \mathcal{A}}$ to the adversary.
  - The simulator samples a set $\bigcup_{j \notin \mathcal{A}} \{c_B^{(i)} : B \in \mathcal{B}_j \text{ s.t. } B \cap \mathcal{A} \neq \varnothing\} \leftarrow \mathbb{F}$, sends the tuple

$$\left(\bigcup_{j \notin \mathcal{A}} \{c_B^{(i)} : B \in \mathcal{B}_j \text{ s.t. } k \in B\} \right)_{k \in \mathcal{A}}$$

    to the adversary, and receives back a tuple

$$S \leftarrow \left(\bigcup_{k \in \mathcal{A}} \{c_{B,j}^{(i)} : B \in \mathcal{B}_k \text{ s.t. } j \in B\} \right)_{j \notin \mathcal{A}},$$

  - If $c_{B,j_1}^{(i)} \neq c_{B,j_2}^{(i)}$ for any $j_1 \neq j_2$ and the simulator sends S to the functionality, it will abort; instead, in this case, for each $B \in \bigcup_{k \in \mathcal{A}} \mathcal{B}_k$, the simulator chooses $c_{B,j}^{(i)}$ for some $j \in \mathcal{A}$, fixes $c_B^{(i)} \leftarrow c_{B,j}^{(i)}$, and then sends the tuple $(\bigcup_{j \notin \mathcal{A}} \{c_B^{(i)} : B \in \mathcal{B}_k \text{ s.t. } j \in B\})_{j \notin \mathcal{A}}$ to the functionality. The simulator then immediately sends Abort to the functionality and receives back $\{a_B^{(i)}, b_B^{(i)}, c_B^{(i)} : B \in \mathcal{B} \text{ s.t. } B \cap \mathcal{A} = \varnothing\}$.
  - If $c_{B,j_1}^{(i)} = c_{B,j_2}^{(i)}$ for all j_1, j_2, the simulator sends the set S to the functionality. The functionality now waits for the Abort or $\mathsf{Deliver}$ signal. Meanwhile, the simulation continues: for each $B \in \mathcal{B}$, the simulator sets $c_B^{(i)} \leftarrow c_{B,j}^{(i)}$ for any $j \notin \mathcal{A}$ (since they are all the same).
- The simulator then executes $\mathcal{S}_{\mathsf{Triple}}$.**Triple Sacrifice** with the adversary.

Figure 12. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 1/4 (Triple Generation)

combination of this (and other) secrets, so if the adversary sends different values

for the same share to different honest parties, the shares of $z^{(i)}$ will necessarily differ between honest parties and thus be detected in **CompareView** later on.

In **Triple Sacrifice** (Figure 13 and Figure 14), the simulator initialises the hashes as in $\Pi_{\mathsf{Open}}.\mathsf{Init}$ and then runs $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ just as the parties do in the real protocol execution to receive some $r^{(i)}$ and all of its shares. When the adversary and simulator open $r^{(i)}$, the shares are added to the hash in $\Pi_{\mathsf{Open}}.\mathbf{Reveal}()$, so any discrepancies between honest shares are detected in **CompareView** later on.

To the adversary (and the environment) the shares of the public values $\sigma^{(i)}$ and $\rho^{(i)}$, and indeed the values themselves, are indistinguishable from uniformly random in the real world since the secrets $a^{(i+n_T)}$ and $b^{(i+n_T)}$ are never opened. Thus it suffices for the simulator to sample shares uniformly at random for these two public values for all shares held only by honest parties (but since the simulator actually computes them anyway they are included in the simulation). The shares held by the adversary for these values are the result of a linear function on shares already sent from or received by the adversary (i.e. which are in the transcript between the adversary and the simulator). Because, for each share, there is at least one (simulated) honest party which will have computed the linear function faithfully, if the adversary sends a share of $\sigma^{(i)}$ or $\rho^{(i)}$ which is different from what it should have calculated according to previous shares it sent or received, the protocol aborts in **CompareView**.

In the protocol, before the parties open the (alleged) zero, they run **CompareView**, and abort if any two hashes differ. If the parties abort, the simulator also aborts, and otherwise continues. This ensures that all shares of $r^{(i)}$, $\sigma^{(i)}$ and $\rho^{(i)}$ are consistent, since otherwise the adversary would have broken the collisin resistance of the hash function. Moreover, if **CompareView** did not abort, this means that the adversary cannot have introduced an error on any of these three values, by the correctness of $\Pi_{\mathsf{Rand}}.\mathsf{PRSS}$.

