Perfectly-Secure MPC with Linear Communication Complexity*

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, even when up to t players are under the control of an adversary. Known *perfectly secure* MPC protocols require communication of at least $\Omega(n^3)$ field elements per multiplication, whereas cryptographic or unconditional security is possible with communication linear in the number of players. We present a perfectly secure MPC protocol communicating $\mathcal{O}(n)$ field elements per multiplication. Our protocol provides perfect security against an active, adaptive adversary corrupting $t < n/3$ players, which is optimal. Thus our protocol improves the security of the most efficient information-theoretically secure protocol at no extra costs, respectively improves the efficiency of perfectly secure MPC protocols by a factor of $\Omega(n^2)$. To achieve this, we introduce a novel technique – constructing detectable protocols with the help of so-called hyper-invertible matrices, which we believe to be of independent interest. Hyper-invertible matrices allow (among other things) to perform efficient correctness checks of many instances in parallel, which was until now possible only if error-probability was allowed.

Keywords: Multi-party computation, efficiency, perfect security, hyper-invertible matrix.

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 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 some information he is not entitled to. An *active adversary* can additionally make the corrupted players deviate from the protocol, trying to falsify the outcome of the computation. In this work, we consider active adversaries.

The MPC problem dates back to Yao [Yao82]. The first generic solutions presented in [GMW87, CDvdG87, GHY87] (based on cryptographic intractability assumptions) and later [BGW88, CCD88, RB89, Bea91b] (with information-theoretic security) are rather inefficient and thus of theoretical interest mainly.

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

R. Canetti (Ed.): TCC 2008, LNCS 4948, pp. 213–230, 2008.

1.2 Efficiency of MPC Protocols

In the recent years lots of research concentrated on designing protocols with lower communication complexity. In this paper we concentrate on bit-complexity, measured in bits sent by honest players. The following table gives an overview on the currently most efficient MPC protocols (in the respective security model), where κ denotes the bit-length of a field element (resp. the security parameter).

Thresh.	Security	Bits/Mult.	Reference
$t < n/3$	perfect	$\mathcal{O}(n^3\kappa)$	[HMP00]
$t < n/2$	unconditional	$\mathcal{O}(n^2\kappa)$	[BH06]
$t < n/2$	cryptographic	$\mathcal{O}(n\kappa)$	[HN06]
$t < n/3$	unconditional	$\mathcal{O}(n\kappa)$	[DN07]

All above protocols use "player elimination" (or its generalization "dispute control") – a technique that enables converting non-robust (but detectable) protocols into robust protocols, essentially without any efficiency loss. Furthermore, all but the perfectly secure protocol use circuit randomization [Bea91a], which reduces the multiplication of two shared values to two reconstructions, given a precomputed sharing of a random multiplication triple (a, b, c) with $c = ab$. Such triples can be non-robustly generated and checked in advance – making use of parallelization. Checking the correctness of many instances in parallel can be done very efficiently when negligible error-probability is allowed, however until now no perfectly secure efficient parallel correctness-checks are known.

1.3 Contributions

In this paper, we present a novel technique which, at the same time, allows to perfectly and very efficiently verify a bunch of sharings and (if the check says that they are correct) to extract a set of (new) correct random sharings given that a sub-set of the original sharings is random.

More precisely, given n supposedly random sharings, up to t of them distributed by corrupted players (and thus possibly of a wrong degree, non-random, etc), we can check whether they are all correct and if so (locally) compute $n - 2t$ correct and uniform random sharings. The check is (despite of being perfectly secure) highly efficient; it only requires the reconstruction of $2t$ sharings, each towards a single player.

In other words, we can non-robustly but detectably generate $\Omega(n)$ uniform random sharings, unknown to the adversary, with perfect security and communicating $\mathcal{O}(n^2)$ field elements. By now, similarly efficient protocols to generate random sharings are known only with probabilistic checks, which provides a lower level of security and is less elegant.

The novel technique is based on so-called *hyper-invertible matrices*, i.e., matrices whose every square sub-matrix is invertible. Applying n sharings to such a matrix results in n sharings with the property that (i) if *any* (up to t) of the inputs sharings are broken, then this can be seen in *every* subset of t output sharings, and (ii) if *any* $n - t$ input sharings are uniform random, then *every* subset of size $n - t$ of output sharings is uniform random.

Using hyper-invertible matrices and some techniques from [Bea91a, HMP00, DN07], we construct a perfectly secure multi-party protocol with optimal resilience and linear communication complexity. This can be seen as an efficiency improvement (the most efficient known MPC protocol with perfect security communicates $\mathcal{O}(n^3)$ field elements per multiplication [HMP00]), or alternatively as a security improvement (the most secure known MPC protocol with linear communication provides error probability [DN07]). In either case, we consider the new protocol to be more elegant, as it employs neither two-dimensional sharings (like all previous perfectly-secure MPC protocols) nor probabilistic checks (like all previous MPC protocols with linear communication complexity).

