

# Asynchronous Perfectly Secure Computation Tolerating Generalized Adversaries

M.V.N. Ashwin Kumar, K. Srinathan\*, and C. Pandu Rangan

Department of Computer Science and Engineering  
Indian Institute of Technology, Madras  
Chennai - 600036, India  
{mvnak,ksrinath}@cs.iitm.ernet.in, rangan@iitm.ernet.in

**Abstract.** We initiate the study of perfectly secure multiparty computation over asynchronous networks tolerating generalized adversaries. The classical results in information-theoretically secure asynchronous multiparty computation among  $n$  players state that less than  $\frac{n}{4}$  active adversaries can be tolerated in the perfect setting [4]. Strictly generalizing these results to the non-threshold setting, we show that perfectly secure asynchronous multiparty computation among  $n$  players tolerating the adversary structure  $\mathcal{A}$  is possible if and only if the union of no *four* sets in the adversary structure cover the full set of players. The computation and communication complexities of the presented protocols are polynomial in the size of the maximal basis of the adversary structure. Our results generalize the results of [16,10] to the asynchronous setting. Furthermore, when restricted to the threshold setting, the protocols of this paper result in solutions as good as the best known asynchronous threshold protocols for the perfect setting. Incidentally, the problems of designing efficient asynchronous secure protocols and adapting the efficiency improvement techniques of the threshold setting to the non-threshold setting were mentioned as open in [18,17].

## 1 Introduction

Consider the scenario where there are  $n$  players (or processors)  $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ , each  $P_i$  with a local input  $x_i$ . These players do not trust each other. Nevertheless, they wish to correctly compute a function  $f(x_1, \dots, x_n)$  of their local inputs, whilst keeping their local data as private as possible. This is the well-known problem of *secure multiparty computation*. This problem takes many different forms depending on the underlying network, the function to be computed, and on the amount of distrust the players have in each other and the network. The problem of secure multiparty computation has been extensively studied in several models of computation. The *communication* facilities assumed in the underlying network differ with respect to whether secure communication channels are available [5,9] or not available [14], whether or not broadcast channels are available [23,1,11] and whether the communication channels are

---

\* Financial support from Infosys Technologies Limited, India, is acknowledged.

synchronous or asynchronous [4,6]. The *correctness* requirements of the protocol differ with respect to whether exponentially small probability of error is allowed (*unconditional*) or not allowed (*perfect*). The corrupted players are usually modeled via a central *adversary* that can corrupt up to  $t$  players. The adversary may be computationally bounded (*computational setting*) or unbounded (*secure channels setting*). One also generally distinguishes between actively corrupted players (*Byzantine*), passively corrupted players (*eavesdropping*).

We consider the problem of perfect asynchronous secure multiparty computation over a fully connected network of  $n$  players (processors) in which a non-trivial subset of the players may be corrupted by a Byzantine adversary, where every two players are connected via a secure and reliable communication channel (secure channels setting). The network is *asynchronous*, meaning that any message sent on these channels can have arbitrary (finite) delay.

To model the faulty players' behavior, we postulate a computationally unbounded generalized Byzantine adversary characterized by an adversary structure. An adversary structure is a monotone set of subsets of the player set. The adversary may choose to corrupt the players in any *one* of the sets in the adversary structure. Once a player is corrupted by the adversary, the player hands over all his private data to the adversary, and gives the adversary the control on all his subsequent moves. To model the network's asynchrony, we further assume that the adversary can schedule the communication channel, i.e. he can determine the time delays of all the messages (however, he can neither read nor change those messages). Note that the threshold adversary is a special case of our setting where the adversary structure consists of all the subsets of  $t$  or less players.

It was shown in [4] that perfect asynchronous secure multiparty computation is possible in the threshold setting if and only if  $t < \frac{n}{4}$ . In [6] an  $(\lceil \frac{n}{3} \rceil - 1)$ -resilient protocol is described that securely computes any function  $f$  when exponentially small probability of error is allowed.

Our investigations are motivated by the following observations.

1. The complicated exchanges of messages and zero-knowledge proofs in protocols like [4,6] might render them impractical.
2. The threshold adversarial model is insufficient to model all types of mutual (dis)trust [15,16].

In the synchronous secure multiparty computation setting there have been many attempts to reduce the communication/round complexity of multiparty protocols [3,2,12,13,18,10]. More recently, the results in [10] and [17] significantly improve the message complexity of non-threshold and threshold secure synchronous multiparty computation respectively. The problem of reducing the communication complexity of secure multiparty computation over an asynchronous network was left open in [18,17]. In this work, we initiate an investigation of perfectly secure multiparty computation over asynchronous networks tolerating generalized adversaries. We generalize the results of [4] to the generalized adversary model. We prove the necessary and sufficient conditions for the existence of asynchronous

secure protocols tolerating generalized adversaries. Furthermore, whenever such protocols exist, we design fast protocols for the same. Our solutions heavily draw on the techniques of [8,4,10,18].

## 2 Definitions and Model

We consider  $n$  players  $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$  connected by a complete asynchronous network. We assume that the players in  $\mathcal{P}$  have (polynomially) bounded computational power<sup>1</sup> and can communicate with their neighbours. We assume that randomization is achieved through random coins. A computationally unbounded *Byzantine adversary*  $\mathcal{B}$  is a probabilistic strategy that controls/corrupts a subset of players and endeavors to violate the security of the system. We assume that the adversary can corrupt only the players and not the links connecting them.<sup>2</sup>

The adversary  $\mathcal{B}$  is characterized by a generalized adversary structure  $\mathcal{A} \subseteq 2^{\mathcal{P}}$ , a monotone set of subsets of the player set, where the adversary  $\mathcal{B}$  may corrupt the players of any one set in the structure. To characterize necessary and sufficient conditions for secure multiparty computation, we define the  $\mathcal{Q}^{(k)}$  predicate.

**Definition 1.** *Let  $\mathcal{Y}$  be a finite non-empty set and  $k$  be an integer,  $k > 0$ . A structure  $\mathcal{Z} \subseteq 2^{\mathcal{Y}}$  satisfies the predicate  $\mathcal{Q}^{(k)}$  if no  $k$  sets in  $\mathcal{Z}$  cover the full set  $\mathcal{Y}$ .  $\mathcal{Q}^{(k)} \iff \forall Z_{i_1}, \dots, Z_{i_k} \in \mathcal{Z} : \bigcup_{j=1}^k Z_{i_j} \neq \mathcal{Y}$*

Since the underlying network is asynchronous, we “overload” the adversary with the power to schedule all the sent messages (over all channels) as it wishes and hence the order in which the communicated messages are received is totally under the adversarial control.

