Reducing Communication Channels in MPC

Marcel Keller[1], Dragos Rotaru[1,2], Nigel P. Smart[1,2], and Tim Wood[1,2]

[1] University of Bristol, Bristol, UK.
[2] imec-COSIC, KU Leuven, Leuven, Belgium.
mks.keller@gmail.com,Dragos.Rotaru@esat.kuleuven.be,
nigel.smart@kuleuven.be,t.wood@kuleuven.be

Abstract. We show that the recent, highly efficient, three-party honest-majority computationally-secure MPC protocol of Araki et al. can be generalised to an arbitrary Q_2 access structure. Part of the performance of the Araki et al. protocol is from the fact it does not use a complete communication network for the most costly part of the computation. Our generalisation also preserves this property. We present both passively- and actively-secure (with abort) variants of our protocol. In all cases we require fewer communication channels for secure multiplication than Maurer's "MPC-Made-Simple" protocol for Q_2 structures, at the expense of requiring pre-shared secret keys for Pseudo-Random Functions.

1 Introduction

Secret-sharing-based secure MPC (multi-party computation) 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. We will call these access structures (n, t)-*threshold*. To obtain passively-secure protocols one requires $t < n/2$, and to obtain actively-secure protocols one requires $t < n/3$; these conditions are also sufficient. Passive adversaries follow the protocol but possibly try to learn information about other parties' inputs, whereas active adversaries may deviate arbitrarily from the protocol.

These results for 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. These notions will be discussed in more detail later, but in brief an access structure is Q_ℓ if the union of no ℓ unqualified sets is the whole set of parties; for example, an (n, t)-threshold scheme is Q_ℓ if and only if $t < n/\ell$.

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* and one can obtain actively-secure protocols in the threshold case when $t < n/2$ (i.e. honest majority), and active security *with abort* when only one party is honest. Generally speaking, such computationally-secure protocols are less efficient than the information-theoretic protocols as they usually need some form of public-key cryptography.

In recent years there has been considerable progress in *practical* MPC by marrying the two approaches. For example, the BDOZ [BDOZ11], VIFF [DGKN09], SPDZ [DPSZ12] and Tiny-OT [NNOB12] protocols are computationally secure and use information-theoretic primitives in an online phase, but only 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. situations in which only one 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 in the *practically important online phase* are the main bottleneck.

However, recent work has provided a new method to unify information-theoretic and computationally-secure protocols. Araki et al. [AFL+16] provide a very efficient passively-secure MPC evaluation of the AES

circuit in the case of a 1-out-of-3 adversary structure. This was then generalised to an actively secure protocol by Furukawa et al. [FLNW17]. 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.

The passively-secure protocol of [AFL+16] makes use of a number of optimisations to the basic offline/online paradigm. Firstly, the offline phase is only used to produce additive sharings of zero. 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*: each party only sends data to *one* other party via a *secure channel*, and only receives data from *the third* party via a *secure channel*. Thirdly, parties 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 (such as Shamir's) in which each party only holds one finite-field element per secret.

The underlying protocol of Araki et al., 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, they are also 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 how to use cryptographic assumptions to optimise the information-theoretic approach of [Mau06]. The active variant of the protocol given by Furakawa et al. [FLNW17] 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 also runs over an *incomplete network* of *authenticated channels*. The question therefore naturally arises as to whether the approach outlined in [AFL+16], [BLW08] and [FLNW17] is particularly tied to the 1-out-of-3 adversary structure, or whether it generalises to other access/adversary structures.

1.1 Our Work

In this paper we show that the basic passively-secure protocol of Araki et al. generalises to arbitrary 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. Moreover, the generalised protocol offers significant advantages in terms of communication cost when compared to the prior protocols in this setting.

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 [FLNW17] to obtain active security. In particular we utilise our passive protocol as an offline phase, and 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 and in a short setup phase. The proofs of security are also given in the appendix.

The main challenge we meet in attempting to generalise the work of Araki et al. is that it is not immediately clear what the conditions on its shares mean in a wider context; more specifically, their protocol relies heavily on the fact that in the $(3, 1)$-threshold setting replicated shares are necessarily "consistent" and consequently their communication pattern allows errors to be detected in the active variant due to Furukawa et al. [FLNW17].

General, as opposed to threshold, access structures are practically interesting in situations where different groups of parties play different organisational roles. For example consider a financial application where one may have a computation performed between a number of banks and regulators; the required access structures for collaboration between the banks and the regulators may be different. Thus general access structures, such as the Q_2 structures considered in this paper, may have important real-world applications. All protocols have been implemented in the SCALE-MAMBA system[3].

[3] See `https://homes.esat.kuleuven.be/~nsmart/SCALE/` for details.

We now proceed to give a high-level overview of our protocol and its main components. We divide the discussion into looking at the passively secure protocol first, and then give the changes needed to consider the actively secure (with abort) variant.

Passively Secure Protocol: If the access structure is Q_2 then the product of the shared values can be expressed as a linear combination of products of the values held by individual players. Hence, the product can be expressed as the sum of a single value held by each party. This is exploited in the protocol of Maurer to obtain a sharing (in the original replicated scheme) of the product, by each party producing a resharing of their component of the product. Thus multiplication of secrets in the passively-secure protocol of Maurer requires all parties to produce one secret-sharing.

In our protocol we take start as in Maurer's protocol in forming a representation of the product as a full threshold additive secret sharing. We then mask this using a pseudo-random zero sharing (PRZS), and then use the resulting full threshold sharing as a basis for the original replicated sharing. This means that each party need only communicate the share they hold to the other parties which need to replicate it. This produces savings in both the number of elements transmitted and the number of communication channels. In Sections 3.1 and 3.2 we outline and compare Maurer's and our protocol.

In a further optimisation, given in Section 3.3, we reduce the number of channels, which we denote by $\mathcal{G}_\Gamma$, and the required number of finite field elements transmitted, even further. This optimisation, however, comes at the expense of requiring more pre-distributed keys and PRF evaluations. But we present a simple six party access structure for which this optimization that gives a 93 percent saving on transmitted finite-field elements, and a 50 percent saving on the number of secure channels, compared to the original protocol of Maurer.

To obtain the output from our passively protocol we require a full set of either authenticated or secure channels (depending on the specific subprotocol being executed). However, these operations are not performed nearly as often as multiplication operations. It is the high bandwidth requirements of multiplication operations that form a bottleneck in many practical instantiations of MPC protocols.

Actively Secure Protocol: In the appendix we then extend this basic protocol to the case of active security (with abort), again with the objective of minimising the number of pairwise connections and transmitted finite field elements. Our actively-secure protocol follows the paradigm of Furukawa et al. However, we need to make a few small changes to allow for arbitrary Q_2 access structures. We adopt a relatively standard three step approach to obtaining active security.

1. We use our passively-secure multiplication protocol in an offline phase to obtain so-called Beaver triples.
2. These triples are then checked using the trick of sacrificing (see e.g. [BDOZ11]) to ensure that the triples are actually valid, and have not been tampered with by a malicious adversary. This requires communication over a reduced set of *authenticated* channels $\mathcal{H}_\Gamma$.
3. The triples are then used in a standard Beaver-like online phase which is executed over the same subnetwork of *authenticated* channels.