2 Preliminaries

2.1 Model

We consider a set $\mathcal{U}$ of users, who can give input and receive output, and a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \ldots, P_n\}$, who perform the computation. The players and users are connected by a complete network of secure (private and authentic) synchronous channels.

The function to be computed is specified as an arithmetic circuit over a finite field $\mathcal{F}$ (with $|\mathcal{F}| > 2n$), 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 or users is modeled in terms of a central adversary corrupting players and users. The adversary can corrupt up to t players for any fixed t with $t < n/3$ and any number of users, and make them deviate from the protocol in any desired manner. The adversary is computationally unbounded, active, adaptive, and rushing. The security of our protocols is perfect, i.e., information-theoretic without any error probability.

To every player $P_i \in \mathcal{P}$ a unique, non-zero element $\alpha_i \in \mathcal{F} \setminus \{0\}$ is assigned.

For the ease of presentation, we always assume that the messages sent through the channels are from the right domain — if a player receives a message which is not in the right domain (e.g., no message at all), he replaces it with an arbitrary message from the specified domain.

2.2 Byzantine Agreement

In our multi-party protocol we use Byzantine agreement in both its shapes, broadcast and consensus. Broadcast allows a sender to distribute a value x, such that all players receive the same value x' (even if the sender is faulty), and $x = x'$ if the sender is honest. Consensus allows the players, each holding an input x_i, to reach agreement on a value x', where $x = x'$ if every honest players holds $x_i = x$. For $t < n/3$, both broadcast and consensus can be simulated with perfect security by a sub-protocol communicating $\mathcal{O}(n^2)$ bits [BGP92, CW92]. We denote the communication complexity needed for agreeing on a k bit message as $\mathcal{BA}(k) = n^2k$.

2.3 Player-Elimination Framework

Player Elimination [HMP00] is a general technique, used for constructing efficient MPC protocols. It allows to transform (typically very efficient) non-robust protocols into robust protocols at essentially no additional costs.

The basic idea is to divide the computation into segments and repeat the non-robust evaluation of each segment until it succeeds, whereby limiting the total number of times the adversary can cause a segment to fail. Each evaluation of a segment proceeds in three steps: (1.) detectable computation (2.) fault detection and (3.) fault localization.

Definition 1. *A* detectable *protocol is a passively secure protocol that can (in the presence of an active adversary) produce incorrect output, however this will be detected by at least one honest player. We say that after detecting a fault the player* gets unhappy *(sets his happy-bit to unhappy).*

In the detectable computation, the actual non-robust (but detectable) protocol is invoked to compute the segment. In the fault detection the players agree on whether or not there are some unhappy players. If all players are happy the computation of the segment was successful, the players keep the output and proceed to the next segment. Otherwise the segment failed, the output is discarded and a pair of players $E = \{P_i, P_j\}$ containing at least one corrupted player is localized in the fault localization, eliminated from the actual player set and the segment is repeated with the new player set.[1] We denote the original player set as $\mathcal{P}$ (containing n players, up to t of them faulty), and the actual (reduced) player set as $\mathcal{P}'$ (containing n' players, up to t' of them faulty).

By selecting the size of a segment such that there are t segments, the overall costs of the resulting robust protocol are at most twice the costs of the non-robust protocol (plus the overhead costs for the fault detection and the player elimination).

Special care needs to be taken such that the computation after a (sequence of) player elimination is "compatible" with the outputs of previous segments. We ensure this compatibility be fixing the degree of all sharings to t, independent of the actual threshold t'. Note that a sharing (among $\mathcal{P}'$) of degree t can be reconstructed as long as $t + 2t' < n'$, what is clearly satisfied when $t < n/3$.

Technically, a player-elimination protocol proceeds as follows:

[1] Note that we eliminate *players* and not *users*. If a party playing the role of a player as well as the role of a user is eliminated from the player set, it still keeps its user role – can give input and receive output.

Protocol with Player-Elimination.
Let $\mathcal{P}' \leftarrow \mathcal{P}$, $n' \leftarrow n$, $t' \leftarrow t$. Divide computation into t segments of similar size, and do the following for each segment:

0. Every $P_i \in \mathcal{P}'$ sets his happy-bit to happy (i.e., P_i did not observe a fault).
1. DETECTABLE COMPUTATION: Compute the actual segment in detectable manner, such that (i) if all players in $\mathcal{P}'$ follow their protocol, then the computation succeeds and all players remain happy, and (ii) if the output is incorrect, then at least one honest player in $\mathcal{P}'$ detects so and gets unhappy.
2. FAULT DETECTION: Reach agreement on whether or not all players in $\mathcal{P}'$ are happy (involves Byzantine Agreement). If all players are happy, proceed with the next segment. If at least one player is unhappy, proceed with the following fault-localization procedure.
3. FAULT LOCALIZATION: Find $E \subseteq \mathcal{P}'$ with $|E| = 2$, containing at least one corrupted player.
4. PLAYER ELIMINATION: Set $\mathcal{P}' \leftarrow \mathcal{P}' \setminus E$, $n' \leftarrow n' - 2$, $t' \leftarrow t' - 1$, and repeat the segment.