Now we will discuss what happens when the simulator and adversary compute $[\![z^{(i)}]\!]$. If the values $c^{(i)} \leftarrow a^{(i)} \cdot b^{(i)}$ or $c^{(i+n_T)} \leftarrow a^{(i+n_T)} \cdot b^{(i+n_T)}$ have had errors $\Delta_{c^{(i)}}$ and $\Delta_{c^{(i+n_T)}}$ introduced on them, then value $z^{(i)}$ becomes

$$\begin{aligned}[\![z^{(i)}]\!] \leftarrow r^{(i)} \cdot [\![c^{(i)} + \Delta_{c^{(i)}}]\!] - \sigma^{(i)} \cdot [\![a^{(i+n_T)}]\!] - \rho^{(i)} \cdot [\![b^{(i+n_T)}]\!] \\ - [\![c^{(i+n_T)} + \Delta_{c^{(i+n_T)}}]\!] - \sigma^{(i)} \cdot \rho^{(i)},\end{aligned}$$

and will be zero if and only if $r \cdot \Delta_{c^{(i)}} - \Delta_{c^{(i+n_T)}} = 0$, which occurs with probability $1/q$ (where q is the field size) since r is chosen (computationally indistinguishably from) uniformly at random (and is chosen after the adversary has already chosen the errors on $c^{(i)}$ and $c^{(i+n_T)}$.

The simulator performs local operations on the shares it sent and received to produce shares $z_B^{(i)}$ for all $B \in \mathcal{B}$ where $B \cap \mathcal{A} \neq \varnothing$. Note that different (simulated) honest parties will (potentially) compute different values for the same shares for certain shares of $z^{(i)}$, depending on what the adversary sent to the simulator for the shares of $c^{(i)}$: for example, if the adversary sent $c_{B,1}^{(i)}$ to

party 1 and $c^{(i)}_{B,2}$ to party 2, then the defining equation for $z^{(i)}$ shows that their shares for $z^{(i)}_B$ will differ by $r \cdot (c^{(i)}_{B,1} - c^{(i)}_{B,2})$.

For the remaining shares, that is, $z^{(i)}_B$ for $B \in \mathcal{B}$ with $B \cap \mathcal{A} = \varnothing$, the simulator must sample shares so that they appear to be consistent with the values the adversary has seen before, i.e. as something indistinguishable from what it would see in an execution of the protocol. To do this, the simulator must first compute some errors; in particular, the simulator uses the fact that it knows the shares $a^{(i)}_B$, $b^{(i)}_B$, and PRZSs t_i held by corrupt parties to compute any errors the adversary introduced when multiplying secrets during **Triple Generation**.

Observe that the errors computed in the simulation depend on the choice of shares for $c^{(i)}_B$: recall that the adversary sent the simulator a set of shares $(\bigcup_{k\in\mathcal{A}}\{c^{(i)}_{B,j} : B \in \mathcal{B}_k \text{ s.t. } j \in B\})_{j\notin\mathcal{A}}$ from which it arbitrarily assigned $c^{(i)}_B$ to be $c^{(i)}_{B,j}$ for any $j \in B \setminus \mathcal{A}$. Importantly, the value $r\cdot\Delta_{c^{(i)}} - \Delta_{c^{(i+n_T)}} - \sum_{B:B\cap\mathcal{A}\neq\varnothing} z^{(i)}_B$ is independent of the choice made for the $c^{(i)}_B$'s since the errors are dependent on the choice. (More explicitly, observe that

