Simple and Efficient Perfectly-Secure Asynchronous MPC *

Zuzana Beerliová-Trubíniová and Martin Hirt

ETH Zurich, Department of Computer Science, CH-8092 Zurich
{bzuzana,hirt}@inf.ethz.ch

Abstract. Secure multi-party computation (MPC) allows a set of n players to securely compute an agreed function of their inputs, even when up to t players are under the control of an adversary. Known *asynchronous* MPC protocols require communication of at least $\Omega(n^3)$ (with cryptographic security), respectively $\Omega(n^4)$ (with information-theoretic security, but with error probability and non-optimal resilience) field elements per multiplication.

We present an asynchronous MPC protocol communicating $\mathcal{O}(n^3)$ field elements per multiplication. Our protocol provides perfect security against an active, adaptive adversary corrupting $t < n/4$ players, which is optimal. This communication complexity is to be compared with the most efficient previously known protocol for the same model, which requires $\Omega(n^5)$ field elements of communication (i.e., $\Omega(n^3)$ broadcasts). Our protocol is as efficient as the most efficient perfectly secure protocol for the synchronous model and the most efficient asynchronous protocol with cryptographic security.

Furthermore, we enhance our MPC protocol for a hybrid model. In the fully asynchronous model, up to t honest players might not be able to provide their input in the computation. In the hybrid model, all players are able to provide their input, given that the very first round of communication is synchronous. We provide an MPC protocol with communicating $\mathcal{O}(n^3)$ field elements per multiplication, where all players can provide their input if the first communication round turns out to be synchronous, and all but at most t players can provide their input if the communication is fully asynchronous. The protocol does not need to know whether or not the first communication round is synchronous, thus combining the advantages of the synchronous world and the asynchronous world. The proposed MPC protocol is the first protocol with this property.

Keywords: Multi-party computation, asynchronous, hybrid model, efficiency, perfect security.

* This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

K. Kurosawa (Ed.): ASIACRYPT 2007, LNCS 4833, pp. 376–392, 2007.

1 Introduction

1.1 Secure Multi-Party Computation

Secure multi-party computation (MPC) enables a set of n players to securely evaluate an agreed function of their inputs even when t of the players are corrupted by a central adversary. A *passive adversary* can read the internal state of the corrupted players, trying to obtain information about the honest players' inputs. An *active adversary* can additionally make the corrupted players deviate from the protocol, trying to falsify the outcome of the computation.

The MPC problem dates back to Yao [Yao82]. The first generic solutions presented in [GMW87, CDG87, GHY87] (based on cryptographic intractability assumptions) and later [BGW88, CCD88, RB89, Bea91] (with information-theoretic security) assume the existence of a synchronous network. Synchronous networks assume that there is a global clock, and the delay of any message in the network is bounded by a constant. Such networks do not well model real-life networks like the internet.

1.2 Asynchronous Networks

In asynchronous networks, messages are delayed arbitrarily. As worst-case assumption, the adversary is given the power to schedule the delivery of messages. Asynchronous communication models real-world networks (like the Internet) much better than synchronous communication. However, protocols for asynchronous networks are much more involved than their synchronous counterparts. This comes from the fact that when a player does not receive an expected message, he cannot decide whether the sender is corrupted (and did not send the message at all) or the message is just delayed in the network.

This implies also that in fully asynchronous settings it is impossible to consider the inputs of *all* uncorrupted players. The inputs of up to t (potentially honest) players have to be ignored, because waiting for them could turn out to be endless.

For a good introduction to asynchronous protocols, see [Can95]. Due to its complexity, asynchronous MPC has attracted much less research than synchronous MPC. The most important results on asynchronous MPC are [BCG93, BKR94, SR00, PSR02, HNP05].

In the asynchronous setting perfect information-theoretic security against an active adversary is possible if and only if $t < n/4$ (whereas cryptographic and unconditional security are possible if and only if $t < n/3$).

1.3 Communication Complexity of MPC protocols

The first proposed MPC protocols secure against active adversaries were very inefficient and so of theoretical relevance mainly. In the recent years lots of research concentrated on designing protocols with lower communication complexity (measured in bits sent by honest players). The currently most efficient MPC protocols for the synchronous model are [HMP00] (perfect security with $t < n/3$, $\mathcal{O}(n^3)$

communication per multiplication), [DN07] (information-theoretic security with $t < n/3$, $\mathcal{O}(n)$ communication per multiplication), [BH06] (information-theoretic security with $t < n/2$, communicating $\mathcal{O}(n^2)$ per multiplication), [HN06] (cryptographic security with $t < n/2$, communicating $\mathcal{O}(n)$ per multiplication).

However known MPC protocols for asynchronous networks still feature (impractically) high communication complexities. The most efficient asynchronous protocol is the one of [HNP05] communicating $\mathcal{O}(n^3)$ per multiplication while providing cryptographic security only. The most efficient information-theoretically secure protocols were proposed in [SR00, PSR02]. Both protocols are secure against an unbounded adversary corrupting up to $t < n/4$ players. The first one makes extensive use of the (communication-intensive) BA primitive – $\mathcal{O}(n^2)$ invocations per multiplication, which amounts to $\Omega(n^5)$[1] bits of communication per multiplication. The second one requires only $\mathcal{O}(n^2)$ invocations to BA in total, however, still communicates $\mathcal{O}(n^4)$ bits per multiplication, and provides unconditional security only (for which $t < n/4$ is not optimal).

1.4 Contributions

Known MPC protocols for the asynchronous setting suffer from two main disadvantages in contrast to their more restrictive synchronous counterparts, both significantly reducing their practicability: Asynchronous protocol tend to have substantially higher communication complexity, and they do not allow to take the inputs of all honest players. In this work, we propose a solution to both these problems.