Active security is obtained, as in [AFL+16], by each player hashing their view during a multiplication and comparing the resulting hashes at the end (which requires a complete graph of authenticated channels). We show that this obviates the need for *every* party holding a given share to send it to every party who does not. However, in generalising to arbitrary access structures it is no longer sufficient to hash the view of the *values opened* in the multiplication sub-protocol: one also needs to hash the *vector of shares* used to produce these values. This hash-checking is analogous to the MAC checking in full threshold protocols such as SPDZ [DPSZ12].

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, for the sacrifice step of our actively-secure protocol with abort, that q is sufficiently large to have a cheating detection probability of $1 - 2^{-\mathsf{sec}}$ for a suitable choice of sec; i.e. $q > 2^{\mathsf{sec}}$. If this is not the case, then by repeating our checking procedures $\mathsf{sec}/\log_2 q$ times, we can reduce the cheating probability to $2^{-\mathsf{sec}}$. We do not analyse this aspect in this paper so as to aid the reader in

seeing the main concepts more clearly. 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. In this section we are working over an arbitrary finite field $\mathbb{F}_q$ where q is a prime power, although our protocols also work over any finite ring R. For any $n \in \mathbb{N}$ we denote the set $\{1, \ldots, n\}$ by $[n]$. We denote the computational security parameter by λ and the statistical security parameter by sec.

2.1 Access Structures and Secret Sharing

Access Structures. Let $\mathcal{P}$ denote the set of parties, $\mathcal{P} = [n]$, and let $\Gamma, \Delta \in 2^{\mathcal{P}}$. If $\Gamma \cap \Delta = \varnothing$ then we call the pair (Γ, Δ) an access structure. We call a set of parties $B \in \Gamma$ qualified, and a set in $A \in \Delta$ unqualified. As is typical in the literature, we assume monotonicity of the access structure: supersets of qualified sets are qualified, and subsets of unqualified sets are unqualified. The access structure is said to be *complete* if $\Delta = 2^{\mathcal{P}} \setminus \Gamma$, (i.e. every set of parties is either qualified or unqualified), and in this case we will sometimes just write Γ for the access structure instead of the pair.

A set of parties $A \in \Delta$ is called *maximally* unqualified if Δ contains no proper supersets of A. For a complete access structure, this implies that adding any party not already in A makes the set qualified. We denote by $\mathcal{M} \subseteq \Delta$ the set of maximally unqualified sets. Similarly, 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 $\mathcal{B} = \{\mathcal{P} \setminus M : M \in \mathcal{M}\}$. Note that, in general, it is not true that the set $\mathcal{B}$ is equal to the set of minimally qualified sets.

Q_ℓ Access Structures. The set Δ, called the adversary structure, is said to be Q_ℓ (for **q**uorum), where $\ell \in \mathbb{N}$, if no set of 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. In fact, if $\mathcal{M}$ is Q_2 then Δ is 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 called 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. Hirt and Maurer's result is realised by showing that if an access structure is Q_2 then it can be realised by a multiplicative secret sharing scheme, i.e. given two secret shared values $[\![a]\!]$ and $[\![b]\!]$, the product $a \cdot b$ can be represented as a linear combination of the elements in the *local* Schur products

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

The fact that by local computations the parties each obtain one summand of the product is the reason one is able to build an MPC protocol secure against passive adversaries for any Q_2 access structure. For the details, we refer the reader to [HM00].

Replicated Secret Sharing. Given a monotone access structure (Γ, Δ), we will make extensive use of the replicated secret sharing scheme which respects it. Let $\mathcal{B}$ be, as above, the set of sets which are complements of maximally unqualified sets in the access structure. Then to share secret x, a set of shares $\{x_B\}_{B \in \mathcal{B}}$ is sampled uniformly at random from the field subject to $x = \sum_{B \in \mathcal{B}} x_B$ and x_B is given to each player 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 $S_{x,i} = \{x_B : i \in B \text{ and } B \in \mathcal{B}\}$. Since every unqualified set is a (not necessarily proper) subset of a maximally unqualified set, every set of unqualified parties is missing at least one member of the set $\{x_B\}_{B\in\mathcal{B}}$, and hence these parties learn no information about the secret. Replicated secret-sharing is therefore *perfect*, which is defined to mean that no unqualified set of parties has any advantage over uniformly guessing the secret. Conversely, a qualified set A of parties is not a subset of any $M \in \mathcal{M}$ (i.e., for every $M \in \mathcal{M}$, A contains some i where $i \notin M$), and hence for every $B \in \mathcal{B}$, there is at least one party in A which receives the share x_B.

To see that a replicated secret-sharing scheme is multiplicative if the access structure it realises is Q_2, observe that given secrets x and y, for every pair of 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 by definition of Q_2. Then party i can compute the cross terms $x_{B_1} \cdot y_{B_2}$ and $x_{B_2} \cdot y_{B_1}$ (and also the diagonal terms $x_{B_1} \cdot y_{B_1}$ 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 to achieve this is the main goal of this paper. Note also that there may be multiple parties in the intersection of two sets in $\mathcal{B}$, but we only require one of these parties to include the term in their computation.

Example. We will use the following example later to demonstrate the savings which can result from our method and also to examine the communication channels in the next paragraph. Consider the following set of maximally unqualified sets for a six-party access structure, which we shall use as a running example throughout this section.

$$\mathcal{M} = \Big\{\{2,5,6\},\{3,5,6\},\{4,5,6\},\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{1,6\},\\ \{2,3\},\{2,4\},\{3,4\}\Big\}.$$

Here the set $\mathcal{B}$ becomes

$$\mathcal{B} = \Big\{\{1,3,4\},\{1,2,4\},\{1,2,3\},\{3,4,5,6\},\{2,4,5,6\},\{2,3,5,6\},\\ \{2,3,4,6\},\{2,3,4,5\},\{1,4,5,6\},\{1,3,5,6\},\{1,2,5,6\}\Big\}.$$

As stated above, in replicated secret sharing a secret s is shared as an additive sum $s = \sum_{B\in\mathcal{B}} s_B$, with party i holding value s_B if and only if $i \in B$.

Redundancy. A redundant player is one whose shares are not *necessarily* needed to reconstruct the secret (if it is shared using replicated secret-sharing), and so one could define an MPC protocol achieving the same (passive) security by ignoring this player entirely in the computation and just providing it with the final output. To provide a more formal definition, consider the replicated scheme above: if there is a party $i \in \mathcal{P}$ for which there exists some other party $j \in \mathcal{P}$ such that for all $B \in \mathcal{B}$ we have $i \in B$ implies $j \in B$, then every share given to party i is also given to party j, and hence we consider party i redundant.