$$
\begin{aligned}
r \cdot \Delta_{c^{(i)}} - \Delta_{c^{(i+n_T)}} - \sum_{B:B\cap\mathcal{A}\neq\varnothing} z^{(i)}_B & \\
&= r^{(i)} \cdot \sum_{k\in\mathcal{A}} \left(\sum_{B\in\mathcal{B}_k} c^{(i)}_B - \left(t^{(i)}_k + \sum_{\rho(B_1,B_2)=k} a^{(i)}_{B_1} \cdot b^{(i)}_{B_2} \right) \right) \\
&\quad - \sum_{k\in\mathcal{A}} \left(\sum_{B\in\mathcal{B}_k} c^{(i+n_T)}_B - \left(t^{(i+n_T)}_k + \sum_{\rho(B_1,B_2)=k} a^{(i+n_T)}_{B_1} \cdot b^{(i+n_T)}_{B_2} \right) \right) \\
&\quad - \sum_{B:B\cap\mathcal{A}\neq\varnothing} r^{(i)} \cdot c^{(i)}_B - \sigma^{(i)} \cdot a^{(i+n_T)}_B - \rho^{(i)} \cdot b^{(i+n_T)}_B - c^{(i+n_T)}_B \\
&\quad - \sigma^{(i)} \cdot \rho^{(i)} \\
&= -\Big(\sum_{B\in\mathcal{B}_j : B\cap\mathcal{A}\neq\varnothing} r^{(i)} \cdot c^{(i)}_B - c^{(i+n_T)}_B \Big) + k,
\end{aligned}
$$

where k is a constant independent of $c^{(i)}_B$ and $c^{(i+n_T)}_B$, is independent of any shares sent by the adversary.) This means that whatever is sampled for the remaining shares of $z^{(i)}$ is consistent with the choice made before.

In **Opening Checking** (Figure 15), the simulator follows the protocol, and aborts exactly when the real-world execution of the protocol would abort, since the simulator only sets the flag Abort to true if the protocol is able to detect the adversary has cheated. In particular, this check ensures that whenever the adversary sends different values for the same shares of a given secret to different honest parties (e.g. when opening $\sigma^{(i)}$, $\rho^{(i)}$, $r^{(i)}$ or $z^{(i)}$), the hashes will differ, causing an abort.

We have shown that distributions of shares of individual secrets are indistinguishable in both worlds, but we must also ensure that the combined distribution

of all shares received by the adversary and the outputs of the honest parties is indistinguishable as a whole. This follows trivially from the fact that, for each of the n_T triples output, the parties in the ideal world only receive one triple and the other is discarded (sacrificed), and not output by the honest parties, so these triples mask the triples which are output, and from the fact that all public values are indistinguishable from uniformly random and are independent, and that there are no secrets which the environment can reconstruct at the end of the computation except the triples, which are identical in both worlds by construction. □

Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 2/4 (Triple Sacrifice Part I)

Triple Sacrifice:

– The simulator first initialises the hash function for all (simulated) honest parties.

– The simulator does the following n_T times, indexed by i:

- The simulator responds to the adversary's call to $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ by executing it locally (obtaining some $r^{(i)}$) and sending $(\{r_B^{(i)} : B \in \mathcal{B} \text{ s.t. } k \in B\})_{k \in \mathcal{A}}$ to the adversary.
- The simulator then sends

$$\left(\bigcup_{j \notin \mathcal{A}} \{r_B^{(i)} : B \in \mathcal{B}_j \text{ s.t. } k \notin B\} \right)_{k \in \mathcal{A}}$$

to the adversary to open the secret $r^{(i)}$, and receives back a tuple

$$\left(\bigcup_{k \in \mathcal{A}} \{r_{B,j}^{(i)} : B \in \mathcal{B}_k \text{ s.t. } j \notin B\} \right)_{j \notin \mathcal{A}}.$$

If $r_{B,j}^{(i)} \neq r_B^{(i)}$ for any $j \notin \mathcal{A}$, the simulator signals Abort to the functionality (if it has not already done so) and receives $\{a_B^{(i)}, b_B^{(i)}, c_B^{(i)} : B \in \mathcal{B} \text{ s.t. } B \cap \mathcal{A} = \varnothing\}$.

- Then the simulator does the following:
  - For each $B \in \bigcup_{j \notin \mathcal{A}} \mathcal{B}_j$, the simulator sets $r_{B,j}^{(i)} \leftarrow r_B^{(i)}$ for all $j \in B \setminus \mathcal{A}$.
  - For each $j \notin \mathcal{A}$, the simulator executes $H_j.\mathsf{Update}(r_{B,j}^{(i)})$ for all $B \in \mathcal{B}$.
- The simulator then samples $\bigcup_{j \notin \mathcal{A}} \{\sigma_B^{(i)}, \rho_B^{(i)} : B \in \mathcal{B}_j\} \leftarrow \mathbb{F}$ for the public values $\sigma^{(i)}$ and $\rho^{(i)}$, sends to the adversary the set

$$\left(\bigcup_{j \notin \mathcal{A}} \{\sigma_B^{(i)},\ \rho_B^{(i)} : B \in \mathcal{B}_j \text{ s.t. } k \notin B\} \right)_{k \in \mathcal{A}},$$

and receives back a tuple

$$\left(\bigcup_{k \in \mathcal{A}} \{\sigma_{B,j}^{(i)},\ \rho_{B,j}^{(i)} : B \in \mathcal{B}_k \text{ s.t. } j \notin B\} \right)_{j \notin \mathcal{A}}.$$