First, we present an perfectly secure asynchronous MPC protocol that communicates only $\mathcal{O}(n^3)$ field elements per multiplication. This very same communication complexity is also required by the most efficient known perfectly secure protocol for the synchronous model [HMP00], as well as by the most efficient asynchronous protocol only secure against computationally bounded adversaries [HNP05]. The protocol provides perfect security against an unbounded adaptive active adversary corrupting up to $t < n/4$ players, which is optimal. In contrast to the previous asynchronous protocols, the new protocol is very simple.

Second, we extended the protocol for a hybrid communication model (with the same security properties and the same communication complexity), allowing *all* players to give input if the *very first round* of the communication is synchronous, and takes at least $n - t$ inputs in a fully asynchronous setting. It is well-known that fully asynchronous protocols cannot take the inputs of all players; however, we show that a single round of synchronous communication is sufficient to take all inputs. We stress that it is important that this round is the first round, because assuming the k-th round to be synchronous implies that all rounds up to k must also be synchronous. Furthermore, the protocol achieves the best of both worlds, i.e., takes the inputs of *all* players when indeed the first round is synchronous, and still takes the inputs of at least $n-t$ players even if the synchronity assumptions cannot be fulfilled. More precisely, the protocol takes

[1] The most efficient known asynchronous BA protocol requires $\Omega(n^3)$.

the inputs of at least $n - t$ players, and additionally, always takes the inputs of players whose first-round messages are delivered synchronously.

2 Preliminaries

2.1 Model

We consider a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \ldots, P_n\}$, which are connected with a complete network of secure (private and authentic) asynchronous channels. The function to be computed is specified as an arithmetic circuit over a finite field $\mathcal{F} = \mathbb{Z}_p$ (with $p > n$), with input, addition, multiplication, random, and output gates. We denote the number of gates of each type by c_I, c_A, c_M, c_R, and c_O, respectively.

The faultiness of players is modeled in terms of a central adversary corrupting players. The adversary can corrupt up to t players for any fixed t with $t < n/4$, and make them deviate from the protocol in any desired manner. The adversary is computationally unbounded, active, adaptive, and rushing. Furthermore, in order to model the asynchronism of the network, the adversary can schedule the delivery of the messages in the network, i.e., she can delay any message arbitrarily. In particular, the order of the messages does not have to be preserved. However, every sent message will eventually be delivered.

The security of our protocols is perfect, i.e., information-theoretic without any error probability.

2.2 Design of Asynchronous MPC Protocols

Asynchronous protocols are executed in *steps*. Each step begins by the scheduler choosing one message (out of the queue) to be delivered to its designated recipient. The recipient is activated by receiving the message, he performs some (internal) computation and possibly sends messages on his outgoing channel (and waits for the next message).

The action to be taken by the recipient is defined by the relevant sub-protocol[2] consisting of a number of instructions what is to be done upon receiving a specified message. If the received message refers to a sub-protocol which is not yet "in execution", then the player keeps the message until the relevant sub-protocol is invoked.

2.3 Partial Termination

Many "asymmetric" tasks with a designated dealer (broadcast, secret-sharing) cannot be implemented with guaranteed termination in an asynchronous world; the players cannot distinguish whether the dealer is corrupted and does not start the protocol, or the dealer is correct but his messages are delayed in the network. Hence, these protocol are required to terminate only if the dealer is correct. However, we require that if such a sub-protocol terminated for one (correct) player, then it must eventually terminate for all correct players.

[2] We assume that for each message it is clear to which sub-protocol it belongs.

The issue with partial termination is typically attacked by invoking n instances of the protocol with partial termination in parallel, every player acting as dealer in one instance. Then, every player can wait till $n - t$ instances have terminated (from his point of view). In order to reach agreement on the set of terminated instances, a specialized sub-protocol is invoked, called agreement on a core-set. A player can only be contained in the core-set if his protocol instance has terminated for at least one honest player, and hence will eventually terminate for all honest players. The core-set contains at least $n - t$ players.

2.4 Input Provision

Providing input is an inherently asymmetric task, and it is not possible to distinguish between a corrupted input player who does not send any message and a correct input player whose messages are delayed in the network. For this reason, in a fully asynchronous world it is not possible to take the inputs of *all* players; up to t (possible correct) players cannot be waited for, as this waiting could turn out to be endless. Hence, the protocol waits only till $n - t$ of the players have achieved to provide input, and then goes on with the computation.

2.5 Byzantine Agreement

We need three flavors of Byzantine agreement, namely broadcast, consensus, and core-set agreement.

The broadcast (BC) primitive allows a sender to distribute a message among the players such that all players get the same message (even when the sender is corrupted), and the message they get is the sender's message if he is honest. As explained above, broadcast cannot be realized with complete termination; instead, termination of all (correct) players is required only when the sender is correct; however, as soon as at least one correct player terminates, all players must eventually terminate. Such a broadcast primitive can be realized rather easily [Bra84]. The required communication for broadcasting an ℓ-bit message is $\mathcal{O}(n^2\ell)$, where the hidden constant is small.

Consensus enables a set of players to agree on a value. If all honest players start the consensus protocol with the same input value v then all honest players will eventually terminate the protocol with the same value v as output. If they start with different input values, then they will eventually reach agreement on some value. All known i.t.-secure asynchronous consensus protocols start by having every player broadcast his input value, which results to communication complexity $\Omega(n^3\ell)$, where ℓ denotes the length of the inputs.

Agreement on a core set (ACS) is a primitive presented in [BCG93]. We use it to determine a set of at least $n - t$ players that correctly shared their values. More concretely, every player starts the ACS protocol with a accumulative set of players who from his point of view correctly shared one or more values (the share sub-protocol in which they acted as dealers terminated properly). The output of the protocol is a set of at least $n - t$ players, who really correctly shared their values, which means that every honest player will eventually get a share of

every sharing dealt by a dealer from the core set. The communication cost of a ACS protocol are essentially the costs of n invocations to consensus (where the messages are index of players), i.e. $\Omega(n^4 \log n)$ bits.