For an access structure Γ with set $\mathcal{M}$ of maximal unqualified sets, we define party i to be redundant if for every $M \in \mathcal{M}$ there exists $j \in \mathcal{P} \setminus \{i\}$ such that $i \notin M$ implies $j \notin M$, and non-redundant otherwise; equivalently, i is non-redundant if for every $j \in \mathcal{P}$ there exists $M \in \mathcal{M}$ such that $i \notin M$ but $j \in M$, and we say that Γ is non-redundant if every party in $\mathcal{P}$ is non-redundant.

For example, consider the set of maximally unqualified sets over $\mathcal{P} = [4]$ given by $\mathcal{M} = \{\{1\},\{2\},\{3,4\}\}$. 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 one the shares $\{s_{134}, s_{12}\}$, player two $\{s_{234}, s_{12}\}$, player three $\{s_{234}, s_{134}\}$ and player four $\{s_{234}, s_{134}\}$. Both shares obtained by player

three are also obtained by player four, so we can essentially ignore player four in any protocol design and just provide the output to this player at the end.

Note that if any party is omitted from all sets in $\mathcal{M}$ then it is present in all sets in $\mathcal{B}$ and hence every party, but this party, is redundant, which makes the MPC protocol trivial: the omitted party can simply perform the entire computation itself and output the result to all parties.

Partition. In our protocol, we partition the set $\mathcal{B}$ into sets indexed by the parties $\{\mathcal{B}_i\}_{i\in\mathcal{P}}$ such that for every $i \in \mathcal{P}$ we have $B \in \mathcal{B}_i$ implies $i \in B$. To make this assignment of sets in $\mathcal{B}$ to parties, we consider all the maps $f : \mathcal{B} \to \mathcal{P}$ such that for every $i \in \mathcal{P}$, $f(B) = i$ implies $i \in B$, and choose an f such that $\mathrm{Im}(f)$ is as large as possible; then for each $i \in \mathcal{P}$ we let $\mathcal{B}_i = f^{-1}(i)$, where $f^{-1}(i)$ denotes the preimage of i under the map f.

Note that if f is not surjective then there is at least one set $\mathcal{B}_i$ (for some i) which is empty. For the rest of the main body of this paper, we assume that $\mathcal{B}_i$ is *not* empty for all i, since for small numbers of parties on a non-redundant Q_2 access structure, we can always find a surjective f. For the necessary adaptation to the protocol when this is not the case, and further relevant discussion, see Section 5.

Note that in general non-redundancy implies a lower bound on the size of $\mathcal{M}$: let n' be the number of parties which are *not* maximally unqualified sets as singleton sets, and let x be the number of sets in $\mathcal{M}$. The lower bound on number of sets there must be in $\mathcal{M}$ so that there is no redundancy amongst these n' parties is the number of ways of putting each party into at least two sets so that for every pair of parties there is a set containing one and not the other. Thus we require $\binom{x}{2} \geq n'$, which means that $x \geq \frac{1+\sqrt{1+8n'}}{2}$. Since there are more sets in $\mathcal{M}$ for non-redundant access structures, it becomes "easier" to find the surjective maps f required by our main protocol.

In our earlier six party example we could set the partition to be

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

Channel Sets. Given the above partition of $\mathcal{B}$ we define the following graphs of channels:

$$\mathcal{G}_\Gamma = \bigcup_{i\in\mathcal{P}} \bigcup_{B\in\mathcal{B}_i} \bigcup_{j\in B\setminus\{i\}} \{(i,j)\}$$

$$\mathcal{H}_\Gamma = \bigcup_{i\in\mathcal{P}} \bigcup_{B\in\mathcal{B}_i} \bigcup_{j\notin B} \{(i,j)\}$$

Our (passively-secure) multiplication protocol makes use of the set of secure channels denoted by $\mathsf{SC}(\mathcal{G}_\Gamma)$, namely $(i,j) \in \mathsf{SC}(\mathcal{G}_\Gamma)$ implies that party i is connected to party j by a uni-directional secure channel. The sacrificing step and online multiplication protocol in our actively secure protocol requires communication over an authentic set of channels $\mathsf{AC}(\mathcal{H}_\Gamma)$, where $(i,j) \in \mathsf{AC}(\mathcal{H}_\Gamma)$ implies that party i is connected to party j by an authenticated channel.

The key operation in both sacrificing and the online phase is being able to open a value to all parties in an authenticated manner. Publicly opening a secret requires every party to receive every share it does not have from at least one other party holding that share. Thus the definition of $\mathcal{H}_\Gamma$.

2.2 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 $\langle v \rangle$. This type of secret-sharing does not respect a Q_2 access structure since all shares are required to determine the secret, 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 (PRZSs) for the additive secret sharing scheme $\langle v \rangle$, and pseudo-random secret sharings (PRSSs) for the replicated scheme $[\![v]\!]$. Note, we could produce these using additional interaction, but recall our goal is to reduce communication. In particular, we make black-box use of the functionality given in Figure 1. Pseudo-random secret sharings, and pseudo-random zero sharings in particular, for arbitrary access structures can involve a set-up phase requiring the agreement of exponentially-many keys in general. The protocol is given in [CDI05] and so we omit it here, though the reader may refer to Appendix A for an overview of our variant (specialised for replicated secret-sharing).

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

Set-up: The functionality accepts Initialise or Abort from all parties and the adversary. If any party inputs Abort, the functionality sends the message Abort to all parties.

PRZS:
- On input PRZS(count) from all parties, 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 $t_i \xleftarrow{\$} \mathbb{F}$ uniformly at random, fixes $t_n \leftarrow -\sum_{i=1}^{n-1} t_i$ and sends t_i to party i for each $i \in \mathcal{P}$.

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

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

3 Passively-Secure MPC Protocol

In this section we outline our optimisation 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. As stated in Section 2, we will assume 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 call such a partition (where $\mathcal{B}_i \neq \varnothing$ for all i) a *surjective partition*; when this is not possible we provide the requisite alterations to the protocol in Section 5. We consider $\mathcal{B}_i$ to be the set of sets for which party i will be "responsible".

3.1 Maurer's "MPC-Made-Simple" Protocol

The information-theoretic protocol we describe is based on 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. For comparison with our protocol, we explain Maurer's protocol here.

We 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 there are possibly multiple choices for ρ, and that it is certainly not true in general that $i = \rho(B_1, B_2)$ implies $B_1 \cap B_2 = \{i\}$ (though clearly $B_1 \cap B_2 \supseteq \{i\}$). 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
$$v_i \leftarrow \sum_{B_1, B_2 \in \mathcal{B} \,:\, \rho(B_1, B_2) = i} x_{B_1} \cdot y_{B_2}$$
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 $|\mathcal{B}|$ 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. $l = \sum_{B \in \mathcal{B}} |B| / |\mathcal{B}|$, then it is clear that the total communication required is $n \cdot |\mathcal{B}| \cdot l$ finite-field elements.