If $\sigma_{B,j_1} \neq \sigma_{B,j_2}$ or $\rho_{B,j_1} \neq \rho_{B,j_2}$ for any $j_1 \neq j_2$, for any $B \in \mathcal{B}$, the simulator signals Abort to the functionality (if it has not already done so) and receives $\{a_B^{(i)}, b_B^{(i)}, c_B^{(i)} : B \in \mathcal{B} \text{ s.t. } B \cap \mathcal{A} = \varnothing\}$.

- Then the simulator does the following:
  - For each $B \in \bigcup_{j \notin \mathcal{A}} \mathcal{B}_j$, the simulator sets $\sigma_{B,j} \leftarrow \sigma_B$ and $\rho_{B,j} \leftarrow \rho_B$ for each $j \in B \setminus \mathcal{A}$.
  - For each $j \notin \mathcal{A}$, the simulator executes $H_j.\mathsf{Update}(\sigma_{B,j}^{(i)})$ and $H_j.\mathsf{Update}(\rho_{B,j}^{(i)})$ for all $B \in \mathcal{B}$.

[Continued]

Figure 13. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 2/4 (Triple Sacrifice Part I)

Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 3/4 (Triple Sacrifice Part II)

- For each $j \notin \mathcal{A}$, the simulator computes $h_j \leftarrow H_j.\mathsf{Finalise}()$ and sends these to the adversary. It also receives a set of hashes from the adversary. If any two hashes differ, the simulator sets the flag Abort to true. Otherwise, the parties all have consistent shares for $r^{(i)}$, $\sigma^{(i)}$ and $\rho^{(i)}$, so we label them r_B for $B \in \mathcal{B}$ etc.
- The simulator then forms honest players' shares $z^{(i)}_{B,j}$ for B with $B \cap \mathcal{A} \neq \varnothing$ of

$$[\![z^{(i)}]\!] \leftarrow r^{(i)} \cdot [\![c^{(i)}]\!] - \sigma^{(i)} \cdot [\![a^{(i+n_T)}]\!] - \rho^{(i)} \cdot [\![b^{(i+n_T)}]\!] - [\![c^{(i+n_T)}]\!] - \sigma^{(i)} \cdot \rho^{(i)}$$

  using the values it already has.
- Before the simulator computes shares for honest parties $z^{(i)}_B$ for B with $B \cap \mathcal{A} = \varnothing$, the simulator computes various errors which could have been introduced by the adversary,

$$\Delta_{c^{(i)}} \leftarrow \sum_{k \in \mathcal{A}} \left(\sum_{B \in \mathcal{B}_k} c^{(i)}_B - \left(t^{(i)}_k + \sum_{\rho(B_1,B_2)=k} a^{(i)}_{B_1} \cdot b^{(i)}_{B_2} \right) \right),$$

$$\Delta_{c^{(i+n_T)}} \leftarrow \sum_{k \in \mathcal{A}} \left(\sum_{B \in \mathcal{B}_k} c^{(i+n_T)}_B - \left(t^{(i+n_T)}_k + \sum_{\rho(B_1,B_2)=k} a^{(i+n_T)}_{B_1} \cdot b^{(i+n_T)}_{B_2} \right) \right),$$

- The simulator then computes $a^{(i+n_T)} \leftarrow \sum_{B \in \mathcal{B}} a^{(i+n_T)}_B$ and $b^{(i+n_T)} \leftarrow \sum_{B \in \mathcal{B}} b^{(i+n_T)}_B$. (Recall these were just the outputs of $\Pi_{\mathsf{Rand}}.\mathsf{PRSS}$.)
- Using the chosen shares $\{c_B : B \in \mathcal{B} \text{ s.t. } B \cap \mathcal{A} \neq \varnothing\}$, the simulator computes corresponding shares $\{z_B : B \in \mathcal{B} \text{ s.t. } B \cap \mathcal{A} \neq \varnothing\}$ and then samples $\{z_B : B \in \mathcal{B} \text{ s.t. } B \cap \mathcal{A} = \varnothing\} \leftarrow \mathbb{F}$ such that

$$\sum_{B \cap \mathcal{A} = \varnothing} z^{(i)}_B + \sum_{B \cap \mathcal{A} \neq \varnothing} z^{(i)}_B = r^{(i)} \cdot \Delta_{c^{(i)}} - \Delta_{c^{(i+n_T)}}$$