2.6 Super-Invertible Matrices

We consider r-by-c matrices M over a field $\mathcal{F}$. When $r = c$, M is called *invertible* if all column-vectors are linearly independent. When $r \leq c$, M is called *super-invertible* if every subset of r column-vectors are linearly independent.

Formally, for an r-by-c matrix M and an index set $C \subseteq \{1, \ldots, c\}$, we denote by M_C the matrix consisting of the columns $i \in C$ of M. Then, M is super-invertible if for all C with $|C| = r$, M_C is invertible.

Super-invertible matrices over $\mathcal{F}$ can be constructed as follows: Fix c disjoint elements $\alpha_1, \ldots, \alpha_c \in \mathcal{F}$, and for $i = 1, \ldots, r$, let $f_i(\cdot)$ be a polynomial of degree at most $r - 1$ with $f_i(\alpha_i) = 1$ and $f_i(\alpha_j) = 0$ for $j \in \{1, \ldots, r\} \setminus \{i\}$. Then, $M = \{m_{i,j} = f_i(\alpha_j)\}$. M is super-invertible because $M_{\{1,\ldots,r\}}$ is invertible (it is the identity matrix), and any M_C for $C \subseteq \{1, \ldots, c\}$, $|C| = r$ can be mapped onto $M_{\{1,\ldots,r\}}$ using an invertible matrix given by Lagrange interpolation.

Super-invertible matrices are of great help to extract random elements from a set of some random and some non-random elements: Consider a vector $(x_1, \ldots, x_c)$ of elements, where for some $C \subseteq \{1, \ldots, c\}$ with $|C| = r$, the elements $\{x_i\}_{i \in C}$ are chosen uniformly at random (by honest players), and the elements $\{x_j\}_{j \notin C}$ are chosen maliciously (by corrupted players). Then, the vector $(y_1, \ldots, y_r) = M(x_1, \ldots, x_c)$ is uniformly random and unknown to the adversary.[3]

This means that given a super-invertible matrix and a set of c elements out of which at least r elements are chosen uniformly at random (and unknown to the adversary), we can generate r uniformly random elements (unknown to the adversary).

3 Protocol Overview

The new protocol proceeds in three phases: the preparation phase, the input phase and the computation phase. Every honest player will eventually complete every phase.

In the preparation phase many sharings of random values will be generated in parallel. For every multiplication gate, $3t + 1$ random sharing will be generated. For every random gate, one random sharing will be generated.

In the input phase the players share their inputs and agree on a core set of correctly shared inputs (every honest player will eventually get a share of every input from the core set).

In the computation phase, the actual circuit will be computed gate by gate, based on the core-set inputs. Due to the linearity of the used secret-sharing, the

[3] This follows from the observation that the $c - r$ maliciously chosen elements $\{x_j\}_{j \notin C}$ define a bijection from the r random elements $\{x_i\}_{i \in C}$ onto $(y_1, \ldots, y_r)$.

linear gates can be computed locally – without communication. Each multiplication gate will be evaluated with the help of $3t + 1$ of the prepared sharings.

4 Secret Sharing

4.1 Definitions and Notations

As secret-sharing scheme, we use the standard Shamir scheme [Sha79]: We say that a value s is *d-shared* if every correct player P_i is holding a share s_i of s, such that there exists a degree-d polynomial $p(x)$ with $p(0) = s$ and $p(i) = s_i$ for all $i = 1, \ldots, n$. We call the vector $(s_1, \ldots, s_n)$ of shares a d-sharing of s. A (possibly incomplete) set of shares is called *d-consistent* if these shares lie on a degree d polynomial.

Most of our Sharings will be t-sharings (where t denotes the maximum number of corrupted players). We denote a t-sharing of s by $[s]$. In the multiplication sub-protocol, we will also use $2t$-sharings, which will be denoted by $[[s]]$.

4.2 Share_1 and Recons— The Vanilla Protocols

In the following, we recap the Share_1 and Recons protocol of [BCG93].[4] Share_1 allows a dealer P_D to t-share a secret value $s \in \mathcal{F}$. Recons allows the players to reconstruct a d-sharing (for $d \leq 2t$) towards a receiver P_R. We stress that the protocol Share_1 does not necessarily terminate when the dealer P_D is corrupted. However, when it terminates for some correct player, then it eventually terminates for all players. The protocol Recons always terminates.

The intuition behind the protocol Share_1 is the following: In order to share a secret s, the dealer chooses a random two-dimensional polynomial $f(\cdot, \cdot)$ with $f(0, 0) = s$, and sends to every player P_i the polynomials $g_i(\cdot) = f(i, \cdot)$ and $h_i(\cdot) = f(\cdot, i)$. Then the players pairwisely check the consistency of the received polynomials, and publicly confirm successful checks. Once $n - t$ players are mutually consistent, the other players use the checking points received from these players to determine their respective polynomial $g_i(\cdot)$, and all players compute the share $s_i = g_i(0)$.

Protocol Share_1 (Dealer P_D, secret $s \in \mathcal{F}$).

- Distribution — Code for Dealer P_D: Choose a random two-dimensional degree-t polynomial $f(\cdot, \cdot)$ with $f(0, 0) = s$ and send to each player P_i the two degree-t polynomials $g_i(\cdot) = f(i, \cdot)$ and $h_i(\cdot) = f(\cdot, i)$.
- Consistency Checks — Code for player P_i:
  1. Wait for $g_i(\cdot)$ and $h_i(\cdot)$ from P_D.
  2. To each player P_j send the share-share $s_{ji} = h_i(j)$.
  3. Upon receiving s_{ij} from P_j check whether $s_{ij} = g_i(j)$. If so broadcast (ok, i, j).