2.4 Circuit Randomization

Circuit randomization [Bea91a] allows to compute a sharing $[z]$ of the product z of two factors x and y, shared as $[x]$ and $[y]$, at the costs of two public reconstructions, when a pre-shared random triple $([a], [b], [c])$ with $c = ab$ is available. This technique allows to first prepare c_M shared multiplication triples $([a], [b], [c])$, and then to evaluate a circuit with c_M multiplication by a sequence of public reconstructions.

The trick of circuit randomization is that $z = xy$ can be expressed as $z = \big((x-a)+a\big)\big((y-b)+b\big)$, hence $z = de+db+ae+c$, where (a, b, c) is a multiplication triple and $d = x - a$ and $e = y - b$. For a random multiplication triple, d and e are random values independent of x and y, hence a sharing $[z]$ can be linearly computed as $[z] = [de] + d[b] + e[a] + [c]$, by reconstructing $[d] = [x] - [a]$ and $[e] = [y] - [b]$.

3 Hyper-Invertible Matrices

3.1 Definition

A hyper-invertible matrix is a matrix of which every (non-trivial) square sub-matrix is invertible.

Definition 2. *An r-by-c matrix M is* hyper-invertible *if for any index sets $R \subseteq \{1, \ldots, r\}$ and $C \subseteq \{1, \ldots, c\}$ with $|R| = |C| > 0$, the matrix M_R^C is invertible, where M_R denotes the matrix consisting of the rows $i \in R$ of M, M^C denotes the matrix consisting of the columns $j \in C$ of M, and $M_R^C = \big(M_R\big)^C$.*

3.2 Construction

We present a construction of a hyper-invertible n-by-n matrix M over a finite field $\mathcal{F}$ with $|\mathcal{F}| \geq 2n$. A hyper-invertible r-by-c matrix can be extracted as a sub-matrix of such a matrix with $n = \max(r, c)$.

Construction 1. Let $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ denote fixed distinct elements in $\mathcal{F}$, and consider the function $f: \mathcal{F}^n \rightarrow \mathcal{F}^n$, mapping $(x_1, \ldots, x_n)$ to $(y_1, \ldots, y_n)$ such that the points $(\beta_1, y_1), \ldots, (\beta_n, y_n)$ lie on the polynomial $g(\cdot)$ of degree $n-1$ defined by the points $(\alpha_1, x_1), \ldots, (\alpha_n, x_n)$. Due to the linearity of Lagrange interpolation, f is linear and can be expressed as a matrix $M = \{\lambda_{i,j}\}_{i=1,\ldots,n}^{j=1,\ldots n}$, where $\lambda_{i,j} = \prod_{\substack{k=1\\k\neq j}}^{n} \frac{\beta_i - \alpha_k}{\alpha_j - \alpha_k}$.

Lemma 1. *Construction 1 yields a hyper-invertible n-by-n matrix M.*

Proof. We have to show that for any index sets $R, C \subseteq \{1, \ldots, n\}$ with $|R| = |C| > 0$, M_R^C is invertible. As $|R| = |C|$, it is sufficient to show that the mapping defined by M_R^C is surjective, i.e., for every $\vec{y}_R$ there exists an $\vec{x}_C$ such that $\vec{y}_R = M_R^C \vec{x}_C$. Equivalently, we show that for every $\vec{y}_R$ there exists an $\vec{x}$ such that $\vec{y}_R = M_R \vec{x}$ and $\vec{x}_{\overline{C}} = \vec{0}$, where $\overline{C} = \{1, \ldots, n\} \setminus C$. Remember that M is defined such that the points $(\alpha_1, x_1), \ldots, (\alpha_n, x_n), (\beta_1, y_1), \ldots, (\beta_n, y_n)$ lie on a polynomial $g(\cdot)$ of degree $n-1$. Given the n points $\{(\alpha_j, 0)\}_{j \notin C}$ and $\{(\beta_i, y_i)\}_{i \in R}$, the polynomial $g(\cdot)$ can be determined by Lagrange interpolation, and $\vec{x}_C$ can be computed linearly from $\vec{y}_R$. Hence, M_R^C is invertible. □

3.3 Properties

The mappings defined by hyper-invertible matrices have a very nice symmetry property: Any subset of n input/output values can be expressed as a linear function of the remaining n input/output values:

Lemma 2. *Let M be a hyper-invertible n-by-n matrix and $(y_1, \ldots, y_n) = M(x_1, \ldots, x_n)$. Then for any index sets $A, B \subseteq \{1, \ldots, n\}$ with $|A| + |B| = n$, there exists an invertible linear function $f: \mathcal{F}^n \rightarrow \mathcal{F}^n$, mapping the values $\{x_i\}_{i \in A}, \{y_i\}_{i \in B}$ onto the values $\{x_i\}_{i \notin A}, \{y_i\}_{i \notin B}$.*