Defining security for secure multiparty computation is subtle and has been a field of research on its own [21,1,22,7]. In essence, any protocol  $\Pi$  for secure multiparty computation is defined to be *secure* if  $\Pi$  “emulates” what is called an “ideal” protocol  $\mathcal{I}$ . In  $\mathcal{I}$ , all the players hand their inputs to an *incorruptible* “trusted” player, who locally computes the desired output and hands it back to the players (there is no communication among the players). Therefore, in  $\mathcal{I}$ , the adversary, essentially, learns/modifies the inputs and outputs of only the corrupted players. Now,  $\Pi$  is said to “emulate”  $\mathcal{I}$  if *for all* adversaries attacking  $\Pi$  in the given setting, *there exists* a comparable (in complexity) adversary attacking  $\mathcal{I}$  that induces an *identical* “output” in  $\mathcal{I}$ , where the “output” is the concatenation of the local outputs of all the honest players and the VIEW of adversary.

<sup>1</sup> The players can be modeled as Probabilistic Polynomial Time Turing Machines (PPT).

<sup>2</sup> Our techniques can be easily adapted to provide a solution even when the adversary corrupts only the links or both links and channels. Such an adversary on network  $\mathcal{N}$  can be simulated by an adversary corrupting nodes alone on a new network  $\mathcal{N}'$  got by replacing each insecure link  $e = (u, v)$  by a node  $w$  and two links  $e_1 = (u, w)$  and  $e_2 = (w, v)$ .

In the case of the *asynchronous* setting, unlike the *synchronous* counterpart where there is no “functional” approximation, the local output of the honest players is only an approximation of the pre-specified function  $f$  over a subset  $S$  of the local inputs, the rest taken to be 0, where  $S \supseteq (\mathcal{P} \setminus D) \mid D \in \mathcal{A}$ , for the given generalized adversary structure  $\mathcal{A}$  (this is analogous to the definitions of [4] regarding the threshold model). Furthermore,  $\Pi$  is *perfectly* secure if the local outputs of the honest players are *correct*,  $\Pi$  *terminates* with certainty and the “output” of  $\Pi$  is identically distributed with respect to the “output” of ideal model (which involves a trusted party that approximates  $f$ ).

Tackling generalized adversaries requires new paradigms and tools. In response, we introduce the concept of GMSPs (see Definition 3).

**Definition 2 (MSP).** A *Monotone Span Program* [20] is defined as the triple  $(\mathcal{F}, M, \mathfrak{S})$  where  $\mathcal{F}$  represents a finite field,  $M$  is a  $d \times e$  matrix with entries in  $\mathcal{F}$ , and  $\mathfrak{S} : \{1 \dots d\} \rightarrow \{P_1 \dots P_n\}$  is a function. Each row of the matrix  $M$  is labeled by players in the sense that  $\mathfrak{S}$  assigns the label  $\mathfrak{S}(k)$  to the  $k$ -th row of  $M$ ,  $1 \leq k \leq d$ . For  $A \subset \{P_1 \dots P_n\}$ ,  $M_A$  denotes the matrix that consists of all rows in  $M$  labeled by players in  $A$ . Let  $\mathbf{T} \in \mathcal{F}^e$  be the target vector. A MSP is said to accept (or reject) a structure  $\mathcal{Z}$  if  $\forall Z \in \mathcal{Z}$ , there exists (does not exist, respectively) a linear combination of the rows of  $M_Z$  which equals  $\mathbf{T}$ . A MSP is said to correspond to an adversary structure  $\mathcal{A}_{adv}$  if it rejects exactly  $\mathcal{A}_{adv}$  and accepts exactly  $2^{\{P_1, \dots, P_n\}} \setminus \mathcal{A}_{adv}$ . By the size of an MSP, we mean the number of rows in  $M$ .

Let  $\mathcal{A} = \{D_1, D_2, \dots, D_{|\mathcal{A}|}\}$ , be a generalized adversary structure satisfying the predicate  $\mathcal{Q}^{(k)}$ , over the player set  $\mathcal{P}$ . Trivially, the *corresponding access structure* can be defined as  $2^{\mathcal{P}} \setminus \mathcal{A}$ . However, the size of the smallest MSP corresponding to  $\mathcal{A}$  may not be the best since there may exist a structure  $W \supset \mathcal{A}$  such that the MSP corresponding to  $W$  is *smaller* (see [10]). Therefore, we define the corresponding access structure  $\mathcal{A}_{acc}$  so that the *best* corresponding MSP is not ruled out by the definition itself!

Let  $\mathcal{I}$  denote the set of all combinations of  $(k-1)$  classes from  $\mathcal{A}$ .<sup>3</sup> Define  $\mathcal{A}_{acc} = \{\mathcal{P} \setminus \alpha, \forall \alpha \in \mathcal{I}\}$ . Finally, define the set of “don’t care” sets<sup>4</sup>  $\mathcal{A}_x = 2^{\mathcal{P}} \setminus (\mathcal{A}_{acc} \cup \mathcal{A})$ .

**Definition 3 (Generalized MSP).** A *GMSP* is an MSP (see Definition 2) which corresponds to the generalized adversary structure  $\mathcal{A}$  if it rejects exactly  $\mathcal{A}$ , accepts exactly its corresponding access structure  $\mathcal{A}_{acc}$  and either accepts or rejects the corresponding “don’t care” structure  $\mathcal{A}_x$ .

<sup>3</sup> Note that for all  $\alpha \in \mathcal{I}$ ,  $\alpha \subset \mathcal{A}$ ,  $|\alpha| = (k-1)$  and that  $|\mathcal{I}| = \binom{|\mathcal{A}|}{k-1}$ .

<sup>4</sup> These are the sets that can be freely appended to the adversary structure to reduce the corresponding MSP size to a minimum.

### 3 Characterization of Tolerable Adversaries

**Theorem 1.** *Asynchronous perfectly secure multiparty computation tolerating  $\mathcal{A}$  is possible if and only if  $\mathcal{A}$  satisfies  $\mathcal{Q}^{(4)}$ , i.e. if no four sets in the adversary structure cover the full player set.*

PROOF:

Necessary: For the sake of contradiction assume that asynchronous perfectly secure multiparty computation tolerating  $\mathcal{A}$  is possible even when  $\mathcal{A}$  does not satisfy  $\mathcal{Q}^{(4)}$ . This implies the existence of four sets  $D_1, D_2, D_3$  and  $D_4$  such that<sup>5</sup>  $D_1 \cup D_2 \cup D_3 \cup D_4 = \mathcal{P}$ . Consider four players  $p_1, p_2, p_3$  and  $p_4$  where  $p_1$  plays for all the players in  $D_1$ ,  $p_2$  plays for those in  $D_2$ ,  $p_3$  plays for those in  $D_3$ , and  $p_4$  for those in  $D_4$ . By our hypothesis, we can construct a protocol that tolerates an adversary that corrupts one of  $p_1, p_2, p_3$  or  $p_4$ . However, we know that no such protocol exists [4].

Sufficiency: See Section 4 for a protocol for perfectly secure asynchronous multiparty computation tolerating  $\mathcal{A}$ , which satisfies  $\mathcal{Q}^{(4)}$ .  $\square$

### 4 Asynchronous Perfectly Secure Multiparty Computation

Each player  $P_i$  holds a private input  $x_i$ , and the players wish to securely compute the exact value of a function  $f(x_1, \dots, x_n)$ . However, since the network is asynchronous, and the players in some set  $D \in \mathcal{A}$  may be faulty, the players can never ever wait for more than  $|\mathcal{P} \setminus D|$  of the inputs to be entered to the computation. Furthermore, the missing inputs (treated as 0) are not necessarily of the faulty players (as already mentioned earlier).