In fact each party i sends a total of

$$\sum_{B \in \mathcal{B} : B \ni i} (|B| - 1) + \sum_{B \in \mathcal{B} : B \not\ni i} |B| = \sum_{B \in \mathcal{B}} |B| - \sum_{B \in \mathcal{B} : B \ni i} 1$$

finite-field elements, and hence the total communication (for all parties) for one multiplication is

$$\sum_{i=1}^{n} \left(\sum_{B \in \mathcal{B}} |B| - \sum_{B \in \mathcal{B} : B \ni i} 1 \right) = (n-1) \cdot \sum_{B \in \mathcal{B}} |B|$$

finite-field elements over $n \cdot (n-1)$ uni-directional secure channels[4]. For our example Q_2 access structure this translates into sending $(6-1) \cdot 41 = 205$ finite-field elements over $6 \cdot 5 = 30$ secure channels. Note that the same finite-field element will be sent to multiple parties (every set of parties $B \in \mathcal{B}$ obtains a share common to them all), but we count these elements as distinct when analysing communication costs.

3.2 New Protocol

Our protocol is largely the same as Maurer's, with one major difference: in our protocol, the parties do not each create a replicated sharing of the partial product v_i – instead, they do the following. Notice that the v_i form an additive sharing $\langle z \rangle$ of the sum. Our basic idea is first to re-randomise this sum using a PRZS, and then to consider each re-randomised v_i as one share of the new secret (namely, the product of the previous two secrets), i.e. consider each share v_i as 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.

Our method directly generalises 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. It results in each party not needing to be connected to each other party by a secure channel. The total number of distinct finite field elements transmitted for a threshold structure via this method is then $O(n \cdot 2^n)$, as opposed to the $O(n^2 \cdot 2^n)$ of Maurer's protocol. For other Q_2 structures the saving in communication is more significant, as our earlier example demonstrates.

[4] Note, as is common in security systems we assume channels are uni-directional; as good security practice is to have different secret keys securing communication in different directions so as to avoid various reflection attacks etc. This is exactly how TLS and IPSec secure channels are configured.

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 the following:

1. Party i computes
$$v_i \leftarrow \sum_{B_1, B_2 \in \mathcal{B} \,:\, \rho(B_1,B_2)=i} x_{B_1} \cdot y_{B_2}$$
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 Figure 1; thus party i holds t_i such that $\sum_{i=1}^{n} t_i = 0$.
3. Party i samples u_B for $B \in \mathcal{B}_i$ such that $\sum_{B \in \mathcal{B}_i} u_B = v_i + t_i$.
4. Party i sends, for all $B \in B_i$, the value u_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 $\sum_{B \in \mathcal{B}}(|B| - 1)$. Also notice that we obtain a valid sharing of the product as we have assumed $\mathcal{B}_i \neq \varnothing$, and thus every share v_i has been utilised in the final sharing.

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

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

Multiply: On input $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, 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 $i = 0$, and to player i only otherwise.

Figure 2. 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 arbitrary finite fields; see Figure 3 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 2. That the protocol securely implements this functionality is given by the following theorem, whose proof is given in Appendix B.

Theorem 1. *Suppose we have a non-redundant Q_2 access structure with a surjective partition $\{\mathcal{B}_i\}$ of the set $\mathcal{B}$. Then the protocol Π_{PMPC} securely realises the functionality $\mathcal{F}_{\mathsf{PMPC}}$ against passive adversaries in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model*[5]*.*

Assuming a surjective partition, the protocol requires at most $\sum_{B \in \mathcal{B}}(|B| - 1)$ field elements of communication, over $|\mathcal{G}_\Gamma|$ secure channels, per multiplication gate, and the same number to perform the input procedure.

[5] The alterations to the protocol for when there is no surjective partition are discussed in Section 5.

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 that 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 usually a *strict* subset of the sets containing i).

Set-up: The parties set $\mathsf{count} \leftarrow 0$.

Input: For party i to provide input x,
1. The parties call $\mathcal{F}_{\mathsf{Rand}}$ with input $\mathsf{PRZS}(\mathsf{count})$ so that each player $j \in \mathcal{P}$ obtains t_j such that $\sum_{j \in \mathcal{P}} t_j = 0$.
2. Party i samples $\{u_B\}_{B \in \mathcal{B}_i} \leftarrow \mathbb{F}$ such that $\sum_{B \in \mathcal{B}_i} u_B = x + t_i$.
3. For each $j \in \mathcal{P} \setminus \{i\}$, party j samples $\{u_B\}_{B \in \mathcal{B}_j} \leftarrow \mathbb{F}$ such that $\sum_{B \in \mathcal{B}_j} u_B = t_j$.
4. For each $j \in \mathcal{P}$, for each $B \in \mathcal{B}_j$, for each $k \in B$, party j sends u_B securely to party k.
5. The parties increment count by one.

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 $v_i \leftarrow \sum_{B_1, B_2 \in \mathcal{B} \,:\, \rho(B_1, B_2) = i} x_{B_1} \cdot y_{B_2}$.
2. The parties call $\mathcal{F}_{\mathsf{Rand}}$ with input $\mathsf{PRZS}(\mathsf{count})$ 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 $i \in \mathcal{P}$, for each $B \in \mathcal{B}_i$, for each $j \in B \setminus \{i\}$, party i securely sends the value u_B to party j.
5. The parties increment count by one.

Output$([\![x]\!], i)$:
1. If $i \neq 0$, for each $j \in \mathcal{P}$, for each $B \in \mathcal{B}_j$, party j securely sends x_B to i if $i \notin B$. 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 3. Protocol Π_{PMPC}

In our previous six party example we have

$$\mathsf{SC}(\mathcal{G}_\Gamma) = \Big\{(1,3),(1,4),(2,1),(2,4),(3,1),(3,2),(4,2),(4,3),(4,5),\\ (5,1),(5,2),(5,3),(5,4),(5,6),(6,2),(6,3),(6,4),(6,5)\Big\}.$$

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 transmitted, and 40 percent on the number of secure channels needed.

3.3 An Optimisation

We end this section by presenting a minor optimisation of our passively secure multiplication protocol, which can result in a further reduction in both the number of communication channels and the number of finite-field elements that need to be sent. However, this comes at the expense of needing further PRF evaluations.

Recall that to each player i we associated a set $\mathcal{B}_i$, of sets B for which player i is "responsible" for producing the sharing u_B during the multiplication protocol. In our optimisation we make player i responsible for only a single set, which we call B_i, which is an element of $\mathcal{B}_i$. All other values u_B for $B \in \mathcal{B}_i \setminus \{B_i\}$ are generated by a PRF evaluation.

We informally describe the extensions needed here in the case of a surjective partition; the extension to non-surjective partitions is immediate. First we extend the $\mathcal{F}_{\mathsf{Rand}}$ functionality so that it contains an additional command $\mathcal{F}_{\mathsf{Rand}}.\mathbf{Rand}(B)$. This command, on input of a set of players B, will output the same uniformly random value z_B to all players in B. Clearly, this additional command is a component of the existing command $\mathcal{F}_{\mathsf{Rand}}.\mathbf{PRSS}$, and so can be implemented in the same way.

Our optimisation of the multiplication protocol is then given in Figure 4. It is then clear that we need to transmit only n distinct, finite-field elements over the set

$$\widehat{\mathcal{G}_\Gamma} = \bigcup_{i \in \mathcal{P}} \bigcup_{j \in B_i \setminus \{i\}} \{(i,j)\} \subseteq \mathcal{G}_\Gamma$$

of secure channels, which we denote by $\mathsf{SC}(\widehat{\mathcal{G}_\Gamma})$. The total number of (non-distinct) finite fields elements that need to be sent is equal to $\sum_{i=1}^{n}(|B_i| - 1)$.