Proof. We have $\vec{y} = M\vec{x}$ and $\vec{y}_B = M_B \vec{x} = M_B^A \vec{x}_A + M_B^{\overline{A}} \vec{x}_{\overline{A}}$. Due to hyper-invertibility, $M_B^{\overline{A}}$ is invertible, and $\vec{x}_{\overline{A}} = \left(M_B^{\overline{A}}\right)^{-1} \left(\vec{y}_B - M_B^A \vec{x}_A\right)$. $\vec{y}_{\overline{B}}$ can be computed similarly. □

4 Protocol Overview

The new MPC protocol proceeds in two phases: the preparation phase and the computation phase.

In the preparation phase, degree-t sharings of random (a, b, c)-triples are generated (in parallel), one for every multiplication gate. Furthermore, for every random gate as well as for every input gate, a t-sharing of a random r is generated. For the sake of simplicity, we generate $c_M + c_R + c_I$ random triples, where for random and input gates, only the first component is used. The preparation phase makes use of the player-elimination technique.

In the computation phase, the actual circuit is computed. Input gates are evaluated with help of a pre-shared random value r. Due to the linearity of the used secret-sharing, the linear gates can be computed locally – without communication. Random gates are evaluated simply by picking an unused pre-shared random value r. Multiplication gates are evaluated with help of one prepared (a, b, c)-triple, using Beaver's circuit randomization technique [Bea91a]. Output gates involve a (robust) secret reconstruction.

5 Secret Sharing

5.1 Definitions and Notation

As secret-sharing scheme, we use the standard Shamir sharing scheme [Sha79].

Definition 3. *We say that a value s is (correctly) d-*shared *(among the players in $\mathcal{P}'$) if every honest player $P_i \in \mathcal{P}'$ is holding a share s_i of s, such that there exists a degree-d polynomial $p(\cdot)$ with $p(0) = s$ and $p(\alpha_i) = s_i$ for every $P_i \in \mathcal{P}'$.*[2] *The vector $(s_1, \ldots, s_{n'})$ of shares is called a d-sharing of s, and is denoted by $[s]_d$. A (possibly incomplete) set of shares is called d-*consistent *if these shares lie on a degree d polynomial.*

Most of the sharings used in our protocol are t-sharings – denoted as $[\cdot]_t$. In the preparation phase we also temporarily use t'- and $2t'$-sharings (denoted by $[\cdot]_{t'}$ and $[\cdot]_{2t'}$, respectively).

By saying that the players in $\mathcal{P}'$ compute (locally) $([y^{(1)}]_{d'}, \ldots, [y^{(m')}]_{d'}) = f([x^{(1)}]_d, \ldots, [x^{(m)}]_d)$ (for any function $f : \mathcal{F}^m \to \mathcal{F}^{m'}$) we mean that every player P_i applies this function to his shares, i.e. computes $(y_i^{(1)}, \ldots, y_i^{(m')}) = f(x_i^{(1)}, \ldots, x_i^{(m)})$. Note that by applying any linear function to correct d-sharings we get a correct d-sharing of the output. However, by multiplying two correct d-sharings we get a correct $2d$-sharing of the product, i.e. $[a]_d[b]_d = [ab]_{2d}$.

5.2 The Share Protocol

The following (trivial) Share protocol allows an honest dealer P_D to correctly d-share a secret s among the players in $\mathcal{P}'$, while communicating $\mathcal{O}(n\kappa)$ bits. We stress that this protocol does not ensure that the resulting sharing is consistent; a corrupted dealer might distribute totally inconsistent shares. The consistency of sharings must be verified separately.

Protocol $\mathsf{Share}(P_D \in (\mathcal{P} \cup \mathcal{U}), s, d)$**.**

1. P_D chooses a random degree-d polynomial $p(\cdot)$ with $s = p(0)$ and sends $s_i = p(\alpha_i)$ to every $P_i \in \mathcal{P}'$.

[2] Where α_i denotes the unique fixed value assigned to P_i.

5.3 The Reconstruct Protocols

We use two reconstruction protocols: one for private and one for public reconstruction. Both can be either robust or only detectable – depending on the degree of the sharings to be reconstructed.

In the private reconstruction protocol the players simply send their shares to the receiver P_R (a player or a user) who interpolates the secret (if possible).

Protocol $\mathsf{ReconsPriv}(P_R \in (\mathcal{P} \cup \mathcal{U}), d, [s]_d)$**.**

1. Every player $P_i \in \mathcal{P}'$ sends his share s_i of s to P_R.
2. If there exists a degree-d polynomial $p(\cdot)$ such that at least $d + t' + 1$ of the received shares lie on it, then P_R computes the secret $s = p(0)$. Otherwise P_R gets unhappy.