Efficient protocols for non-threshold perfect asynchronous secure computation can be constructed based on sub-protocols for *Agreement on a Common Subset*, *Input Sharing*, *Multiplication*, *Segment Fault Localization* and *Output Reconstruction*. We deal with the primitives, viz. *Non-Threshold Broadcast* and *Non-Threshold Asynchronous Byzantine Agreement* separately (see [19]).

In a nutshell, an agreed function is computed as follows: Let  $x_i$  be the input of  $P_i$ . Let  $\mathcal{F}$  be a finite field known to all players, and let  $f : \mathcal{F}^n \rightarrow \mathcal{F}$  be the computed function. We assume that the players have an arithmetic circuit computing  $f$ ; the circuit consists of addition gates and multiplication gates of in-degree 2. All the computations in the sequel are done in  $\mathcal{F}$ .

First each player “shares” his input among the  $n$  players using the sub-protocol for *InputShare* (see Section 4.3). This sub-protocol runs in two phases. In the first phase it uses a technique similar to the commitment technique of [10], modified to the asynchronous setting (*NAVSS-SHARE*, see Section 4.2). In the second phase, the players agree, using the protocol for *Agreement on a Common Subset*, on a core set  $\mathcal{G}$  of players that have successfully shared their input. Once

<sup>5</sup> From the monotonicity of  $\mathcal{A}$ , the  $D_i$ 's can be made pair-wise disjoint.

$\mathcal{G}$  is computed, the players proceed to compute the function in the following way. First, the input values of the players not in  $\mathcal{G}$  are set to a default value, say 0. Note that in the process of the above sharing, in order to tolerate the adversary structure  $\mathcal{A}$  it may be necessary for some of the players to “act” for more than one similar players, i.e., they may receive more than one share etc; hereafter, we use the notation  $\mathcal{P}_{phy}$  to mean the physical set of players and  $\mathcal{P}_{log}$  to denote the logical set of players (wherein two players simulated by the same physical player are considered different). Next, the players evaluate the given circuit gate by gate as follows. If the gate is an *addition* gate, then simply adding the secret-shares of the input lines would suffice (due to the linearity of the NAVSS-SHARE scheme). Multiplication gates are evaluated using the sub-protocol *MUL* (see Section 4.4) which applies the player-elimination technique [18] (by dividing the circuit into segments); either the outcome is a proper sharing of the correct product, or a fault is detected. In the latter case, the sub-protocol for *Segment Fault Localization* (see Section 4.5) is applied (at the end of that segment) to determine a localization  $\mathcal{D}$ . Then the (logical) players in  $\mathcal{D}$  are eliminated from the further protocol, and the shared values are re-shared for this new setting involving fewer (logical) players using the sub-protocol for *Re-SHARING* (see Section 4.4). Finally, after all gates have been evaluated, the output value is reconstructed from the shares of the output line toward each of the (physical) players using the sub-protocol NAVSS-REC. The top-level protocol is described in Section 5.

#### 4.1 Agreement on a Common Subset

In a perfect asynchronous resilient computation, very often the players in  $\mathcal{P}_{phy}$  need to decide on a subset  $\mathcal{G}$  of players, that satisfy some property, such that  $\mathcal{G} \supseteq (\mathcal{P}_{phy} \setminus D)$ , for some  $D \in \mathcal{A}$ . It is known that all the honest players will eventually satisfy this property, but some faulty players may satisfy it as well. In our context, we need to agree on the set  $\mathcal{G}$  of players who have completed correctly sharing their input. For implementing this primitive, we adapt the protocol presented in [6] to the non-threshold setting. The idea is to execute a *BA* (Byzantine Agreement) protocol for each player, to determine whether it will be in the agreed set. Notice that if some  $P_j$  knows that  $P_i$  satisfies the required property, then eventually all the players will know the same. Thus, we can suppose the existence of a predicate  $\mathcal{J}$  that assigns a binary value to each player  $P_i$ , denoted  $\mathcal{J}(i)$ , based on whether  $P_i$  has satisfied the property as yet. The protocol is denoted by  $AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]$  and given in Fig. 1.

**Theorem 2.** *Using the protocol  $AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]$  the players indeed agree on a common subset of players, denoted by  $\mathcal{G}$  such that  $\mathcal{G} \supseteq (\mathcal{P}_{phy} \setminus D)$ , for some  $D \in \mathcal{A}$ . Moreover, for every  $P_j$  in  $\mathcal{G}$ , we have  $\mathcal{J}(j) = 1$ .*

*Proof.* Similar in lines with [6]. We omit it due to space constraints.  $\square$

## 4.2 NAVSS Protocol

**Definition 4 (NAVSS).** Let  $(\text{SHARE}, \text{REC})$  be a pair of protocols in which a dealer  $P_D$ , shares a secret  $s$ . We say that  $(\text{SHARE}, \text{REC})$  is a NAVSS scheme tolerating the adversary structure  $\mathcal{A}$  if the following hold for every adversary  $\mathcal{B}$  characterized by  $\mathcal{A}$ , and every input.

- Termination: With certainty, the following conditions hold:
  1. If the dealer is honest, then each honest player will eventually will complete protocol SHARE.
  2. If some honest player has completed protocol SHARE, then each honest player will eventually complete protocol SHARE.
  3. If an honest player has completed protocol SHARE, and all the honest players invoke protocol REC, then each honest player will complete protocol REC.
- Correctness: Once an honest player has completed protocol SHARE, then there exists a unique value,  $r$ , such that certainly the following holds:
  1. If the dealer is honest, then  $r$  is the shared secret, i.e.,  $r = s$ .
  2. If all the honest players invoke protocol REC, then each honest player outputs  $r$ . (Namely,  $r$  is the reconstructed secret).
- Secrecy:<sup>6</sup> If the dealer is honest and no honest player has begun executing protocol REC, then the faulty players have no information about the shared secret.

Our construction of the NAVSS protocol is given in Fig. 2.

**Theorem 3.** The pair NAVSS-(SHARE, REC) is a NAVSS scheme tolerating the adversary structure  $\mathcal{A}$ , provided  $\mathcal{A}$  satisfies  $\mathcal{Q}^{(4)}$ .