Optimised Passively Secure Multiplication Protocol

Multiply:

1. For each $i \in \mathcal{P}$, party i computes $v_i \leftarrow \sum_{B_1, B_2 \in \mathcal{B} \,:\, \rho(B_1,B_2)=i} x_{B_1} \cdot y_{B_2}$.
2. The parties call $\mathcal{F}_{\mathsf{Rand}}.\mathbf{PRZS}$ so that each player $i \in \mathcal{P}$ obtains t_i such that $\sum_{i \in \mathcal{P}} t_i = 0$.
3. For each $B \in \mathcal{B}_i \setminus \{B_i\}$ the players execute $\mathcal{F}_{\mathsf{Rand}}.\mathbf{Rand}(B)$, so that each player $j \in B$ obtains a uniformly random element u_B.
4. Party i defines u_{B_i} by setting $u_{B_i} \leftarrow v_i + t_i - \sum_{B \in \mathcal{B}_i \setminus \{B_i\}} u_B$.
5. For each $i \in \mathcal{P}$, party i sends the value u_{B_i} securely to party j for all $j \in B_i$.

Figure 4. Optimised Passively Secure Multiplication Protocol

When specialised to our six-party example from the introduction, and taking $B_5 = \{1,2,5,6\}$ and $B_6 = \{2,3,4,6\}$ (with the obvious definition of B_1, B_2, B_3, and B_4), we find

$$\widehat{\mathcal{G}_\Gamma} = \Big\{(1,3),(1,4),(2,1),(2,4),(3,1),(3,2),(4,2),(4,3),(4,5),\\ (5,1),(5,2),(5,6),(6,2),(6,3),(6,4)\Big\}.$$

Thus we need to send only 15 finite fields elements over 15 uni-directional secure channels. This equates to a bandwidth saving of an additional 50 percent over our initial protocol, and a 17 percent saving over the number of secure channels. Compared to the initial protocol of Maurer we obtain a saving of 93 percent in the number of transmitted finite field elements, and a saving of 50 percent in the number of secure channels.

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 [FLNW17]. 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 [FLNW17] 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. Note that although the hash function is not guaranteed to be hiding we only hash shares that are opened to each party, hence no extra information is leaked. In [FLNW17] 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.

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

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

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 $\mathsf{SC}(\mathcal{H}_\Gamma)$ (defined in the introduction) of secure channels to implement $\mathbf{Reveal}(\llbracket x \rrbracket, i)$ for all $i \neq 0$.

Note that, in $\mathbf{Reveal}(\llbracket x \rrbracket, i \neq 0)$ 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 method is used in $\mathbf{Reveal}(\llbracket x \rrbracket, 0)$ (i.e. when a secret is opened to everyone) but is achieved 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. Note that since there is at least one set in $\mathcal{B}$ which contains no corrupt parties, the functionality is able to choose shares indexed by (at least) one $B \in \mathcal{B}$ with $B \cap \mathcal{A} = \varnothing$ so that the triple generated is correct regardless of what shares the adversary sent the functionality.

The protocol Π_{Triple} in Figure 7 implements the $\mathcal{F}_{\mathsf{Triple}}$ functionality, with Figure 7 itself using 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 C.

Theorem 2. *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 $|\mathcal{G}_\Gamma|$ secure channels to execute the passively secure multiplication protocol, which is at the core of the **Triple Generation** step, and $|\mathcal{H}_\Gamma|$ 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: $H_i \leftarrow \mathsf{Init}()$ 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}

4.1 Actively Secure Online 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 of which we give in Appendix C.

Theorem 3. *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 $|\mathcal{G}_\Gamma|$ secure channels in the offline phase (i.e. for Π_{Triple}), and requires $|\mathcal{H}_\Gamma|$ authenticated channels in the online phase for the multiplication operation.

Inputting values requires $|\mathcal{H}_\Gamma|$ 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 requires $|\mathcal{H}_\Gamma|$ secure channels, or $|\mathcal{H}_\Gamma|$ authenticated channels if only public output is required.

In our running six party example the set of authenticated channels is given by

$$\mathsf{AC}(\mathcal{H}_\Gamma) = \Big\{(1,2),(1,5),(1,6),(2,3),(2,5),(2,6),(3,4),(3,5),(3,6),(4,1),\\ (4,6),(5,2),(5,3),(5,4),(6,1),(6,2),(6,3),(6,3),(6,5)\Big\}.$$

Each online multiplication requires opening a value to all parties, thus each online multiplication will require a total of

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

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 (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
   $$(\{a_B^{(i)}, b_B^{(i)} : B \in \mathcal{B} \text{ s.t. } k \in B\})_{k\in\mathcal{A}}.$$
   [This is a tuple of shares, indexed by the corrupt parties, 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.]
   (f) If Abort is false the functionality samples $\{c_B^{(i)} : B \in \mathcal{B} \text{ s.t. } B\cap\mathcal{A} = \varnothing\}$ so that
   $$\left(\sum_{B\in\mathcal{B}} a_B^{(i)}\right) \cdot \left(\sum_{B\in\mathcal{B}} b_B^{(i)}\right) = \left(\sum_{B\in\mathcal{B}} c_B^{(i)}\right).$$
2. 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 $\mathcal{F}_{\mathsf{Triple}}$

finite-field elements to be sent over these authenticated secure channels. Thus each multiplication requires the transmission of $6 \cdot 11 - (3 \cdot 3 + 8 \cdot 4) = 25$ finite-field elements, over 19 authenticated channels. However, opening output values and receiving inputs still requires a complete network of authenticated channels. But, again, these are operations performed less often than the basic multiplication operation.

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 behaviour on malicious parties can result in a reduction of uni-directional authenticated channels from $n \cdot (n-1)$ down to $n \cdot t$ for the online phase.

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 $\mathcal{F}_{\mathsf{Rand}}$.**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 $\mathcal{F}_{\mathsf{Rand}}$.**PRZS** so that player j obtains t_j subject to $\sum_{l \in \mathcal{P}} t_l = 0$.
   (d) For each $j \in \mathcal{P}$, party j samples $c^{(i)}_B$ for $B \in \mathcal{B}_j$ subject to $\sum_{B \in \mathcal{B}_j} c^{(i)}_B = v_j + t_j$.
   (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 Π_{Open}.**Init**.
2. For $i = 1, \ldots, n_T$ do
   (a) The parties run $\mathcal{F}_{\mathsf{Rand}}$.**PRSS** to obtain $[\![r^{(i)}]\!]$.
   (b) The parties run Π_{Open}.**Reveal**$([\![r^{(i)}]\!], 0)$ to obtain $r^{(i)}$.
   (c) The parties run Π_{Open}.**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 Π_{Open}.**CompareView**, and if Abort is not set to true then they perform Π_{Open}.**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 Π_{Open}.**CompareView** again.
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}

5 Passive Multiplication Protocol when f is not Surjective

We now describe the modifications to our basic protocol when we cannot find a partition of the set $\mathcal{B}$ into non-empty sets $\{\mathcal{B}_i\}_{i \in [n]}$ such that $i \in B$ for all $B \in \mathcal{B}_i$. We also work out how this change affects our overall consumption of bandwidth, and the number (and type) of communication channels. For efficiency, we first select any map $f : \mathcal{B} \longrightarrow \mathcal{P}$ for which $\mathrm{Im}(f)$ is as large as possible.