[4] We denote their sharing protocol by Share_1, as it allows to share only one single value.

- Output-computing — Code for Player P_i:
  1. Wait until there is a $(n-t)$-clique in the graph implicitly defined by the broadcasted confirmations.[5]
  2. Upon receiving at least $2t+1$ t-consistent share-shares s_{ij} (for $j \in \{1,\ldots,n\}$) from the players in the clique, find the interpolation polynomial $\widetilde{g}_i(\cdot)$ and (re)compute your share $s_i = \widetilde{g}_i(0)$.[6]
  3. Output the share s_i.

Lemma 1. *For every coalition of up to t bad players and every scheduler, the protocol* Share_1 *achieves the following properties:*

- *Termination: If the dealer is correct, then every correct player will eventually complete* Share_1*, and if some correct player has completed* Share_1*, then all the correct players will eventually complete* Share_1*.*
- *Correctness: Once a correct player has completed* Share_1*, then there exists a unique value r which is t-shared among the players, where $r = s$ if the dealer is correct.*
- *Privacy: If the dealer is correct, then the adversary obtains no information about the shared secret.*

The communication complexity of Share_1 *is $\mathcal{O}(n^2\kappa + n^2\mathcal{BC}(\kappa))$.*

The intuition behind the protocol Recons is the following: Every player P_i sends his share s_i to P_R. The receiver waits until receiving at least $d+t+1$ d-consistent shares and outputs the value of their interpolation polynomial at 0. Note that corrupted players can send false shares to P_R, but for the latest when P_R has received the shares of all honest players, he has at least $n-t \geq d+t+1$ t-consistent shares (for $t < n/4$ and $d \leq 2t$).

Protocol Recons **(Receiver P_R, degree d, d-sharing of s).**

- Code for player P_i: Send s_i to P_R.
- Code for receiver P_R: Upon receiving at least $d+t+1$ d-consistent shares s_i (and up to t inconsistent shares), interpolate the polynomial $p(\cdot)$ and output $s = p(0)$.

Lemma 2. *For any d-shared value s, where $d+2t < n$, for every coalition of up to t bad players, and for every scheduler, the protocol* Recons *achieves the following properties:*

- *Termination: Every correct player will eventually complete* Recons*.*
- *Correctness: P_R will output s.*
- *Privacy: When P_R is honest, then the adversary obtains no information about the shared secret.*

The communication complexity of the protocol Recons *is $\mathcal{O}(n\kappa)$.*

[5] The graph has n nodes representing the n players and there is an edge between i and j if and only if both (ok, i, j) and (ok, j, i) were broadcasted.

[6] If the dealer is correct or if P_i is a member of the clique $g_i(\cdot) = \widetilde{g}_i(\cdot)$

Note that for $t < n/4$, Recons can be used to reconstruct t-sharings as well as $2t$-sharings. However, the protocol Share_1 can only generate t-sharings.

Proofs of security as well as details on solving the clique-problem in Share_1 (respectively, reducing it to a computationally simpler problem) and on finding (and interpolating) $d + t + 1$ d-consistent shares in Recons, can be found in [BCG93].

4.3 Share*: Sharing Many Values at Once

The following protocol Share^* extends the protocol Share_1 in two ways: First, it allows the dealer to share a vector $\left(s^{(1)}, \ldots, s^{(\ell)}\right)$ of ℓ secrets at once, substantially more efficient than ℓ independent invocations of Share_1. Secondly, Share^* allows to share "empty" secrets, formally $s^{(k)} = \perp$, resulting in all shares of $s^{(k)}$ being $\perp$ as well. This will be used when a dealer should share an unknown value.

Protocol Share^* **(Dealer** P_D**, secrets** $(s^{(1)}, \ldots, s^{(\ell)}) \in (\mathcal{F} \cup \{\perp\})^\ell$**).**

- Distribution — Code for Dealer P_D: For every $s^{(k)} \neq \perp$, choose a random two-dimensional degree-t polynomial $f^{(k)}(\cdot, \cdot)$ with $f^{(k)}(0,0) = s^{(k)}$. Send to every P_i the polynomials $\left(g_i^{(1)}, h_i^{(1)}, \ldots, g_i^{(\ell)}, h_i^{(\ell)}\right)$, where $g_i^{(k)}(\cdot) = f^{(k)}(i, \cdot)$ and $h_i^{(k)}(\cdot) = f^{(k)}(\cdot, i)$ if $s^{(k)} \in \mathcal{F}$, and $g_i^{(k)} = h_i^{(k)} = \perp$ if $s^{(k)} = \perp$.
- Consistency Checks — Code for player P_i:
  1. Wait for $\left(g_i^{(1)}, h_i^{(1)}, \ldots, g_i^{(\ell)}, h_i^{(\ell)}\right)$ from P_D.
  2. To each P_j send $\left(s_{ji}^{(1)}, \ldots, s_{ji}^{(\ell)}\right)$, where $s_{ji}^{(k)} = h_i^{(k)}(j)$, resp. $s_{ji}^{(k)} = \perp$ if $h_i^{(k)} = \perp$.
  3. Upon receiving $\left(s_{ij}^{(1)}, \ldots, s_{ij}^{(\ell)}\right)$ from P_j, broadcast (ok, i, j) if for all $k = 1, \ldots, \ell$ it holds that $s_{ij}^{(k)} = g_i^{(k)}(j)$, resp. $s_{ij}^{(k)} = \perp = g_i^{(k)}$.
- Output-computing — Code for Player P_i:
  1. Wait until there is a $(n-t)$-clique in the graph defined by the broadcasted confirmations.
  2. For $k = 1, \ldots, \ell$, upon receiving at least $2t + 1$ t-consistent share-shares $s_{ij}^{(k)}$ (for $j \in \{1, \ldots, n\}$) from the players in the clique, find the interpolation polynomial $\widetilde{g}_i^{(k)}(\cdot)$ and (re)compute the share $s_i^{(k)} = \widetilde{g}_i^{(k)}(0)$. Upon receiving $2t + 1$ values $s_{ij}^{(k)} = \perp$ (for $j \in \{1, \ldots, n\}$), set $s_i^{(k)} = \perp$.
  3. Output the shares $\left(s_i^{(1)}, \ldots, s_i^{(\ell)}\right)$.