*Proof.* We assert the Termination, Correctness and Secrecy requirements of the above scheme. As per the definition, if the dealer is honest, an NAVSS scheme has to terminate with certainty for all uncorrupted players. In case of a corrupted dealer no requirements are posed on the termination. The protocol NAVSS-SHARE terminates with probability 1 for an honest dealer (this follows quite easily from the proof of the V-Share protocol given in [4]). The protocol NAVSS-REC terminates with certainty since  $P$  will eventually receive messages from at least the players  $\mathcal{P}_{log} \setminus D_{log}$ , for some  $D_{log} \in \mathcal{A}_{log}$ . The correctness of the NAVSS-SHARE protocol follows from the results of [4,10]. The correctness of the protocol NAVSS-REC can be proven as follows: Assume that a player hands a bad vector  $\mathbf{u}_i' \neq \mathbf{u}_i$ . Of the  $\mathcal{P}_{log} \setminus D_{log}$  messages received, this vector is inconsistent with that of at least  $(\mathcal{P} \setminus (D_1 \cup D_2))$  logical players. At least  $(\mathcal{P} \setminus (D_1 \cup D_2))$  logical players gave their correct vectors to  $P$ . Player  $P$  will detect inconsistencies for every set in the access structure and ignore this vector. On the other hand, if  $\mathbf{u}_i$  is the correct vector, at most the vectors of  $D \in \mathcal{A}_{log}$  for some  $D$ , will be inconsistent. Hence  $P$  interpolates only correct vectors and computes the correct

<sup>6</sup> An honest player is not required to complete protocol SHARE in case the dealer is faulty.

### AgreeSet $[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]$

1. For each  $P_j$  for whom  $P_i$  knows that  $\mathcal{J}(j) = 1$ , participate in  $BA_j$  (Byzantine Agreement) with input 1.
2. Upon completing  $BA_j$  protocols for all  $P_j \in (\mathcal{P}_{phy} \setminus D)$ , for some  $D \in \mathcal{A}$  with output 1, enter input 0 to all  $BA$  protocols for which  $P_i$  has not entered a value yet.
3. Upon completing all  $n$   $BA$  protocols, let the  $AgreeSet_i$  be the set of all indices  $j$  for which  $BA_j$  had output 1. Output  $AgreeSet_i$ .

**Fig. 1.** Agreement on a Common Subset

### SHARE Protocol

**Publicly known Inputs:** Player set  $\mathcal{P}_{log}$ , adversary structure  $\mathcal{A}$ , (corresponding) access structure  $\mathcal{A}_{acc}$  and the GMSP  $\mathcal{M} = (\mathcal{F}, M_{d \times e}, \mathfrak{S})$  that corresponds to  $\mathcal{A}$ .  
Code for the Dealer  $P_D$  (on input  $s$ ):

1. Choose a symmetric  $e \times e$  matrix  $R$  at random, with  $R[0][0] = s$ . Let  $\mathbf{v}_i$  be a row in  $M$  assigned to  $P_i$  and let  $\mathbf{v}_i^T$  be its transpose (column vector). Then  $P_D$  sends to  $P_i$  the vector  $\mathbf{u}_i = R \cdot \mathbf{v}_i^T$ . A share  $s_i$  of  $s$  given to  $P_i$  is defined to be the first entry of  $\mathbf{u}_i$ . Hence, the product  $\langle \mathbf{v}_j, \mathbf{u}_i \rangle = s_{ij}$  can be thought of as a share of  $s_i$  given to  $P_j$ . Note that we have  $\langle \mathbf{v}_j, \mathbf{u}_i \rangle = \langle \mathbf{v}_j \cdot R, \mathbf{v}_i^T \rangle = \langle \mathbf{v}_i, R \cdot \mathbf{v}_j^T \rangle = \langle \mathbf{v}_i, \mathbf{u}_j \rangle$ .

Code for player  $P_i$ :

1. Upon receiving  $\mathbf{u}_i$  from  $P_D$ , send to each logical player  $P_j$ ,  $\langle \mathbf{v}_j, \mathbf{u}_i \rangle$ .
2. Upon receiving  $x_{ji}$  from  $P_j$ , if  $s_{ij} = x_{ji}$ , then Broadcast  $(OK, i, j)$ .
3. Upon receiving a broadcast  $(OK, j, k)$ , check for the existence of a  $\mathcal{A}$ -clique in  $OK_i$  graph (The *undirected*  $OK_i$  graph =  $(\mathcal{P}_{log}, \mathcal{E})$  where edge  $(P_j, P_k) \in \mathcal{E}$  if  $P_i$  has received broadcasts  $(OK, j, k)$  &  $(OK, k, j)$ ). If an  $\mathcal{A}$ -clique is found (all players in  $D_i = (\mathcal{P}_{log} \setminus D)$ ,  $D \in \mathcal{A}$  form a clique in the  $OK_i$  graph), go to Step 5 and send  $D_i$  to all physical players.
4. Upon receiving  $D_j$  add  $D_j$  to the set of 'SUGGESTED CLIQUES'. As long as a clique is not yet found, then whenever an  $(OK, k, \ell)$  broadcast is received, check whether  $D_j$  forms an  $\mathcal{A}$ -clique in the  $OK_i$  graph.
5. Upon finding an  $\mathcal{A}$ -clique, and if  $i \notin D_i$ , correct  $\mathbf{u}_i$  as follows: For each  $D \in \mathcal{A}$ ,  $D \subseteq D_i$ , check if there exists a linear combination of the rows in  $M_{D_i \setminus D}$  resulting in the target vector (i.e.,  $(D_i \setminus D) \supseteq A$  for some  $A \in \mathcal{A}_{acc}$ ). Use that  $D^* = D_i \setminus D$  to construct  $\mathbf{u}_i$ . (This is surely possible since the number of linearly independent rows in  $M_{D^*}$  is greater than the number of linearly independent columns, i.e.  $|\mathbf{u}_i| = e$ .)
6. Once  $\mathbf{u}_i$  is correct, (locally) output  $\mathbf{u}_i$ .

### REC Protocol (Toward Player $P$ )

Code for player  $P_i$ :

1. Send  $\mathbf{u}_i$  to player  $P$ .

Code for player  $P$ :

1. Wait for receipt of  $\mathbf{u}_i$ 's from all the players in  $D' = (\mathcal{P}_{log} \setminus D)$ .
2. Reconstruct secret as follows: Find  $D^* \subseteq D'$  such that  $\forall P_i, P_j \in D^*$ ,  $\langle \mathbf{v}_j, \mathbf{u}_i \rangle = \langle \mathbf{v}_i, \mathbf{u}_j \rangle$ . Let  $\mathbf{s}$  be the first elements in each of the  $\mathbf{u}_i$ 's in  $D^*$ . Now find  $\lambda_{D^*}$  such that  $\lambda_{D^*} \times M_{D^*} = \mathbf{T}$ . The required secret is  $\lambda_{D^*} \times \mathbf{s}^T$ .

**Fig. 2.** Error-free NAVSS protocol.

secret  $s$ . The secrecy of protocol NAVSS-SHARE follows from the definitions of an MSP. The privacy of the NAVSS-REC protocol is obvious as no player but  $P$  receives any information.  $\square$

### 4.3 Input Sharing

The protocol for Input Sharing has two phases: first, each party shares its input using NAVSS-SHARE) scheme; next, the parties use the AgreeSet protocol to agree on a set  $\mathcal{G}$  of at least  $|\mathcal{P}_{phy} \setminus D|$ , for some  $D \in \mathcal{A}$  players who have shared their inputs properly. The protocol *InputShare* is formally specified in Fig. 3.