Recall that our basic protocol works in the case that $\mathrm{Im}(f) = \mathcal{P}$. The modification is simply to apply the standard protocol for all $i \in \mathrm{Im}(f)$, and apply Maurer's protocol when $i \notin \mathrm{Im}(f)$. The multiplication protocol then becomes:

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 $\mathcal{F}_{\mathsf{Rand}}$.**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 $i \in \mathrm{Im}(f)$
   (a) Party i samples $\{u_B\}_{B \in \mathcal{B}_i} \leftarrow \mathbb{F}$ such that $\sum_{B \in \mathcal{B}_i} u_B = v_i + t_i$.
   (b) Party i sends, for all $B \in \mathcal{B}_i$, the value u_B securely to party j for all $j \in B \setminus \{i\}$.
4. For each $i \notin \mathrm{Im}(f)$

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

Multiply: On input $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, 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 $\mathcal{F}_{\mathsf{AMPC}}$

(a) Party i samples $\{s^i_B\}_{B\in\mathcal{B}} \leftarrow \mathbb{F}$ such that $\sum_{B\in\mathcal{B}} s^i_B = v_i + t_i$. Note that the sum is over all $B \in \mathcal{B}$ not $B \in \mathcal{B}_i$ (which by assumption is empty).
(b) Party i sends, for all $B \in \mathcal{B}$, the value s^i_B securely to party j for all $j \in B\setminus\{i\}$. Note, the transmission is over all $B \in \mathcal{B}$ not $\mathcal{B}_i$.

5. Party i for each $B \in \mathcal{B}$ with $i \in B$ computes

$$z_B = u_B + \sum_{j\notin \mathrm{Im}(f)} s^j_B.$$

The fact that the multiplication protocol is correct and secure can be easily verified. The only issue is to adapt our formulae for the number of secure and authenticated channels needed. Instead of the graph $\mathcal{G}_\Gamma$, we have

$$\widetilde{\mathcal{G}_\Gamma} = \left(\bigcup_{i\in\mathrm{Im}(f)}\bigcup_{B\in\mathcal{B}_i}\bigcup_{j\in B\setminus\{i\}}\{(i,j)\}\right)\bigcup\left(\bigcup_{i\notin\mathrm{Im}(f)}\bigcup_{B\in\mathcal{B}}\bigcup_{j\in B\setminus\{i\}}\{(i,j)\}\right).$$

and hence we require the set $\mathsf{SC}(\widetilde{\mathcal{G}_\Gamma})$ of secure channels. The number of finite-field elements needed to be transmitted in our passively secure protocol above becomes

$$\left(\sum_{B\in\mathcal{B}}(|B|-1)\right) + \sum_{i\notin\mathrm{Im}(f)}\left(\sum_{B\in\mathcal{B}:B\ni i}(|B|-1) + \sum_{B\in\mathcal{B},B\not\ni i}|B|\right).$$

Recall that for the set of authenticated channels $\mathcal{H}_\Gamma$, needed in the actively secure variant, we just need to guarantee that every party receives one share from at least one player. Hence, each party in $\mathcal{P}\setminus\mathrm{Im}(f)$ can receive all their required values from any one of the parties in $\mathrm{Im}(f)$. Thus, instead of $\mathcal{H}_\Gamma$, we have

$$\widetilde{\mathcal{H}_\Gamma} = \left(\bigcup_{i\in\mathrm{Im}(f)}\bigcup_{B\in\mathcal{B}_i}\bigcup_{j\notin B}\{(i,j)\}\right).$$

and hence a set $\mathsf{AC}(\widetilde{\mathcal{H}_\Gamma})$ of authenticated channels is needed in place of $\mathcal{H}_\Gamma$ in our actively secure protocol.

We end this section by showing that the above protocol is not vacuous, by giving an example of an access structure for which no surjective partition f exists. Consider the access structure with the following set of

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. $H_i \leftarrow \mathsf{Init}()$ 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 $\mathcal{F}_{\mathsf{Rand}}$.**PRSS** to obtain a sharing $[\![r]\!]$.
2. The parties call Π_{Open}.**Reveal**$([\![r]\!], i)$ to open r to player i.
3. The players execute Π_{Open}.**Broadcast**(i, ϵ) 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 Π_{Open}.**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 Π_{Open}.**CompareView**.
2. If Abort is true then the parties output $\perp$ and stop.
3. If $i \neq 0$ then the parties call Π_{Open}.**Reveal**$([\![x]\!], i)$. If Abort is not set to true, the party outputs x.
4. If $i = 0$ then the parties call Π_{Open}.**Reveal**$([\![x]\!], 0)$ to open x, and then Π_{Open}.**CompareView**. If Abort is true then the parties output $\perp$ and stop, otherwise they output x.

[a] This computation is done by fixing some set $B \in \mathcal{B}$ and then the parties holding r_B adding ϵ to r_B; the rest of the shares are left as they are.

Figure 9. The Protocol Π_{Online}

maximally unqualified sets

$$\mathcal{M} = \{\{1,2,4\},\{1,3,5\},\{2,3\},\{4,5\},\{6\}\}$$

and the following set of minimally qualified sets:

$$\{\{1,6\},\{2,5\},\{3,4\},\{1,2,3\},\{1,4,5\},\{2,6\},\{3,6\},\{4,6\},\{5,6\}\}.$$

One can check that every set in $2^{[6]}$ is a subset or superset of at least one maximally unqualified or minimally qualified set, which determines whether or not the set is qualified, hence these sets suffice to define the entire access structure. It is easily verified that this access structure is Q_2 and contains no redundant players. However, since there are only five sets in $\mathcal{M}$, there is no surjective map f from the five sets in $\mathcal{B}$ to the six parties in $\mathcal{P}$.

6 Summary

To make clear what channels are required when, and how many, we provide Table 1. Following the standard mathematical notation, we use $\mathcal{K}_n$ to denote the complete graph on n vertices (i.e. parties) so that, for

example, $\mathsf{SC}(\mathcal{K}_n)$ means that the n parties are connected in a complete network of secure channels. The table presents the costs in terms of the sets of edges $\mathcal{K}_n$, $\mathcal{G}_\Gamma$ and $\mathcal{H}_\Gamma$. Apart from the first set, the cardinalities of these sets depend crucially on the precise access structure one is considering, so it is not possible to give formulae describing their size. However, since $\mathcal{G}_\Gamma$ and $\mathcal{H}_\Gamma$ are strict subsets of $\mathcal{K}_n$, we always obtain benefits over the naive protocol(s).