Lemma 3. *The protocol* Share^* *allows* P_D *to share* ℓ *secrets from* $\mathcal{F} \cup \{\perp\}$ *at once, with the same security properties as required in Lemma 1. The communication complexity of* Share^* *is* $\mathcal{O}(\ell n^2 \kappa + n^2 \mathcal{BC}(\kappa))$.

5 Preparation Phase

The goal of the preparation phase is to generate t-sharings of ℓ uniformly random values $r^{(1)}, \ldots, r^{(\ell)}$, unknown to the adversary, where ℓ will be $c_M(3t + 1) + c_R$.

The idea of the protocol PreparationPhase is the following: First, every player acts as dealer in Share* to share a vector of $\ell' = \lceil \ell/(n-2t) \rceil$ random values. Then the players agree on a core set of $n-t$ correct dealers (such that their Share* protocol was completed by at least one honest player). This results in $n-t$ vectors of ℓ' correct t-sharings, but up to t of these vectors may be known to the adversary (and may not be random). Then, these $n-t$ correct vectors are compressed to $n-2t$ correct *random* vectors, unknown to the adversary, by using a $(n-2t)$-by-$(n-t)$ super-invertible matrix (applied component-wise). This computation is linear, hence the players can compute their shares of the compressed sharings locally from their shares of the original sharings.

Protocol PreparationPhase (ℓ)**.**
Code for player P_i:

- Secret Sharing
  - Act as a dealer in Share* to share a vector of $\ell' = \lceil \ell/(n-2t) \rceil$ random values $\left(s^{(i,1)}, \ldots, s^{(i,\ell')}\right)$.
  - For every $j = 1, \ldots, n$, take part in Share* with dealer P_j, resulting in the shares $\left(s_i^{(j,1)}, \ldots, s_i^{(j,\ell')}\right)$.
- Agreement on a Core Set
  1. Create an accumulative set $C_i = \emptyset$.
  2. Upon completing Share* with dealer P_j, include P_j in C_i.
  3. Take part in ACS with the accumulative set C_i as input.
- Compute Output (local computation)
  1. Wait until ACS completes with output C. For simple notation, assume that $\{P_1, \ldots, P_{n-t}\} \subseteq C$.
  2. For every $k \in \{1, \ldots, \ell'\}$, the $(n-2t)$ t-shared random values, unknown to the adversary, are defined as $\left(r^{(1,k)}, \ldots, r^{(n-2t,k)}\right) = M\left(s^{(1,k)}, \ldots, s^{(n-t,k)}\right)$, where M denotes a $(n-2t)$-by-$(n-t)$ super-invertible matrix, e.g., constructed according to Section 2.6. Compute your shares $\left(r_i^{(1,k)}, \ldots, r_i^{(n-2t,k)}\right)$ accordingly. Denote the resulting $\ell'(n-2t) \geq \ell$ sharings as $[r^{(1)}], \ldots, [r^{(\ell)}]$.

Lemma 4. PreparationPhase *(eventually) terminates for every honest player. It outputs independent random sharings of ℓ secret, independent, uniformly random values $r^(1), \ldots, r^{(\ell)}$.* PreparationPhase *communicates $\mathcal{O}(\ell n^2 \kappa + n^3 \mathcal{BC}(\kappa))$ bits and requires one invocation to ACS.*

6 Input Phase

In the InputPhase protocol every player P_i acts as a dealer in one Share* protocol in order to share his input s_i.[7] However the asynchronity of the network does not allow the players to wait for more than $n-t$ Share*-protocols to be completed.

[7] s_i can be one value or an arbitrary long vector of values from $\mathcal{F}$

In order to agree on the players whose inputs will be taken into to computation one ACS protocol is run.

Protocol InputPhase **(Every** P_i **has input** s_i**).**
Code for player P_i:

- Secret Sharing
  - Share your secret input s_i with Share*.
  - For every $j = 1, \ldots, n$ take part in Share* with dealer P_j.
- Agreement on a Core Set
  1. Create a accumulative set $C_i = \emptyset$.
  2. Upon completing Share* with dealer P_j, include P_j in C_i.
  3. Take part in ACS with your accumulative set C_i as your input.
  4. Output the agreed core set C and your outputs of the Share* protocols with dealers from C.

Lemma 5. *The* InputPhase *protocol will (eventually) terminate for every honest player. It enables the players to agree on a core set of at least* $n - t$ *players who correctly shared their inputs – every honest player will (eventually) complete the* Share* *protocol of every dealer from the core set (and get the correct shares of his shared input values).* InputPhase *communicates* $\mathcal{O}(c_I n^2 \kappa + n^3 \mathcal{BC}(\kappa))$ *bits and requires one invocation to* ACS*.*

7 Computation Phase

In the computation phase, the circuit is evaluated gate by gate, whereby all inputs and intermediate values are shared among the players. As soon as a player holds his shares of the input values of a gate, he joins the computation of the gate.

Due to the linearity of the secret-sharing scheme, linear gates can be computed locally simply by applying the linear function to the shares, i.e. for any linear function $f(\cdot, \cdot)$, a sharing $[c] = [f(a, b)]$ is computed by letting every player P_i compute $c_i = f(a_i, b_i)$. With every random gate, one random sharing (from the preparation phase) is associated, which is directly used as outcome of the random gate. With every multiplication gate, $3t + 1$ random sharings (from the preparation phase) are associated, which are used to compute a sharing of the product as described in the protocol Multiplication.