### 4.4 Multiplication

A protocol for multiplication starts with sharing of  $x, y$  and ends with a sharing of  $z = x \cdot y$  (if there are no inconsistencies) or with a partial fault detection (if there exist inconsistencies). The adversary's view of the computation is distributed independent of the initial sharing for any adversary characterized by  $\mathcal{A}$ . Our implementation of the multiplication protocol is given in Fig. 3.

**Re-SHARING Protocol.** We begin by defining a multiplicative MSP (MMSP).

**Definition 5 (MMSP[10]).** *A multiplicative MSP is an MSP  $\mathcal{M}$  for which there exists an vector  $\mathbf{r}$  called a recombination vector, such that for any two secrets  $s$  and  $s'$ , and any  $\boldsymbol{\rho}, \boldsymbol{\rho}'$ , it holds that*

$$s \cdot s' = \langle \mathbf{r}, M(s, \boldsymbol{\rho}) \diamond M(s', \boldsymbol{\rho}') \rangle$$

where  $\mathbf{x} \diamond \mathbf{y}$  is defined as the vector containing all the entries of the form  $x_i \cdot y_j$ , where  $\mathbf{x} = (x_1, \dots, x_d)$ ,  $\mathbf{y} = (y_1, \dots, y_d)$  and  $\mathfrak{S}(i) = \mathfrak{S}(j)$ . We say that  $\mathcal{M}$  is strongly multiplicative if for any player subset  $A$  that is rejected by  $\mathcal{M}$ ,  $\mathcal{M}_{\bar{A}}$  is multiplicative.

Assume that  $s$  is shared with a strongly MMSP  $\mathcal{M} = (\mathcal{F}, M_{d \times e}, \mathfrak{S})$  and random  $R'_{e' \times e'}$  matrix, with player  $P_i$  holding  $\mathbf{u}_{i_k} = R' \cdot \mathbf{v}_{i_k}^T, \forall i_k \ni \mathfrak{S}(i_k) = P_i, 1 \leq i \leq n$ . The goal of Re-SHARING protocol is to transform this sharing into a proper sharing of  $s$  resulting from the use of the random sharing matrix  $R_{e \times e}$ ,  $e' < 2e$ . A protocol for Re-SHARING starts with a  $R'$ -sharing of  $s$  and ends with either a  $R$ -sharing of  $s$  (if there are no inconsistencies) or with a partial fault detection (if there exist inconsistencies). Also, the adversary view of the protocol is distributed independent of the initial sharing for any adversary characterized by  $\mathcal{A}$ . We implement the Re-SHARING protocol as follows: first, every logical player  $P_i$   $R$ -shares  $s_i = \mathbf{u}_i[0]$  using the protocol *EfficientAVShare*(see Fig. 3) and proves that the value shared is in fact  $s_i$  using the protocol *ACheckShare*(see Fig. 3). For this, since the secret  $s_i$  has already been shared using  $\mathbf{u}_i$  (represented as  $\mathbf{u}'$  in Fig. 3), sharing it again with  $\mathbf{u}$  as the first row of  $R_{e \times e}$ , implies that the player  $P_i$  has to prove  $\mathbf{u}[0] = \mathbf{u}'[0]$ , which is what is accomplished by our protocol *ACheckShare*. Now, every logical player locally computes the linear combination using the known recombination vector of the MMSP which results in a  $R$ -sharing of  $s$  [10]. The formal description of the implementation of the protocol Re-SHARING is given in Fig. 3.

<p><b>Protocol InputShare</b><math>[\mathcal{P}_{phy}, \mathcal{A}]</math></p> <p>Code for party <math>P_i</math> with secret <math>s_i</math></p> <ol style="list-style-type: none"> <li>1. Initiate <math>NAVSS-SHARE_i[\mathcal{P}_{log}, \mathcal{A}, P_i, s_i]</math>. For <math>1 \leq j \leq  \mathcal{P}_{log} </math>, participate in <math>NAVSS-SHARE_j</math>. Let <math>\mathbf{u}_i^{(j)}</math> be the output of <math>NAVSS-SHARE_j</math>.</li> <li>2. Execute protocol <math>AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]</math> with the boolean predicate <math>\mathcal{J}(j) = 1</math> if all the logical players associated with the physical <math>P_j</math> have completed their respective <math>NAVSS-SHARE</math> successfully. Let <math>\mathcal{G}</math> be the output of the protocol.</li> <li>3. Output <math>\mathcal{G}, \mathbf{u}_i^{(j)}, P_j \in \mathcal{G}</math>.</li> </ol>
<p><b>Protocol EfficientAVShare</b><math>[\mathcal{P}_{log}, \mathcal{A}, P, s]</math></p> <p>The dealer <math>P</math> chooses a symmetric random matrix <math>R_{e \times e}</math> such that <math>R[0, 0] = s</math> and sends the vector <math>\mathbf{u}_i = R \cdot \mathbf{v}_i^T</math> to logical player <math>P_i</math>.</p> <ol style="list-style-type: none"> <li>1. Upon receiving <math>\mathbf{u}_i</math>, send <math>\langle \mathbf{v}_j, \mathbf{u}_i \rangle</math> to logical player <math>P_j</math>.</li> <li>2. Upon receiving messages <math>m_{ji}</math> from all logical players in <math>\mathcal{P}_{log} \setminus D</math> for some <math>D \in \mathcal{A}_{log}</math> (we use <math>\mathcal{A}_{log}</math> to denote the adversary structure over logical players) check if <math>m_{ji} = \langle \mathbf{v}_i, \mathbf{u}_j \rangle</math>. If equality holds for all <math>m_{ji}</math>, and for all the logical players that <math>P_i</math> is acting for, then, send a <math>CheckMessage_i := "OK"</math>, else, send <math>CheckMessage_i := j</math>, where <math>j</math> denotes the smallest index such that the value received from <math>P_j</math> was not equal to <math>\langle \mathbf{v}_j, \mathbf{u}_i \rangle</math>, to all logical players.</li> <li>3. Execute protocol <math>AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]</math> with the boolean predicate: <math>\mathcal{J}(j) = 1</math> if all the <math>CheckMessage</math>'s corresponding to physical player <math>P_j</math> has been received. Let <math>\mathcal{G}</math> be the output of this protocol. If <math>CheckMessage_j = "OK" \forall j</math> such that <math>P_j \in \mathcal{G}</math>, then the protocol succeeds with output <math>\mathbf{u}_i</math>. Else, set <math>FaultDetected_i := TRUE</math>.</li> </ol> <p style="text-align: center;"><b>Protocol ACheckShare</b><math>[\mathcal{P}_{log}, e', \mathcal{A}, P, s]</math></p> <ol style="list-style-type: none"> <li>1. The dealer sets <math>(0, \mathbf{g}) := \mathbf{u} - \mathbf{u}'</math> and distributes shares using <math>(\mathbf{g}, 0)</math> as the first row of <math>R_{new}</math>, to every logical player <math>P_j</math> using the protocol <math>EfficientAVShare</math>. If this fails, then the whole verification protocol fails.</li> <li>2. Every logical player <math>P_i</math> checks that <math>(R' \cdot \mathbf{v}_i^T)[0] + \frac{\mathbf{v}_i[k+1]}{\mathbf{v}_i[k]} (R_{new} \cdot \mathbf{v}_i^T)[0] = (R \cdot \mathbf{v}_i^T)[0]</math>. If consistent for all the logical players that <math>P_i</math> is acting for, the physical player <math>P_i</math> sends <math>CheckBit_i := 1</math>, else he sends <math>CheckBit_i := 0</math>, to all (physical) players.</li> <li>3. Every physical player executes protocol <math>AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]</math> with <math>\mathcal{J}(j) = 1</math> if all <math>CheckBit</math>'s related to physical player <math>P_j</math> has been received. Let <math>\mathcal{G}</math> be the output of this protocol. If <math>CheckBit_j = 1</math> for all <math>j \mid P_j \in \mathcal{G}</math>, then the verification was successful. Else, set <math>FaultDetected_i := TRUE</math>.</li> </ol> <p style="text-align: center;"><b>Protocol Re-SHARING</b><math>[\mathcal{P}_{log}, e', \mathcal{A}, s]</math></p> <ol style="list-style-type: none"> <li>1. Initiate <math>EfficientAVShare_i[\mathcal{P}_{log}, \mathcal{A}, P_i, s_i]</math>. Participate in <math>EfficientAVShare_j</math> for all logical players <math>P_j</math>. If <math>FaultDetected_i = FALSE</math> then let <math>\mathbf{u}_j</math> be the output of <math>EfficientAVShare_j</math>. Else terminate with output 'NULL'.</li> <li>2. Run <math>ACheckShare_i[\mathcal{P}_{log}, e', \mathcal{A}, s_i]</math>. Participate in <math>ACheckShare_j</math> for all logical players <math>P_j</math>. If <math>FaultDetected_i = TRUE</math> then terminate with output 'NULL'.</li> <li>3. Execute protocol <math>AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]</math> with the boolean predicate: <math>\mathcal{J}(j) = 1</math> if <math>P_j</math> has completed <math>EfficientAVShare</math>'s for all the logical players that he is acting for, successfully to get the output say <math>\mathcal{G}</math>.</li> <li>4. Locally compute the linear combination of the shares received from players in <math>\mathcal{G}</math>, to result in an <math>R_{e \times e}</math>-sharing, as in [10].</li> </ol> <p style="text-align: center;"><b>MUL</b><math>[\mathcal{P}_{log}, \mathcal{A}, x, y]</math></p> <p>Each logical player <math>P_i</math> does the following:</p> <ul style="list-style-type: none"> <li>- Evaluates <math>\mathbf{z}_i[j] = \mathbf{x}_i[j] \cdot \mathbf{y}_i[j]</math>, for <math>j = 0, \dots, e - 1</math>.</li> <li>- Calls <math>Re - Sharing[\mathcal{P}, 2e - 2, e, z]</math> which starts with a <math>2t</math>-sharing of <math>z</math>.</li> <li>- If the above Re-SHARING call outputs 'NULL' then terminates with output 'FAIL', else ends with the <math>R_{e \times e}</math>-sharing of <math>z</math>.</li> </ul>