Protocol	Procedure	Channels required
Π_{Rand}	Set-up	$\mathsf{SC}(\mathcal{K}_n)$
	PRSS	n/a
	PRZS	n/a
Passive Protocol	Input	$\mathsf{SC}(\mathcal{G}_\Gamma)$
	Multiplication	$\mathsf{SC}(\mathcal{G}_\Gamma)$
	Output to one	$\mathsf{SC}(\mathcal{K}_n)$
	Output to all	$\mathsf{AC}(\mathcal{K}_n)$
Active Offline Protocol	Triple Gen.	$\mathsf{SC}(\mathcal{G}_\Gamma)$
	Triple Sac.	$\mathsf{AC}(\mathcal{H}_\Gamma)$
	Authentication check	$\mathsf{AC}(\mathcal{K}_n)$
Active Online Protocol	Input	$\mathsf{SC}(\mathcal{H}_\Gamma) + \mathsf{AC}(\mathcal{K}_n)$
	Multiplication	$\mathsf{AC}(\mathcal{H}_\Gamma)$
	Output to one	$\mathsf{SC}(\mathcal{H}_\Gamma) + \mathsf{AC}(\mathcal{K}_n)$
	Output to all	$\mathsf{AC}(\mathcal{H}_\Gamma) + \mathsf{AC}(\mathcal{K}_n)$

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 in the online phase and that it only requires a complete authenticated network for the hash-comparison stage only.

The set-up of the protocol Π_{Rand} is a one-time offline phase used to generate sharings of random values at various points for zero communication cost. While it requires a complete network of secure channels, the main bottleneck in secret-sharing-based MPC is in multiplication, for which our protocol significantly reduces the communication cost.

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 the corresponding secret-sharing scheme. In particular in the (n, t)-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.

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.

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, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 805–817, Vienna, Austria, October 24–28, 2016. ACM Press.

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

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.

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.

FLNW17. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, Paris, France, May 8–12, 2017. Springer, Heidelberg, Germany.

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.

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.

A PRSSs and PRZSs

For the simple additive secret-sharing scheme it is relatively easy to construct a non-interactive method for producing PRZSs, assuming access to a commitment functionality $\mathcal{F}_{\mathsf{Commit}}$. The commitment functionality is a standard two party functionality allowing one party to first commit, and then decommit, to a value towards another party. We assume that the opened commitment is only available to the recieving party (i.e. it is sent over a secure channel).

The standard setup to generate PRZSs for additive secret sharin requires that each party i ends up sharing 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 $\mathbb{F}_q$. We can use exactly the same setup to obtain the requires PRSS for the replicated scheme, as described in Figure 10.

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. Each pair of parties initialises an instance of $\mathcal{F}_{\mathsf{Commit}}$; call it $\mathcal{F}_{\mathsf{Commit}}^{\{i,j\}}$.
3. For each $i \in \mathcal{P}$, for each $j \in \mathcal{P} \setminus \{i\}$, the parties call $\mathcal{F}_{\mathsf{Commit}}^{\{i,j\}}$ where i submits input $(\mathsf{Commit}, \mathsf{id}_{i,j}, \kappa_{i,j})$ and j gives $(\mathsf{Commit}, \mathsf{id}_{i,j}, \perp)$.
4. For each $i \in \mathcal{P}$, for each $j \in \mathcal{P} \setminus \{i\}$, the parties execute $\mathcal{F}_{\mathsf{Commit}}^{\{i,j\}}$ with input $(\mathsf{Open}, \mathsf{id}_{i,j})$ and abort if $\mathcal{F}_{\mathsf{Commit}}$ returns Abort.
5. For each $B \in \mathcal{B}$, each party $i \in B$ samples $\kappa_{B,i} \xleftarrow{\$} \{0,1\}^\lambda$.
6. Each set of parties $B \in \mathcal{B}$ initialises an instance of $\mathcal{F}_{\mathsf{Commit}}$; call it $\mathcal{F}_{\mathsf{Commit}}^{B}$.
7. For each $B \in \mathcal{B}$, for each $i \in B$, the parties call $\mathcal{F}_{\mathsf{Commit}}^{B}$ where i submits input $(\mathsf{Commit}, \mathsf{id}_{B,i}, \kappa_{B,i})$ and all $j \in B \setminus \{i\}$ give $(\mathsf{Commit}, \mathsf{id}_{B,i}, \perp)$.
8. For each $B \in \mathcal{B}$ and for each $i, j \in \mathcal{P}$ with $i, j \in B$, the parties i and j call $\mathcal{F}_{\mathsf{Commit}}^{B}$ with input $(\mathsf{Open}, \mathsf{id}_{B,i})$ and abort if $\mathcal{F}_{\mathsf{Commit}}^{B}$ returns Abort.
9. Each party i sets $\kappa_B \leftarrow \oplus_{j \in B} \kappa_{B,j}$ for each B containing i.
10. Finally, each party i sets $\mathsf{count} \leftarrow 0$, and $\mathsf{count}_B \leftarrow 0$ for all $B \in \mathcal{B}$ where $i \in B$.

PRZS: Party i retrieves count from memory and computes

$$t_i \leftarrow \sum_{j \neq i} F_{\kappa_{i,j}}(\mathsf{count}) - F_{\kappa_{j,i}}(\mathsf{count})$$

and increments count.

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

$$r_B \leftarrow F_{\kappa_B}(\mathsf{count}_B)$$

and increments count_B.

Figure 10. Protocol Π_{Rand}

Theorem 4. *Assuming F is a pseudo-random function, the protocol Π_{Rand} securely realises $\mathcal{F}_{\mathsf{Rand}}$ in the $\mathcal{F}_{\mathsf{Commit}}$-hybrid model.*

Proof. The **Set-up** procedure is clearly secure assuming an secure commitment functionality. As there is no interaction after **Set-up**, the protocol is clearly 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). □

Note, we do not require that the above protocol is actively secure as we only use it in our passively secure multiplication protocols.

B Proof of Theorem 1

Proof. We prove security in the universal composability (UC) framework. In the UC model, all the possible operations of the world outside the single execution of the protocol are modelled 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.

Given a real-world adversary $\mathcal{A}$, the simulator emulates a set of honest parties interacting with $\mathcal{A}$ as it would during a real-world exection of the protocol. During this interaction, the simulator must additionally extract the corrupt parties' inputs and forward these inputs, along with any errors induced, to the ideal-world functionality. The functionality receives inputs (chosen by the environment) from the honest parties and returns outputs (of different forms, depending on what routines the functionality has executed) to the honest parties and the simulator. When the simulator receives information from the functionality, it must be able to generate a view for $\mathcal{A}$ that is indistinguishable from the view the adversary would have had if it had instead been interacting with the actual honest parties (not via the simulator). 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 11 and Figure 12. Based on this simulation, we now argue that the view of the adversary $\mathcal{A}$ is indistinguishable between worlds.