Lemma 3. *For $d < n' - 2t'$, the protocol* ReconsPriv robustly *reconstructs $[s]_d$ towards P_R. For $d < n' - t'$,* ReconsPriv detectably *reconstructs $[s]_d$ towards P_R (i.e., P_R either outputs s or gets unhappy, where the latter only happens when some players are faulty).* ReconsPriv *communicates $\mathcal{O}(n\kappa)$ bits.*

The public reconstruction protocol ReconsPubl takes $T = n' - 2t' = n - 2t = \Omega(n)$ correct d-sharings $[s_1]_d, \dots, [s_T]_d$ and publicly (to all players in $\mathcal{P}'$) outputs the (correct) values $s_1, \dots, s_T$ or fails (with at least one honest player being unhappy). In ReconsPubl we use the idea of [DN07]: first the T sharings $[s_1]_d, \dots, [s_T]_d$ are expanded (using a linear error-correcting code) to n' sharings $[u_1]_d, \dots, [u_{n'}]_d$,[3] each of which is reconstructed towards *one* player in $\mathcal{P}'$ (using ReconsPriv). Then every $P_i \in \mathcal{P}'$ sends his reconstructed value u_i to every other player in $\mathcal{P}'$, who tries to decode (with error correction) the received code word $(u_1, \dots, u_{n'})$ to $s_1, \dots, s_T$. ReconsPubl communicates $\mathcal{O}(n^2\kappa)$ bits to reconstruct $T = \Omega(n)$ sharings.

Protocol $\mathsf{ReconsPubl}(d, [s_1]_d, \dots, [s_T]_d)$**.**

1. For every $j = 1, \dots, n'$ the players in $\mathcal{P}'$ (locally) compute $[u_j]_d$ as:
$$[u_j]_d = [s_1]_d + [s_2]_d\beta_j + [s_3]_d\beta_j^2 + \ldots + [s_T]_d\beta_j^{T-1}$$
2. For every $P_i \in \mathcal{P}'$, ReconsPriv is invoked to reconstruct $[u_i]_d$ towards P_i.
3. Every $P_i \in \mathcal{P}'$ sends u_i (or $\perp$ if unhappy) to every $P_j \in \mathcal{P}'$.
4. $\forall P_i \in \mathcal{P}'$: If P_i received at least $T + t'$ $(T-1)$-consistent values (in the previous step), he computes $s_1, \dots, s_T$ from any T of them. Otherwise he gets unhappy.

Lemma 4. *For $d < n' - 2t'$, the protocol* ReconsPubl robustly *reconstructs $[s_1]_d, \dots, [s_T]_d$ towards all players in $\mathcal{P}'$. For $d < n' - t'$,* ReconsPubl detectably *reconstructs $[s_1]_d, \dots, [s_T]_d$ towards all players in $\mathcal{P}'$ (i.e., every $P_i \in \mathcal{P}'$ either outputs $s_1, \dots, s_T$ or gets unhappy, where the latter only happens when some players are faulty).* ReconsPubl *communicates $\mathcal{O}(n^2\kappa)$ bits.*

[3] for this we interpret $s_1, \dots, s_T$ as coefficients of a degree $T - 1$ polynomial and $u_1, \dots, u_{n'}$ as evaluations of this polynomial at n' fixed positions $\beta_1, \dots, \beta_{n'}$.

6 Preparation Phase

6.1 Overview

The goal of the preparation phase is to generate correct t-sharings of $c_M + c_R + c_I$ secret random triples (a_k, b_k, c_k), such that $c_k = a_k b_k$ for $k = 1, \ldots, c_M + c_R + c_I$. We stress that all resulting sharings must be t-sharings (rather than t'-sharings) among the player set $\mathcal{P}'$.[4]

The preparation phase uses player elimination, i.e. the generation of the triples is divided into t segments of length $\ell = \lceil \frac{c_M + c_R + c_I}{t} \rceil$. In every segment the non-robust protocol GenerateTriples is invoked, which either generates correct triples, or fails with at least one honest player being unhappy.

The generation of the triples follows the approach of [DN07]: First, the players generate random a and b values, both simultaneously shared with degree t (for outputting) and degree t' (for multiplication). Additionally, the players generate random value r, simultaneously shared with degree t and degree $2t'$. Then, they locally compute the $2t'$-sharing $[ab]_{2t'}$ (by every player multiplying his respective shares), publicly reconstruct the difference $[ab]_{2t'} - [r]_{2t'}$ and add it (locally) to $[r]_t$, resulting in $[ab]_t$. Finally, the players output the triple $([a]_t, [b]_t, [ab]_t)$.

Definition 4. *A value x is* (d, d')-shared *among the players $\mathcal{P}'$, denoted as $[x]_{d,d'}$, if x is both d-shared and d'-shared. We denote such a sharing as a* double-sharing*, and the pair of shares held by each player as his* double-share*.*

We (trivially) observe that the sum of correct (d, d')-sharings is a correct (d, d')-sharing of the sum.