Protocol ComputationPhase **(**$(3t + 1)c_M + c_R$ **random sharings** $[r^{(1)}], \ldots, [r^{(\ell)}]$**).**
For every gate in the circuit — Code for player P_i:

1. Wait until you have shares of each of the inputs
2. Depending on the type of the gate, proceed as follows:
   - Linear gate $[c] = f([a], [b], \ldots)$: compute your share c_i as $c_i = f(a_i, b_i, \ldots)$.

- Multiplication gate $[c] = [a][b]$: participate in protocol $\mathsf{Multiplication}([a], [b], [r^{(0)}], \ldots, [r^{(3t+1)}])$, where $[r^{(0)}], \ldots, [r^{(3t+1)}]$ denote the $3t+1$ associated random sharing.
- Random gate $[r]$: set your share $r_i = r_i^{(k)}$, where $[r^{(k)}]$ denotes the associated random sharing.
- Output gate $[a] \rightarrow P_R$: participate in $\mathsf{Recons}(P_R, d = t, [a])$.

In order to compute multiplication gates, we use the approach of of [DN07]: First, the players jointly generate a secret random value s, which is both t-shared (by $[s]$) and $2t$-shared (by $[[s]]$). These sharings can easily be generated based on the $3t+1$ t-sharings associated with the multiplication gate. Then, every player locally multiplies his shares of a and b, resulting in a $2t$-sharing of the product $c = ab$, i.e., $[[c]]$. Then, the players compute and reconstruct $[[c - s]]$, resulting in every player knowing $d = c - s$, pick a default t-sharing $[d]$, and (locally) compute $[c] = [d] + [s]$, the correct product $[ab]$.

Protocol $\mathsf{Multiplication}$ **(**$[a], [b], [r^{(0)}], \ldots, [r^{(3t+1)}]$**).**
Code for player P_i:

1. PREPARE $[s]$: The degree-t polynomial $p(\cdot)$ to share s is defined by the shared coefficients $r^{(0)}, r^{(1)}, \ldots, r^{(t)}$. For every P_j, a sharing of his share $s_j = p(j)$ is defined as $[s_j] = [r^{(0)}] + [r^{(1)}]j + \ldots + [r^{(t)}]j^t$. Invoke $\mathsf{Recons}(P_j, d = t, [s_j])$ to let P_j learn his degree-t share s_j.
2. PREPARE $[[s]]$: The degree-$2t$ polynomial $p'(\cdot)$ to share s is defined by the shared coefficients $r^{(0)}, r^{(t+1)}, \ldots, r^{(3t)}$. For every P_j, a sharing of his share $s'_j = p'(j)$ is defined as $[s'_j] = [r^{(0)}] + [r^{(t+1)}]j + \ldots + [r^{(3t)}]j^{2t}$. Invoke $\mathsf{Recons}(P_j, d = t, [s'_j])$ to let P_j learn his degree-$2t$ share s'_j.
3. COMPUTE $[ab]$:
   1. Compute your degree-$2t$ share of $c = ab$ as $c_i = a_i b_i$, resulting in $[[c]]$.
   2. For every $j = 1, \ldots, n$, invoke Recons $(P_j, d = 2t, ([[c]] - [[s]]))$, resulting in every P_j knowing $d = c - s$.
   3. Define $[d]$ as default sharing of d, e.g., the constant degree-0 polynomial.
   4. Compute $[c] = [d] + [s]$.

Lemma 6. *The protocol* $\mathsf{Multiplication}$ *(eventually) terminates for every honest player. Given correct sharings* $[a], [b], [r^{(0)}], \ldots, [r^{(3t+1)}]$ *as input, it outputs a correct sharing* $[ab]$*. The privacy is maintained when* $([r^{(0)}], \ldots, [r^{(3t+1)}])$ *are sharings of random values unknown to the adversary.* $\mathsf{Multiplication}$ *communicated* $\mathcal{O}(n^2\kappa)$ *bits.*

Lemma 7. *The protocol* $\mathsf{ComputationPhase}$ *(eventually) terminates for every honest player. Given that the* $\ell = (3t+1)c_M + c_R$ *sharings* $[r^{(1)}], \ldots, [r^{(\ell)}]$ *are correct* t*-sharings of random values, unknown to the adversary, it computes the outputs of the circuit correctly and privately, while communicating* $\mathcal{O}(n^2 c_M + nc_O\kappa)$ *bits (where* c_M*,* c_R*, and* c_O *denote the number of multiplication, random, and output gates in the circuit, respectively).*

8 The Asynchronous MPC Protocol

The following protocol allows the players to evaluate an agreed arithmetic circuit C of a finite field $\mathcal{F}$: Denote the number of input, multiplication, random and output gates as c_I, c_M, c_R, c_O, respectively.

Protocol AsyncMPC **(C, c_I, c_M, c_R, c_O).**

1. Invoke PreparationPhase to generate $\ell = c_M(3t+1) + c_R$ random sharings.
2. Invoke InputPhase to let the players share their inputs.
3. Invoke ComputationPhase to evaluate the circuit (consisting of linear, multiplication, random, and output gates).

Theorem 1. *For every coalition of up to $t < n/4$ bad players and for every scheduler, the protocol* AsyncMPC *securely computes the circuit C.* AsyncMPC *communicates $\mathcal{O}\big((c_I n^2 + c_M n^3 + c_R n^2 + n c_O)\kappa + n^3 \mathcal{BC}(\kappa)\big)$ bits and requires* 2 *invocations to* ACS,[8] *(which requires $\mathcal{O}(n^2 \mathcal{BC}(\kappa))$).*

9 The Hybrid Model

9.1 Motivation

A big disadvantage of asynchronous networks is the fact that the inputs of up to t honest players cannot be considered in the computation. This restriction disqualifies fully asynchronous models for many real-world applications. Unfortunately, this drawback is intrinsic to the asynchronous model, no (what so ever clever) protocol can circumvent it. The only escape is to move to less general communication models, where at least some restriction on the scheduling of messages is given.