The method **Add** requires no simulation, since there is no communication involved. Furthermore, the uniformly-randomly-sampled shares generated in the simulation of **Input** and **Multiply** are computationally indistinguishable from the shares generated in a real-world execution of the protocol: In both of these parts of the protocol, the secrets (x and z respectively, in the simulation) are are masked by PRZSs from $\mathcal{F}_{\mathsf{Rand}}$; since the adversary is missing at least one share of each of these secrets, all shares are computationally indistinguishable from uniformly random to the adversary.

Thus the only difficulty in creating the view 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 the simulator 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:

1. The output is a linear function of previously seen sharings; in this case, the simulator can compute all of the shares (by computing the linear function) and hence return the valid sharing.
2. The output is not a linear function of previously seen sharings; in this 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, 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. □

C Proof of Theorems 2 and 3

Proof (Of Theorem 2). 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

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.

Input: When party i is to provide input,

- The simulator receives the command $\mathcal{F}_{\mathsf{Rand}}(\mathsf{PRZS}(\mathsf{count}))$ 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 $\mathcal{F}_{\mathsf{PMPC}}$ as input for this party.

Figure 11. Simulator $\mathcal{S}_{\mathsf{PMPC}}$: Part 1/2

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 13, Figure 14, Figure 15 and Figure 16.

The functionality is designed so that the adversary can choose its shares, and subsequently (not necessarily consequently) 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. Thus, 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 succeeds in creating a view for the adversary which is indistinguishable between worlds.

During **Triple Generation** (Figure 13), 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 $\mathcal{F}_{\mathsf{Rand}}$.

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 $\mathcal{F}_{\mathsf{PMPC}}$.

Multiply: To multiply a secret,
- The simulator receives the command $\mathcal{F}_{\mathsf{Rand}}(\mathsf{PRZS}(\mathsf{count}))$ 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 $\mathcal{F}_{\mathsf{PMPC}}$.

Output: On receiving the command to output shares, with input i,
- The simulator sends the same command to the functionality $\mathcal{F}_{\mathsf{PMPC}}$ 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 12. Simulator $\mathcal{S}_{\mathsf{PMPC}}$: Part 2/2

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 $i^*\in I$,

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

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 $\mathcal{F}_{\mathsf{Rand}}$.**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 $\mathcal{F}_{\mathsf{Rand}}$.**PRSS** honestly and returns the appropriate shares to the adversary.
- When the adversary makes a call to $\mathcal{F}_{\mathsf{Rand}}$.**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 eventually abort; however, the protocol will continue (although it too will eventually abort), so instead, 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.
- If $c_{B,j_1}^{(i)} = c_{B,j_2}^{(i)}$ for all j_1, j_2, then for each $B \in \bigcup_{k \in \mathcal{A}} \mathcal{B}_k$, the simulator chooses any $j \notin \mathcal{A}$ and sets $c_B \leftarrow c_{B,j}$, and then sends the set S to the functionality.

– The simulator then executes $\mathcal{S}_{\mathsf{Triple}}$.**Triple Sacrifice** with the adversary.

Figure 13. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 1/4 (Triple Generation)

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 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 14 and Figure 15), the simulator initialises the hashes as in Π_{Open}.Init and then runs $\mathcal{F}_{\mathsf{Rand}}$.**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 Π_{Open}.**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

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 $\mathcal{F}_{\mathsf{Rand}}$.**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 sets the flag Abort.
  - 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 sets the flag Abort.
  - 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 14. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 2/4 (Triple Sacrifice Part I)

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 collision 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 Π_{Rand}.**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

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 $\Pi_{\mathsf{Rand}}.\mathbf{PRSS}$.)
- 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 $\mathcal{S}_{\mathsf{Triple}}.\mathbf{Opening\ Check}$ with the adversary.

Figure 15. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 3/4 (Triple Sacrifice Part II)

$z^{(i)}$ becomes

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

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 $z^{(i)}_B$ 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 $c^{(i)}_{B,1}$ 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 $r \cdot (c^{(i)}_{B,1} - c^{(i)}_{B,2}) + (c^{(i)}_{B,1} - c^{(i)}_{B,2})$.

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

$$
\begin{aligned}
r^{(i)} \cdot \Delta_{c^{(i)}} - \Delta_{c^{(i+n_T)}} - \sum_{B:B\cap\mathcal{A}\neq\varnothing} z_B^{(i)} \\
&= r^{(i)} \cdot \sum_{k\in\mathcal{A}} \left(\sum_{B\in\mathcal{B}_k} c_B^{(i)} - \left(t_k^{(i)} + \sum_{\rho(B_1,B_2)=k} a_{B_1}^{(i)} \cdot b_{B_2}^{(i)} \right) \right) \\
&\quad - \sum_{k\in\mathcal{A}} \left(\sum_{B\in\mathcal{B}_k} c_B^{(i+n_T)} - \left(t_k^{(i+n_T)} + \sum_{\rho(B_1,B_2)=k} a_{B_1}^{(i+n_T)} \cdot b_{B_2}^{(i+n_T)} \right) \right) \\
&\quad - \sum_{B:B\cap\mathcal{A}\neq\varnothing} r^{(i)} \cdot c_B^{(i)} - \sigma^{(i)} \cdot a_B^{(i+n_T)} - \rho^{(i)} \cdot b_B^{(i+n_T)} - c_B^{(i+n_T)} \\
&\quad - \sigma^{(i)} \cdot \rho^{(i)} \\
&= -\Big(\sum_{B\in\mathcal{B}_j : B\cap\mathcal{A}\neq\varnothing} r^{(i)} \cdot c_B^{(i)} - c_B^{(i+n_T)} \Big) + k,
\end{aligned}
$$

(where k is a constant independent of $c_B^{(i)}$ and $c_B^{(i+n_T)}$) is independent of any shares sent by the adversary. To see this, note that in the middle equation, all of the terms involving c's in the first two lines are subtracted in the third line, leaving only terms that the simulator has generated.) This means that whatever is sampled for the remaining shares of $z^{(i)}$ is consistent with the choice made before.

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. The functionality returns the honest parties shares for the values $a^{(i)}$, $b^{(i)}$, and $c^{(i)}$. The simulator passes these on, by sending $(\{(a_{B,j}^{(i)}, b_{B,j}^{(i)}, c_{B,j}^{(i)} : B \in \mathcal{B} \text{ s.t. } j \in B\})_{j\in\mathcal{P}}$ to the adversary. (I.e. the adversary obtains all shares of all parties' triples.)
- Otherwise, the flag Abort has not been set to true, so the simulator signals Deliver to the functionality.

Figure 16. Simulator $\mathcal{S}_{\mathsf{Triple}}$: Part 4/4 (Opening Check)

In **Opening Checking** (Figure 16), 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. □

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 $\mathcal{F}_{\mathsf{Rand}}$.**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 17. Online Simulator $\mathcal{S}_{\mathsf{Online}}$: Part 1/2

Proof (Of Theorem 3). 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 17 and Figure 18.

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 guaranteed that the values the adversary sends to each

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

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 we proved was UC-secure.

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 $\Pi_{\mathsf{Open}}.\mathbf{Reveal}$ 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 $\Pi_{\mathsf{Open}}.\mathbf{Broadcast}$), 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. □