6.2 Generating Random Double-Sharings

The following non-robust protocol DoubleShareRandom(d, d') either generates T independent secret random values $r_1, \ldots, r_T$, each independently (d, d')-shared among $\mathcal{P}'$, or fails with at least one honest player being unhappy.

The generation of the random double-sharings employs hyper-invertible matrices: First, every player $P_i \in \mathcal{P}'$ selects and double-shares a random value s_i. Then, the players compute double-sharings of the values r_i, defined as $(r_1, \ldots, r_{n'}) = M(s_1, \ldots, s_{n'})$, where M is a hyper-invertible n'-by-n' matrix. $2t'$ of the resulting double-sharings are reconstructed, each towards a different player, who verify the correctness of the double-sharings (and gets unhappy in case of a fault). The remaining $n' - 2t' = T$ double-sharings are outputted. This procedure guarantees that if all honest players are happy, then at least n' double-sharings are correct (the $n' - t'$ double-sharings inputted by honest players, as well as the t' double-sharings verified by honest players), and due to the hyper-invertibility of M, *all* $2n'$ double-sharings must be correct (the remaining double-sharings can be computed linearly from the good double-sharings).

[4] Remember that as $t \leq n' - 2t'$ (according to Lemma 3 and 4), such sharings can be robustly reconstructed (regardless of the actual player set $\mathcal{P}'$).

Furthermore, the outputted double-sharings are random and unknown to the adversary, as there is a bijective mapping from any T double-sharings inputted by honest players to the outputted double-sharings.

Protocol $\mathsf{DoubleShareRandom}(d, d')$**.**

1. Secret Share: Every $P_i \in \mathcal{P}'$ chooses a random s_i and acts (twice in parallel) as a dealer in Share to distribute the shares among the players in $\mathcal{P}'$, resulting in $[s_i]_{d,d'}$.
2. Apply M: The players in $\mathcal{P}'$ (locally) compute $\big([r_1]_{d,d'}, \ldots, [r_{n'}]_{d,d'}\big) = M\big([s_1]_{d,d'}, \ldots, [s_{n'}]_{d,d'}\big)$. In order to do so, every P_i computes his double-share of each r_j as linear combination of his double-shares of the s_k-values.
3. Check: For $i = T+1, \ldots, n'$, every $P_j \in \mathcal{P}'$ sends his double-share of $[s_i]_{d,d'}$ to P_i, who checks that *all* n' double-shares define a correct double-sharing of some value s_i. More precisely, P_i checks that all d-shares indeed lie on a polynomial $g(\cdot)$ of degree d, and that all d'-shares indeed lie on a polynomial $g'(\cdot)$ of degree d', and that $g(0) = g'(0)$. If any of the checks fails, P_i gets unhappy.
4. Output: The remaining T double-sharings $[r_1]_{d,d'}, \ldots, [r_T]_{d,d'}$ are outputted.

Lemma 5. *If* $\mathsf{DoubleShareRandom}(d, d')$ *succeeds (i.e., all honest players are happy), it outputs $T = n' - 2t'$ correct and random (d, d')-sharings (among $\mathcal{P}'$), unknown to the adversary.* $\mathsf{DoubleShareRandom}$ *communicates $\mathcal{O}(n^2\kappa)$ bits to generate $\Omega(n)$ double-sharings.*

Proof. Correctness: Assume that all honest players remain happy during the protocol. Then for all honest P_i with $i \in \{T+1, \ldots, n'\}$, the sharing of r_i checked by P_i in Step 3 is a correct (d, d')-sharing. As $T = n' - 2t'$, there are at least t' correct sharings of the values r_k. Furthermore, every sharing of an s_i distributed by an honest P_i in Step 1 is a correct (d, d')-sharing. Thus there are at least $n' - t'$ correct sharings of the values s_k. Given these (at least) n' correct (d, d')-sharings, the sharings of *all* other values s_k and r_k can be computed linearly. As a linear combination of a correct (d, d')-sharing is again a (d, d')-sharing, it follows that all values $s_1, \ldots, s_{n'}, r_1, \ldots, r_{n'}$ are correctly (d, d')-shared.

Privacy: The adversary knows (at most) t' of the input sharings s_k (those provided by corrupted players), and t' of the output sharings r_k (with $k > T$, those reconstructed towards corrupted players). When fixing these $2t'$ sharings, then there exists a bijective mapping between any other (honest) T input sharings and the first T output sharings (Lemma 2), hence the sharings $[r_1]_{d,d'}, \ldots, [r_T]_{d,d'}$ are uniformly at random, unknown to the adversary.