In [HNP05], an asynchronous (cryptographically secure) MPC protocol was presented in which all players can provide their inputs, given that one single round of communication is synchronous. However, this protocol has two serious drawbacks: First, the communication round which is required to be synchronous is round number 7 (we say that a message belongs to round k if it depends on a message received in round $k-1$). This essentially means that *the first* 7 *rounds* must be synchronous, because if not, then the synchronous round can never be started (the players would have to wait until all messages of round 6 are delivered — an endless wait in an asynchronous network).

The second drawback of this protocol is that one must decide a priori the mode in which the protocol is to be executed, namely either in the hybrid mode (with the risk that the protocol fails when some message in the first 7 rounds is not delivered synchronously), or in the fully asynchronous mode (with the risk that up to t honest players cannot provide their input, even when the network is synchronous).

[8] The protocol can easily be modified to use only a single invocation to ACS, by invoking PreparationPhase and InputPhase in parallel, and invoking ACS to find those dealers who have both correctly shared their input(s) as well as correctly shared enough random values.

9.2 Our Hybrid Model

We follow the approach of [HNP05], but strengthen it in both mentioned directions: First, we require only *the very first round* to be synchronous, and second, we guarantee that even if some messages in the first round are not delivered synchronously, still at least $n - t$ inputs are provided — so to speak the best of both worlds. A bit more precisely, we provide a fully asynchronous input protocol with the following properties:

- For every scheduler, the inputs of at least $n - t$ players are taken.
- If all messages sent by P_i in the very first round of communication are delivered synchronously, then P_i's inputs are taken.

This means in particular that if the first round is fully synchronous, then the inputs of all honest players are taken, and if the network is fully asynchronous, then at least $n - t$ inputs are taken.

9.3 PrepareInputs and RestoreInput

We briefly describe the idea of the new input protocol (assuming, for the sake of simple notation, that every player gives exactly one input): In the first (supposedly synchronous) round, every player computes a degree-t Shamir-sharing of his input and sends one share to each player. Then, the players invoke the fully asynchronous input protocol, where the input of each player is a vector consisting of his real input, and his shares of the inputs of the other players. As result of the asynchronous input protocol, a core set C of at least $n - t$ players is found, whose input vectors are (eventually) t-shared among the players. For every player $P_i \in C$, the input is directly taken from his input vector. For every player $P_j \notin C$, the input is computed as follows: There are $n - t$ shares of his input, each t-shared as a component of the input vector of some player $P_i \in C$. Up to t of these players might be corrupted and have input a wrong share. Therefore, these t-shared shares are error-corrected and used as P_j's input. For error correction, $t + 1$ random t-sharings are used. These will be generated (additionally) in the preparation phase. Then, right before the computation phase, sharings of the missing inputs are computed.

In the following, we present a (trivial) sub-protocol PrepareInputs, which prepares the inputs of all players (to be invoked in the first, supposedly synchronous round), and a protocol RestoreInput, which restores the sharing of an input $s^{(k)}$ of a player not in the core set, if possible (to be invoked right before the computation phase). The protocol RestoreInput needs $t+1$ t-sharings of random values, which must be generated in the preparation phase.

Protocol PrepareInputs **(every** P_i **holding input** $s^{(i)}$**).**
Code for player P_i:

1. Choose random degree-t polynomial $p(\cdot)$ with $p(0) = s^{(i)}$ and send to every P_j his share $s_j^{(i)} = p(j)$.

2. Collect shares $s_i^{(j)}$ (from P_j) till the first round is over. Then compose your new input $\widetilde{s}^{(i)} = \left(s^{(i)}, s_i^{(1)}, \ldots, s_i^{(n)}\right)$, where $s_i^{(j)} = \perp$ if no share $s_i^{(j)}$ was received from P_j within the first round.

Protocol RestoreInput **(Core Set** C**, Input Sharings** $[\widetilde{s}^{(i)}]$ **of** $P_i \in C$**,** $[r^{(0)}], \ldots, [r^{(t+1)}]$**,** k**).**
Code for player P_i:

1. Define the blinding polynomial $b(x) = r^{(0} + r^{(1)}x + \ldots + r^{(t)}x^t$, and for every P_j, define $[b_j] = [b(j)] = [r^{(0)}] + [r^{(1)}]j + \ldots + [r^{(t)}]j^t$. Invoke Recons to reconstruct b_j towards P_j, for every P_j.
2. For every $P_j \in C$, denote by $[s_j^{(k)}]$ the sharing of P_j's share of P_k's input $s^{(k)}$. Note that $[s_j^{(k)}]$ is a part of the input vector $[\widetilde{s}^{(j)}]$. If $[s_j^{(k)}] \neq \perp$, then compute $[d_j] = [s_j^{(k)}] + [b_j]$, and invoke Recons to reconstruct d_j towards every player.
3. If there exists a degree-t polynomial $p(\cdot)$ such that at least $2t+1$ of the reconstructed values d_j lie on it, define $d'_i = p(i)$, and compute your share $s_i^{(k)}$ of P_k's input $s^{(k)}$ as $d'_i - b_i$. The sharing of input $[s^{(k)}]$ was successfully restored. If no such polynomial $p(\cdot)$ exists, then $[s^{(k)}]$ cannot be restored.