**Fig. 3.** Input Sharing with Fault Detection, Re-SHARING and Multiplication

**Theorem 4.** *The protocol Re-SHARING is a  $\mathcal{A}$ -resilient protocol for the re-sharing functionality.*

*Proof. (sketch)* The protocol *EfficientAVShare* terminates because the players of at most one set  $D \in \mathcal{A}_{log}$ , for some set  $D$ , is corrupted and hence at least  $(\mathcal{P}_{log} \setminus D)$  players will distribute consistent vectors and so the set  $\mathcal{G}$  is well defined. On similar lines, we can see that the protocol *ACheckShare* terminates as well. The correctness of the Re-SHARING protocol follows: If the protocol *EfficientAVShare* succeeds, then it implies that there exists a set of players  $\mathcal{G} \supseteq (\mathcal{P} \setminus D)$  such that each player  $P_i \in \mathcal{G}$  is consistent with the vectors of at least players in  $(\mathcal{P} \setminus D)$  of which at least  $(\mathcal{P} \setminus (D_1 \cup D_2))$  are honest. Since  $\mathcal{A}$  satisfies  $\mathcal{Q}^{(4)}$ , the players in  $\mathcal{G}$  have verifiable vectors and the  $(\mathcal{P} \setminus (D_1 \cup D_2))$  honest players in  $\mathcal{G}$  define a unique sharing of  $s$ . Also, if the sharing phase of the protocol *ACheckShare* succeeds, then indeed the vectors of all players are as per  $R'$ -sharing. Hence, if there exists a set of players  $\mathcal{G} \supseteq (\mathcal{P} \setminus D)$  such that each honest player  $P_i \in \mathcal{G}$  is consistent with  $(R' \cdot \mathbf{v}_i^T)[0] + \frac{\mathbf{v}_i^{[k+1]}}{\mathbf{v}_i^{[k]}} (R_{new} \cdot \mathbf{v}_i^T)[0] = (R \cdot \mathbf{v}_i^T)[0]$ , then the vectors  $\mathbf{u}$ ,  $\mathbf{u}'$  and  $(0, \mathbf{g})$  are uniquely defined which implies that  $\mathbf{u}[0] = \mathbf{u}'[0]$  (i.e. same secrets!). The secrecy of this protocol is due to the independence of the the sharings.  $\square$

#### 4.5 Segment Fault Localization

The purpose of fault localization is to find out which players are corrupted or, because agreement about this can usually not be reached, at least to narrow down the set of players containing the cheaters. The output of an  $(r, p)$ -localization is a set  $\mathcal{D}_{log}$  with  $|\mathcal{D}_{log}| = p$  logical players, guaranteed to contain at least  $r$  corrupted logical players.

For our lazy re-sharing procedure, to preserve the ability for continuing the computation, it is required that after a (sequence of)  $(r, p)$ -localizations and logical player eliminations (without Re-SHARING), still  $\mathcal{A}$  satisfies  $\mathcal{Q}^{(4)}$  holds, and each  $R'$  sharing still satisfies  $e' < 2e$ . As will be evident in the sequel, the localizations used in our protocol indeed satisfy these requirements.

In our protocol, if a segment has detected a fault, the first faulty sub-protocol is found and we invoke the corresponding fault localization procedure. In what follows, we outline the fault localization methodology for the various sub-protocols that are allowed to fail with a partial fault detection, viz. *EfficientAVShare*, *ACheckShare* and *Re-SHARING*.