- The simulator sets $z_{B,j} \leftarrow z_B$ for every $B \in \mathcal{B}_j$ where $j \notin \mathcal{A}$ and sends the tuple $(\bigcup_{j \notin \mathcal{A}} \{z^{(i)}_{B,j} : B \in \mathcal{B}_j \text{ s.t. } k \notin B)_{k \in \mathcal{A}}$ to the adversary. The adversary returns a tuple of shares $(\bigcup_{k \in \mathcal{A}} \{z^{(i)}_{B,j} : B \in \mathcal{B}_k \text{ s.t. } j \notin B)_{j \notin \mathcal{A}}$ and for each $j \notin \mathcal{A}$, the simulator executes $H_j.\mathsf{Update}(z^{(i)}_{B,j})$ for all $B \in \mathcal{B}$. If $z^{(i)} \neq 0$, the simulator sets the Abort flag to true.

– The simulator then runs $\mathcal{S}_{\mathsf{Triple}}$.**Opening Check** with the adversary.

Figure 14. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 3/4 (Triple Sacrifice Part II)

Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 4/4 (Opening Check)

Opening Check: The simulator and adversary run the final check before output:

- The simulator sets $h_j \leftarrow H_j.\mathsf{Finalise}$ for each honest party $j \notin \mathcal{A}$ and sends them to the adversary; the adversary returns some set of hashes.
- The simulator compares all hashes and sets Abort to true if two hashes differ.
- If the simulator or adversary has set Abort to true, the simulator sends the message Abort to the functionality (unless it has already done so). The functionality returns the honest parties shares for the values $a^{(i)}$, $b^{(i)}$, and $c^{(i)}$. The simulator passes these on, by sending $(\{(a^{(i)}_{B,j}, b^{(i)}_{B,j}, c^{(i)}_{B,j} : B \in \mathcal{B} \text{ s.t. } j \in \mathcal{B}\})_{j \notin \mathcal{A}}$ to the adversary,
- If the flag Abort has not been set to true, the simulator signals $\mathsf{Deliver}$ to the functionality.

Figure 15. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 4/4 (Opening Check)

Proof (Of Theorem 5).

Finally, we can prove that Π_{Online} securely realises the actively secure functionality $\mathcal{F}_{\mathsf{AMPC}}$. To do this, we first provide a simulator, given in Figure 16 and Figure 17.

Online Simulator $\mathcal{S}_{\mathsf{Online}}$: Part 1/2

During simulation, whenever an input is provided, an output is given, or a multiplication is performed, the simulator is sent or already knows what shares the adversary holds for these secrets, and using this it maintains a list of shares it has seen. It also stores a list of all shares it has sampled and sent to the adversary. When the simulator and adversary compute a linear function on any of these secrets, the simulator must locally compute the same linear function on these shares in order to generate shares for the output which are consistent with the values the adversary has seen before.

Init: The simulator initialises the hash for simulated honest parties, $H_j \leftarrow \mathsf{Init}()$ and then responds to the adversary's call to $\mathcal{F}_{\mathsf{Triple}}$ with the appropriate shares of the triples obtained by running $\mathcal{F}_{\mathsf{Triple}}$ internally. Note that if the adversary signals Abort to the simulator to abort $\mathcal{F}_{\mathsf{Triple}}$, the simulator can abort the functionality and return shares to the adversary, and the protocol aborts. If $\mathcal{F}_{\mathsf{Triple}}$ aborted, the simulator does not send the signal Init to the functionality $\mathcal{F}_{\mathsf{AMPC}}$.