Lemma 8. *The protocol* PrepareInputs *and* RestoreInput *terminate for all players. When all messages of a player P_k in Step 1 of* PrepareInputs *are synchronously delivered, then a sharing of his input $s^{(k)}$ can be successfully restored in* RestoreInput*, by any core set C with $C \geq n - t$ (with up to t cheaters. When an input sharing $[s^{(k)}]$ of an honest player P_k is restored in* RestoreInput*, then the shared value is the correct input of P_k. Furthermore, both* PrepareInputs *and* RestoreInput *preserve the privacy of inputs of honest players.*

Proof (sketch). Termination and privacy are easy to verify. We focus on correctness. First assume that P_k is honest, and all his messages in Round 1 of PrepareInputs were synchronously delivered. Then every honest player P_i embeds the share $s_i^{(k)}$ in his input vector. There will be at least $n - t$ players in the core set, so at least $n - 2t$ honest players P_j. This means that there are at least $n - 2t$ t-consistent shares $s_j^{(k)}$, and hence, at least $n - 2t$ consistent shares d_j. For $t < n/4$, we have $n - 2t \geq 2t + 1$, and the result is a sharing of $d - b = (s^{(k)} + b) - b = s^{(k)}$. Then assume that P_k is honest, but not all his messages in Round 1 have been delivered synchronously. However, if there are $2t+1$ points on the polynomial $p(\cdot)$, at least $t+1$ of these points are from honest players, and hence the right input is restored.

9.4 The Hybrid MPC Protocol

The new main protocol for the hybrid model is as follows:

Protocol HybridMPC (C, c_I, c_M, c_R, c_O)**.**

1. Invoke PrepareInputs to let every P_i with input $s^{(i)}$ Shamir share $s^{(i)}$ among all players.
2. Invoke PreparationPhase to generate $\ell = c_M(3t+1) + c_R + c_I(t+1)$ random sharings.
3. Invoke InputPhase (with P_i's input being the vector $\widetilde{s}^{(i)}$) to let the players share their input vectors.
4. Invoke RestoreInput to restore the inputs of every P_k not in the core set.
5. Invoke ComputationPhase to evaluate the circuit (consisting of linear, multiplication, random, and output gates).

Theorem 2. *For every coalition of up to $t < n/4$ bad players and for every scheduler, the protocol* HybridMPC *securely computes the circuit C, taking the inputs of all players (when the first round is synchronous), or taking the inputs of at least $n-t$ players (independently of any scheduling assumptions).* AsyncMPC *communicates $\mathcal{O}\big((c_I n^3 + c_M n^3 + c_R n^2 + n c_O)\kappa + n^3 \mathcal{BC}(\kappa)\big)$ bits and requires 2 invocations to* ACS *(can be reduced to 1).*

10 Conclusions

We have presented an MPC protocol for the fully asynchronous model, which is perfectly secure against an active, adaptive adversary, corrupting up to $t < n/4$ players, what is optimal. The protocol communicates only $\mathcal{O}(n^3)$ field elements per multiplication. Even in the synchronous model, no perfectly secure MPC protocol with better communication complexity is known. Furthermore, the protocol is as efficient as the most efficient protocol for the asynchronous model, which provides only cryptographic security.

Furthermore, we have enhanced the protocol for a hybrid communication mode, where the inputs of all players can be taken under the only assumption that the very first communication round is synchronous. This assumption is very realistic, as anyway the players have to agree on set of involved players, on the circuit to be evaluated, etc. The proposed protocol combines best of both the hybrid model and the fully asynchronous model; it allows at least $n-t$ players provide their input (even when the communication is fully asynchronous), and additionally guarantees that the input of every player is taken, as long as his first-round messages are delivered synchronously.

Lastly, the proposed protocol is conceptually very simple. It uses neither player elimination nor repetition.

References

[BCG93] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *Proc. 25th STOC*, pp. 52–61, 1993.

[Bea91] D. Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, pp. 75–122, 1991.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th STOC*, pp. 1–10, 1988.

[BH06] Z. Beerliova-Trubiniova and M. Hirt. Efficient multi-party computation with dispute control. In *TCC 2006*, LNCS 3876, pp. 305–328, 2006.

[BKR94] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proc. 13th PODC*, pp. 183–192, 1994.

[Bra84] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proc. 3rd PODC*, pp. 154–162, 1984.

[Can95] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute, Israel, June 1995.

[CCD88] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. 20th STOC*, pp. 11–19, 1988.

[CDG87] D. Chaum, I. Damgård, and J. van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *CRYPTO '87*, LNCS 293, pp. 87–119, 1987.

[DN07] I. Damgård and J. B. Nielsen. Robust multiparty computation with linear communication complexity. In *CRYPTO 2007*, LNCS 4622, 2007.

[GHY87] Z. Galil, S. Haber, and M. Yung. Cryptographic computation: Secure fault-tolerant protocols and the public-key model. In *CRYPTO '87*, LNCS 293, pp. 135–155, 1987.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proc. 19th STOC*, pp. 218–229, 1987.

[HMP00] M. Hirt, U. Maurer, and B. Przydatek. Efficient secure multi-party computation. In *ASIACRYPT 2000*, LNCS 1976, pp. 143–161, 2000.

[HN06] M. Hirt and J. B. Nielsen. Robust multiparty computation with linear communication complexity. In *CRYPTO 2006*, LNCS 4117, pp. 463–482, 2006.

[HNP05] M. Hirt, J. B. Nielsen, and B. Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience. In *EUROCRYPT 2005*, LNCS 3494, pp. 322–340, May 2005.

[PSR02] B. Prabhu, K. Srinathan, and C. P. Rangan. Asynchronous unconditionally secure computation: An efficiency improvement. In *Indocrypt 2002*, LNCS 2551, 2002.

[RB89] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proc. 21st STOC*, pp. 73–85, 1989.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[SR00] K. Srinathan and C. P. Rangan. Efficient asynchronous secure multiparty distributed computation. In *Indocrypt 2000*, LNCS 1977, Dec. 2000.

[Yao82] A. C. Yao. Protocols for secure computations. In *Proc. 23rd FOCS*, pp. 160–164, 1982.