1. **Sharing Protocol (*EfficientAVShare*):** From the corresponding common set  $\mathcal{G}$ , among all the physical players who complained about an inconsistency, each of the uncorrupted players can agree on the physical player  $P_i$  with the smallest index  $i$ . From  $P_i$ 's *CheckMessage<sub>i</sub>* all the players know some  $P_j$  that  $P_i$  complained about. The physical players execute a BA protocol (with their *CheckMessage<sub>i</sub>*, that  $P_i$  sent to them, as input) to agree on a single  $P_j$ . Then every physical player sets  $\mathcal{D}_{phy} := \{P, P_i, P_j\}$ , where  $P$  is the corresponding dealer. It is obvious that all players find the same set  $\mathcal{D}_{phy}$ , and at least one physical player in  $\mathcal{D}_{phy}$  must be corrupted. Define the

weight<sup>7</sup> of a physical player  $P_i$  to be  $w(P_i) = |\{k | \mathfrak{S}(k) = P_i, 1 \leq k \leq d\}|$ . Let  $w_{min} = \min(w(P), w(P_i), w(P_j))$  where  $\mathcal{D}_{phy} := \{P, P_i, P_j\}$ . Define the localization  $\mathcal{D}_{log} = w_{min}$  rows of each of  $P, P_i$  and  $P_j$ . Clearly,  $\mathcal{D}_{log}$  is a  $(w_{min}, 3w_{min})$ -localization.

2. **Verifying Shares Protocol (ACheckShare):** Let  $P_i$  be the physical player with the smallest index in the common set  $\mathcal{G}$  who complained. Then the set  $\mathcal{D}_{phy} := \{P, P_i\}$ , and  $\mathcal{D}_{log}$  is constructed similarly.
3. **Re-SHARING Protocol:** Failure of Re-SHARING protocol is due to the failure of either or both of the above sub-protocols. Then the same  $\mathcal{D}_{log}$  is determined as in the first failed sub-protocol.

## 5 The Top-Level Protocol

Let  $f : \mathcal{F}^n \rightarrow \mathcal{F}$  be given by an arithmetic circuit  $\mathcal{C}$ . Our protocol for securely computing  $f(x_1, x_2, \dots, x_n)$  is described in Fig. 4.

## 6 Complexity Analysis of Our Protocol

In this section, MC stands for message complexity, BC denotes Broadcast complexity, BAC denotes Byzantine Agreement complexity and  $d$  denotes the size of the minimum monotone span program corresponding to the  $\mathcal{A}$ .

*Complexity Analysis of Initial Sharing :* In the NAVSS-SHARE <sub>$i$</sub>  $[\mathcal{P}_{phy}, \mathcal{A}, P]$  protocol the distribution phase communicates  $MC = ne \lg |\mathcal{F}| + d^2 \lg |\mathcal{F}|$  bits. In the verification phase each party needs to send a clique in the graph which requires  $O(d)$  bits. So, this phase has  $MC = O(d^2)$  and  $BC = O(n \lg n)$ .

The *InputShare* $[\mathcal{P}, \mathcal{A}]$  protocol runs NAVSS-SHARE <sub>$i$</sub>   $n$  times followed by an *AgreeSet* protocol. Hence,  $MC = O(nd^2 \lg |\mathcal{F}|)$  bits and  $BC = O(n^2 \lg n)$  bits.

*Complexity Analysis of Receiving Output :* The NAVSS-REC $[\mathcal{P}_{log}, \mathcal{A}, P]$  protocol requires each logical player to send  $e$  field elements to  $P$ . This requires  $MC = O(ed \lg \mathcal{F})$  bits.

*Complexity Analysis of our Efficient VSS:* In the *EfficientAVShare <sub>$i$</sub>*  protocol the distribution phase communicates  $MC = ne \lg |\mathcal{F}| + d^2 \lg |\mathcal{F}|$  bits. The pairwise consistency checks need another  $MC = O(d^2 \lg |\mathcal{F}|)$  bits to be sent. The agreement on  $\mathcal{G}$  has  $BAC = O(n)$  bits. The *ACheckShare* protocol runs an *EfficientAVShare <sub>$i$</sub>*  followed by  $n^2$  bits of communication and an *AgreeSet* protocol. Hence,  $MC = O(d^2 \lg |\mathcal{F}|)$  and  $BAC = O(n)$  bits.

*Complexity Analysis of Re-SHARING :* The Re-SHARING protocol amounts to running each of the above two protocols, namely *EfficientAVShare* and *ACheckShare*  $d$  times followed by an *AgreeSet* protocol. This requires  $MC = O(d^3 \lg |\mathcal{F}|)$  bits and  $BAC = O(dn)$  bits.

*VSS with Fault Localization* uses  $BAC = O(\lg n)$ .

*Analysis of a Segment with  $\ell$  Multiplications:* Every multiplication gate takes up to three Re-SHARING (re-sharing of arguments and the actual multiplication),

<sup>7</sup> Informally, the weight of a player is the number of rows in the MSP assigned to him; i.e. the number of logical players that he acts for.

**Protocol AsyncPerfectSecureCompute** $[n, \mathcal{C}, x_1, \dots, x_n]$

1. **Initialization:** Set  $\mathcal{P}_{phy} := \{P_1, P_2, \dots, P_n\}$  and  $\mathcal{P}_{log} := \{P_1, \dots, P_d\}$  and the adversary structure  $\mathcal{A}$  satisfies  $\mathcal{Q}^{(4)}$ .
2. **Input Sharing:** Set  $(\mathcal{G}, \mathbf{u}_i^{(j)}) := InputShare[\mathcal{P}_{log}, \mathcal{A}]$ ,  $j \in \mathcal{G}_{log}$  (we use  $\mathcal{G}_{log}$  to denote all the logical players enacted by the physical players in  $\mathcal{G}$ ).  
The secret  $s_i$  of a physical player  $P_i$  in the above sub-protocol is  $x_i$ . For a line  $l$  in the circuit, let  $l^{(i)}$  denote the share of logical player  $P_i$  in the value of this line. If  $l$  is the  $j^{th}$  input line of the circuit, then: Set  $l^{(i)} := \mathbf{u}_i^{(j)}[0]$  if  $j \in \mathcal{G}$  and  $l^{(i)} := 0$  otherwise.

3. **Computation:**

For each segment of the circuit :

Repeat

For each physical  $P_i$ : Set  $FaultDetected_i := FALSE$ .

For each gate  $g$  in the segment:

Each logical player  $P_i$  does the following:

Wait until the  $i^{th}$  shares of all input lines of  $g$  are computed.

If  $g$  is an addition gate with output line  $l$  and input lines  $l_1, l_2$ :

Set  $l^{(i)} := l_1^{(i)} + l_2^{(i)}$ .

Logical players  $P_i$  with  $FaultDetected_i = TRUE$  use random shares.

If  $g$  is a multiplication gate with output line  $l$  and input lines  $l_1, l_2$ :

If  $l_k$ ,  $k = 1$  or  $2$  is shared with  $R_{e' \times e'}$ , with  $e' > e$ :

Call  $Re - Sharing[\mathcal{P}_{log}, e', \mathcal{A}, l_k]$

Every physical player  $P_i$  with  $FaultDetected_i = TRUE$  uses random shares in the sub-protocol. If the sub-protocol outputs 'NULL', then  $P_i$  sets  $FaultDetected_i := TRUE$

Set  $l := MUL[\mathcal{P}_{log}, \mathcal{A}, l_1, l_2]$

Every physical player  $P_i$  with  $FaultDetected_i = TRUE$  uses random shares in the sub-protocol. If the sub-protocol outputs 'FAIL', then  $P_i$  sets  $FaultDetected_i := TRUE$

For each physical  $P_i$ , broadcast  $FaultDetected_i$ .