Input: If input is to be given by party i,

- The simulator first responds to the call by the adversary to $\mathcal{F}_{\mathsf{Rand}}.\mathsf{PRSS}$ by returning $(\{r_B : B \in \mathcal{B} \text{ s.t. } k \in B)_{k \in \mathcal{A}}$ from an internally-executed instance of $\mathcal{F}_{\mathsf{Rand}}$.
- If i is honest,
  - The simulator waits for the adversary to send shares $(\{r_{B,j} : B \in \mathcal{B} \text{ s.t. } j \in B \text{ and } i \notin B\})_{j \in \mathcal{A}}$. If $r_{B,j} \neq r_B$ for any $j \in \mathcal{A}$, the simulator sets the flag Abort to true. The simulator and stores these shares in the list of shares (see preamble).
  - The simulator samples $\epsilon \leftarrow \mathbb{F}$ and sends $(\epsilon)_{j \in \mathcal{A}}$ to the adversary. The simulator also updates the hash $H_j.\mathsf{Update}(\epsilon)$ for every $j \notin \mathcal{A}$.
- If i is corrupt,
  - The simulator sends shares $(\{r_B : B \in \mathcal{B} \text{ s.t. } j \in B \text{ and } i \notin B\})_{j \notin \mathcal{A}}$ to the adversary, which is the opening of $[\![r]\!]$ to corrupt party $i \in \mathcal{A}$.
  - The adversary returns a tuple $(\epsilon_j)_{j \notin \mathcal{A}}$ and the simulator updates the hash $H_j.\mathsf{Update}(\epsilon_j)$ for every $j \notin \mathcal{A}$.
  - If the ϵ_j are equal for all $j \in \mathcal{A}$, the simulator computes $x \leftarrow \epsilon_j + r$ and forwards x to the functionality. If not, the simulator sets the Abort flag to true and chooses any ϵ_j to derive some x to send to the functionality (since the functionality will abort anyway).

Add: The simulator sends the command $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to the functionality.

Figure 16. Online Simulator $\mathcal{S}_{\mathsf{Online}}$: Part 1/2

Online Simulator $\mathcal{S}_{\mathsf{Online}}$: Part 2/2

Multiply: To multiply secrets x and y,

- The simulator retrieves the shares for $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, $[\![x]\!]$ and $[\![y]\!]$ from the list of shares it has stored, samples a set $\bigcup_{j\notin\mathcal{A}}\{\varepsilon_B, \delta_B : B \in \mathcal{B}_j \text{ s.t. } B \cap \mathcal{A} \neq \varnothing\} \leftarrow \mathbb{F}$ and sends $\left(\bigcup_{j\notin\mathcal{A}}\{\varepsilon_B, \delta_B : B \in \mathcal{B}_j \text{ s.t. } k \notin B\}\right)_{k\in\mathcal{A}}$ to the adversary and adds these shares to the list of stored shares.
- The adversary returns a set $\left(\bigcup_{k\in\mathcal{A}}\{\varepsilon_{B,j}, \delta_{B,j} : B \in \mathcal{B}_k \text{ s.t. } j \notin B\}\right)_{j\notin\mathcal{A}}$ which the simulator stores in its list of shares.
- For each $B \in \bigcup_{j\notin\mathcal{A}} \mathcal{B}_j$, the simulator sets $\epsilon_{B,j} \leftarrow \epsilon_B$ and $\rho_{B,j} \leftarrow \rho_B$ and then for each $j \notin \mathcal{A}$ executes $H_j.\mathsf{Update}(\epsilon_{B,j})$ and $H_j.\mathsf{Update}(\rho_{B,j})$ for all $B \in \mathcal{B}$.
- Finally, the simulator sends the command $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to the functionality.)

Output: When the simulator receives the command $\mathbf{Output}([\![x]\!], i)$ from the adversary,