Set  $\mathcal{G} := AgreeSet[\mathcal{J}, \mathcal{P}_{phy}, \mathcal{A}]$ ,  $\mathcal{J}(j) = 1$  for  $P_j$  if it received the broadcast from  $P_j$ .

If at least one physical player in  $\mathcal{G}$  has  $FaultDetected_i = TRUE$ :

Each  $P_i$  broadcasts the index of the first sub-protocol that failed.

Agree on the smallest sub-protocol that failed.

Invoke the fault localization procedure for that sub-protocol to get  $\mathcal{D}_{log}$  from  $\mathcal{D}_{phy}$ .

Set the logical player set  $\mathcal{P}_{log} := \mathcal{P}_{log} \setminus \mathcal{D}_{log}$

and  $\mathcal{A} := \mathcal{A} \setminus \{D \mid D \in \mathcal{A}, \mathcal{D}_{phy} \cap D = \emptyset\}$ .

Until  $((FaultDetected_i = FALSE)$  for all  $P_i \in \mathcal{G}$ ).

For each physical player  $P$  needing output:

Call  $OutputReconstruction[\mathcal{P}_{log}, \mathcal{A}, P]$ .

**Fig. 4.** Protocol for Asynchronous Secure Computation

where the re-sharings can be performed in parallel. In every Re-SHARING only the actual computation with partial fault detection is performed. At the end of the segment, during (strict) fault detection,  $n$  bits are broadcast as well as an *AgreeSet* is run. If there are faults, extended fault localization is performed. There are at most  $O(n\ell)$  partial fault detections, hence to localize the first one which reported some failures  $O(n \lg(n\ell))$  bits are broadcast. The total complexities are  $MC = O(\ell d^3 \lg |\mathcal{F}|)$  and  $BAC = O(\ell nd)$ .

**Cumulative Complexity Analysis:** The protocol *AsyncPerfectSecureCompute*  $[n, \mathcal{C}, x_1, \dots, x_n]$  uses *InputShare* $[\mathcal{P}, \mathcal{A}]$  sub-protocol followed by the computation of  $\frac{m}{\ell}$  segments, each of which has  $\ell$  multiplications. In all, at most  $n$  segments may fail and require repetition. Finally the protocol *NAVSS-REC* $[\mathcal{P}, \mathcal{A}, P]$  is performed  $O(n)$  times, possibly in parallel. The overall complexity is as follows.  $MC = \{O(d^3 \lg |\mathcal{F}|)\} + \{(\frac{m}{\ell} + n) O(\ell d^3 \lg |\mathcal{F}|)\}$  and  $BAC = \{(\frac{m}{\ell} + n) O(\ell nd)\}$ . When setting  $\ell = \frac{m}{n}$ , we have  $MC = O(md^3 \lg |\mathcal{F}|)$  bits and  $BAC = O(mnd)$  bits. When restricted to the threshold case ( $d = n$ ), the protocol broadcasts  $O(mn^2)$  bits which substantially improves over the protocol of [4] that broadcasts  $O(mn^4)$  bits.

## 7 Conclusions

In this work we have initiated the study of perfectly secure multiparty computation over asynchronous networks in the non-threshold adversarial model – we show that perfectly secure asynchronous multiparty computation is possible if and only if no *four* sets in the adversary structure cover the full player set. We provide efficient constructions whenever possible and remark that these constructions are as good as the *best* known asynchronous protocols when restricted to the threshold setting. We remark that drawing upon ideas in [8,6,17] and those in this paper, one can give similar constructions for unconditionally secure asynchronous multiparty computation with less stringent constraints on the adversary structure and much smaller communication complexities – unconditionally secure (i.e., with a non-zero but negligible error probability) asynchronous multiparty computation is possible if and only if no *three* sets in the adversary structure cover the full player set. Furthermore, we remark that by using techniques in [17], a much more efficient protocol is achievable.

## References

1. Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, pages 75–122, 1991.
2. Donald Beaver, Joan Feigenbaum, Joe Kilian, and Phillip Rogaway. Security with low communication overhead. In *CRYPTO '90*, pages 62–76, 1990.
3. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of 22nd ACM STOC*, pages 503–513, 1990.
4. M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computations. In *Proceedings of 25th ACM STOC*, pages 52–61, 1993.

5. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of 20th ACM STOC*, pages 1–10, 1988.
6. M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computation with optimal resilience. In *Proceedings of 13th ACM PODC*, pages 183–192, 1994.
7. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
8. R. Canetti and T. Rabin. Optimal asynchronous byzantine agreement. In *Proceedings of 25th ACM STOC*, pages 42–51, 1993.
9. D. Chaum, C. Crepeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of 20th ACM STOC*, pages 11–19, 1988.
10. R. Cramer, I. Damgard, and U. Maurer. Efficient general secure multiparty computation from any linear secret sharing scheme. In *EUROCRYPT2000*, LNCS, Springer-Verlag, 2000.
11. Ronald Cramer, Ivan Damgard, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In *EUROCRYPT '99*, volume 1592 of LNCS, pages 311–326, 1999.
12. Matthew K. Franklin and Moti Yung. Communication complexity of secure computation. In *Proceedings of 24th ACM STOC*, pages 699–710, 1992.
13. Rosario Gennaro, Micheal O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of 17th ACM PODC*, 1998.
14. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *19th ACM STOC*, pages 218–229. ACM Press, 1987.
15. M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multiparty computation. In *16th ACM PODC*, pages 25–34, August 1997.
16. M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, April 2000.
17. Martin Hirt and Ueli Maurer. Robustness for free in unconditional multi-party computation. In *CRYPTO '01*, LNCS. Springer-Verlag, 2001.
18. Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient multi-party computation. In *ASIACRYPT 2000*, LNCS. Springer-Verlag, December 2000.
19. M. V. N. Ashwin Kumar, K. Srinathan, and C. Pandu Rangan Asynchronous Perfectly Secure Computation tolerating Generalized Adversaries Technical Report, IITM, Chennai, February 2002.
20. M. Karchmer and A. Wigderson. On span programs. In *Proceedings of the 8th Annual IEEE Structure in Complexity Theory*, pages 102–111, 1993.
21. S. Micali and P. Rogaway. Secure computation. In *CRYPTO'91*, volume 576 of LNCS, pages 392–404. Springer-Verlag, 1991.
22. S. Micali and P. Rogaway. *Secure Computation: The information theoretic case.*, 1998. Former version: Secure Computation, In *CRYPTO '91*, volume 576 of LNCS, pages 392–404, Springer-Verlag, 1991.
23. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of 21st ACM STOC*, pages 73–85, 1989.