- The simulator sends the message $\mathbf{Output}(\mathsf{id}_x, i)$ to $\mathcal{F}_{\mathsf{AMPC}}$. If $i = 0$ then the functionality just returns x to the simulator straight away.
- The simulator computes $h_j \leftarrow H_j.\mathsf{Finalise}$ for all $j \notin \mathcal{A}$ and sends them to the adversary. If any two hashes are different or the adversary outputs Abort, the simulator sets its internal Abort flag to true.
- The simulator reinitialises the hash for the (simulated) honest players.
- If the Abort flag has been set to true, the simulator tells the functionality $\mathcal{F}_{\mathsf{AMPC}}$ to abort and outputs $\perp$ to the adversary. Otherwise the simulator continues as follows.
- If $i \neq 0$ is honest, the simulator waits for shares $(\{x_{B,j} : B \in \mathcal{B},\ i \notin B,\ j \in B\})_{j\in\mathcal{A}}$ from the adversary. If $x_{B,j_1} \neq x_{B,j_2}$ for any $j_1 \neq j_2$, the simulator sets Abort to true, signals Abort to the functionality, and sends $\perp$ to the adversary; otherwise, it sends the command OK to the functionality, which passes x to honest player i.
- If $i \neq 0$ is corrupt, the simulator signals OK to the functionality and receives x back. Using the list of shares it stored throughout, the simulator generates a set of shares for x which are consistent with the shares the adversary has seen before and which also sum to the secret x. Finally, the simulator sends the shares $\left(\bigcup_{j\notin\mathcal{A}}\{x_B : B \in \mathcal{B} \text{ s.t. } j \in B \text{ and } k \notin B\}\right)_{k\in\mathcal{A}}$ to the adversary.
- If $i = 0$, the simulator does the same generation of consistent shares so that they sum to the output x as in the case where $i \neq 0$ is corrupt and sends the set of shares $\left(\bigcup_{j\notin\mathcal{A}}\{x_B : B \in \mathcal{B}_j \text{ s.t. } k \notin B\}\right)_{k\in\mathcal{A}}$ to the adversary. The adversary returns a set $\left(\bigcup_{k\in\mathcal{A}}\{x_B : B \in \mathcal{B}_k \text{ s.t. } j \notin B\}\right)_{j\notin\mathcal{A}}$ to the simulator and the simulator and adversary compute the hashes as before and set the flag Abort if either any two hashes differ. If the flag Abort has not been set to true by either the adversary or the simulator, the simulator signals OK to the functionality, and otherwise signals Abort.

Figure 17. Online Simulator $\mathcal{S}_{\mathsf{Online}}$: Part 2/2

We first note that in the simulation, when a secret value is opened using in the **Input**, **Multiply** or **Output** commands by calling Π_{Open}.**Reveal**, we are gauranteed that the values the adversary sends to each honest player are identical. Since otherwise the adversary would be able to break the collision resistance of the hash function.

During **Init**, the simulator just sends output from $\mathcal{F}_{\mathsf{Triple}}$, which was run internally by the simulator, which is what the parties do in the protocol.

For **Input**, if the party providing input is honest, after receiving the opening of $[\![r]\!]$ from the adversary the simulator just broadcasts a uniformly randomly sampled value ϵ to the adversary. This ϵ is (computationally) indistinguishable from what an honest party sends in the real world since an honest party's input is masked with a mask from the PRSS in the protocol execution. The simulator sets the Abort flag to true if the adversary sends different shares of r from what were prescribed during PRSS; this cheating is caught in the protocol because every share party i does not have is sent to by at least one honest party, and Π_{Open}.**Reveal** causes an abort in if any shares differ. If the party providing input is corrupt, and it sends a different ϵ_j to each $j \notin \mathcal{A}$, the simulator sets the Abort flag to true. The ability of the simulator to detect the error is mirrored in the protocol by the fact that the broadcasted value is added to the hash input (in Π_{Open}.**Broadcast**), so if any honest parties receive different values the protocol aborts before output is given (or the adversary has broken collision resistance of the hash function).

No simulation is required for **Add** since there is no communication.

During the method **Multiply**, the simulator just samples shares to send to the adversary and stores shares sent to it. It can do this because the environment will not be able to reconstruct secrets before a value has been output in **Output**, so all shares are indistinguishable from uniformly random, as they would be in a real execution of the protocol.

For **Output**, when values are opened to the adversary, because linear functions on secrets are executed in the protocol by performing the same linear functions on the individual shares, it is necessary for the simulator to ensure that any shares the simulator sends to the adversary are consistent with any shares of secrets which have been opened already or for which the adversary otherwise knows some of the shares. Using the shares it stores at various points in the simulation, the simulator can do precisely this, and moreover can cause the secret being revealed to be opened to whatever it chooses. This is because there is at least one share which is held only by honest parties, and is therefore not already included in the transcript of communication between the adversary and the simulator.

Because the simulator can provide a consistent set of shares for any secret that needs to be opened (by using the shares it stored throughout the simulation), and can also choose one of the shares to be whatever it designs, the environment can use neither the sets of shares sent it by the simulator nor the reconstructed secrets themselves to distinguish between worlds. □