Communication: The stated communication can easily be verified by inspecting the protocol. □

6.3 Generating Random Triples

Now we present the non-robust protocol $\mathsf{GenerateTriples}$ that either generates $T = n' - 2t'$ correctly t-shared (a, b, c)-triples, or fails (with at least one honest player being unhappy). The idea of the protocol GenerateTriples is the following: First DoubleShareRandom is invoked 3 times to generated the random double-sharings $[a_1]_{t,t'}, \ldots, [a_T]_{t,t'}$, $[b_1]_{t,t'}, \ldots, [b_T]_{t,t'}$, and $[r_1]_{t,2t'}, \ldots, [r_T]_{t,2t'}$, respectively. Then for every pair a_k, b_k, a t-sharing of the product $c_k = a_k b_k$ is computed by reducing the locally computed $2t'$-sharing $[c_k]_{2t'} = [a_k]_{t'}[b_k]_{t'}$ to a t-sharing $[c_k]_t$ using the t-sharing $[r_k]_t$ and the $2t'$-sharing $[r_k]_{2t'}$ of the random value r_k.

Protocol GenerateTriples**.**

1. Generate Double-Sharings: Invoke DoubleShareRandom three times in parallel to generate the double-sharings $[a_1]_{t,t'}, \ldots, [a_T]_{t,t'}$, $[b_1]_{t,t'}, \ldots, [b_T]_{t,t'}$, and $[r_1]_{t,2t'}, \ldots, [r_T]_{t,2t'}$.
2. Multiply:
   - 2.1 For $k = 1, \ldots, T$, the players in $\mathcal{P}'$ compute (locally) the $2t'$-sharing $[c_k]_{2t'}$ of $c_k = a_k b_k$ as $[c_k]_{2t'} = [a_k]_{t'}[b_k]_{t'}$ (by every player computing the product of his shares).
   - 2.2 For $k = 1, \ldots, T$, the players in $\mathcal{P}'$ compute (locally) a $2t'$-sharing of the difference $[d_k]_{2t'} = [c_k]_{2t'} - [r_k]_{2t'}$
   - 2.3 Invoke ReconsPubl $(\mathcal{R} = \mathcal{P}', d = 2t', [d_1]_{2t'}, \ldots, [d_T]_{2t'})$ to reconstruct $d_1, \ldots, d_T$ towards every player in $\mathcal{P}'$.
   - 2.4 For $k = 1, \ldots, T$, the players in $\mathcal{P}'$ compute (locally) the t-sharing $[c_k]_t = [r_k]_t + [d_k]_0$, where $[d_k]_0$ denotes the constant sharing $[d_k]_0 = (d_k, \ldots, d_k)$.
3. Output: The t-shared triples $\big([a_1]_t, [b_1]_t, [c_1]_t\big), \ldots, \big([a_T]_t, [b_T]_t, [c_T]_t\big)$ are outputted.

Lemma 6. *If* GenerateTriples *succeeds (i.e., all honest players are happy), it outputs independent random t-sharings of $T = \Omega(n)$ random triples $\big(a_1, b_1, c_1\big), \ldots, \big(a_T, b_T, c_T\big)$ with a_k, b_k independent uniform random values and $c_k = a_k b_k$ for $k = 1, \ldots, T$.* GenerateTriples *communicates $\mathcal{O}(n^2\kappa)$ bits.*

Proof. The security of GenerateTriples follows directly from the security of DoubleShareRandom. □

6.4 Preparation Phase — Main Protocol

The following protocol PreparationPhase divides the generation of the $c_M + c_R + c_I$ triples into t segments of length $\ell = \lceil \frac{c_M + c_R + c_I}{t} \rceil$. In each segment the triples are generated invoking the non-robust protocol GenerateTriples (as often as necessary), then the players reach agreement on whether or not all players are happy. If yes, they proceed to the next segment. Otherwise, a pair of players is identified in FaultLocalization, excluded from the actual player set $\mathcal{P}'$ and the segment is repeated (with the new $\mathcal{P}'$ and all players setting their happy-bit to happy).

Protocol PreparationPhase.
For each segment $k = 1, \ldots, t$ do:

0. Every $P_i \in \mathcal{P}'$ sets his happy-bit to happy.
1. Triple Generation: Invoke GenerateTriples $\lceil \frac{\ell}{T} \rceil$ times in parallel.
2. Fault Detection: Reach agreement whether or not at least one player is unhappy:
   - 2.1 Every $P_i \in \mathcal{P}'$ sends his happy-bit to every $P_j \in \mathcal{P}'$, who gets unhappy if at least one P_i claims to be unhappy.
   - 2.2 The players in $\mathcal{P}'$ run a consensus protocol on their respective happy-bits. If the consensus outputs "happy", then the generated triples are outputted and the segment is finished. Otherwise, the following Fault-Localization step is executed.
3. Fault Localization: Localize $E \subseteq \mathcal{P}'$ with $|E| = 2$ and at least one player in E being corrupted:
   - 3.0 Denote the player $P_r \in \mathcal{P}'$ with the smallest index r as the referee.[5]
   - 3.1 Every $P_i \in \mathcal{P}'$ sends everything he received and all random values he chose during the computation of the actual segment (including fault detection) to P_r.
   - 3.2 Given the values received in Step 3.1, P_r can reproduce every message that should have been sent (by applying the respective protocol instructions of the sender), and compare it with the value that the recipient claims to have received. Then P_r broadcasts (l, i, j, x, x'), where l is the index of a message where P_i should have sent x to P_j, but P_j claims to have received $x' \neq x$.
   - 3.3 The accused players broadcast whether they agree with P_r. If P_i disagrees, set $E = \{P_r, P_i\}$, if P_j disagrees, set $E = \{P_r, P_j\}$, otherwise set $E = \{P_i, P_j\}$.
4. Player Elimination: Set $\mathcal{P}' \leftarrow \mathcal{P}' \setminus E$, $n' \leftarrow n' - 2$, $t' \leftarrow t' - 1$, and repeat the segment.

Lemma 7. *The protocol* PreparationPhase *generates independent random t-sharings of $c_M + c_R + c_I$ secret triples (a_k, b_k, c_k) with a_k, b_k independent uniform random values and $c_k = a_k b_k$ for $k = 1, \ldots, c_M + c_R + c_I$.* PreparationPhase *communicates $\mathcal{O}\big((c_M + c_R + c_I)n\kappa + n^2\kappa + t\,\mathcal{BA}(\kappa)\big)$ bits, which amounts to $\mathcal{O}\big((c_M + c_R + c_I)n\kappa + n^3\kappa\big)$ bits overall.*

7 Computation Phase

In the computation phase, the circuit is robustly evaluated, whereby all intermediate values are t-shared among the players in $\mathcal{P}'$.

[5] The communication can be balanced by selecting a player who has not yet been referee in a previous segment.

Input gates are realized by reconstructing a pre-shared random value r towards the input-providing user, who then broadcasts the difference of this r and his input.

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 $[r]$ (from the preparation phase) is associated and $[r]_t$ is directly used as outcome of the random gate.

With every multiplication gate, one $([a], [b], [c])$-triple (from the preparation phase) is associated, which is used to compute a sharing of the product at the cost of two public reconstruction. For the sake of efficiency, we evaluate $T/2$ multiplication gates at once (such that we can publicly reconstruct T sharings at once). This of course requires that these multiplication gates do not depend on each other, i.e., that they all have the same multiplicative depth in the circuit.[6]

Output gates involve a (robust) secret reconstruction.

Protocol ComputationPhase**.**
Evaluate the gates of the circuit as follows:

- Input Gate (User U inputs s):
  1. Reconstruct the associated sharing $[r]_t$ towards U with $\mathsf{ReconsPriv}(U, t, [r])$. This is robust because $t < n' - 2t'$.
  2. User U computes and broadcasts the difference $d = s - r$.
  3. Every $P_i \in \mathcal{P}'$ computes his share s_i of s locally as $s_i = d + r_i$.
- Addition/Linear Gate: Every $P_i \in \mathcal{P}'$ applies the linear function on his respective shares.
- Random Gate: Pick the sharing $[r]_t$ associated with the gate.
- Multiplication Gate: Up to $\lfloor T/2 \rfloor$ (where $T = n - 2t$) multiplication gates are processed simultaneously. Denote the factor sharings as $\big([x_1], [y_1]\big), \ldots, \big([x_{T/2}], [y_{T/2}]\big)$, and the associated triples as $([a_1], [b_1], [c_1]), \ldots, ([a_{T/2}], [b_{T/2}], [c_{T/2}])$. The products $[z_1], \ldots, [z_{T/2}]$ are computed as follows:
  1. For $k = 1, \ldots, T/2$, the players compute $[d_k] = [x_k] - [a_k]$ and $[e_k] = [y_k] - [b_k]$.
  2. Invoke ReconsPubl to publicly reconstruct the T t-sharings $(d_1, e_1), \ldots, (d_{T/2}, e_{T/2})$. Note that this is robust, as $t < n' - 2t'$.
  3. For $k = 1, \ldots, T/2$, the players compute the product sharings $[z_k]_t = [de]_0 + d[b]_t + e[a]_t + [c]_t$, where $[de]_0$ denotes the (implicitly defined) 0-sharing of de.
- Output Gate (output $[s]$ to User U): Invoke $\mathsf{ReconsPriv}(U, t, [s]_t)$.

[6] The multiplicative depth of a gate is the maximum number of multiplication gates on any path from input/random gates to this gate.

Lemma 8. *The protocol* ComputationPhase *perfectly securely evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, given $c_I + c_R + c_M$ pre-shared random multiplication triples, with communicating $\mathcal{O}\big((c_I n + c_M n + c_O n + D_M n^2)\kappa + c_I\,\mathcal{BA}(\kappa)\big)$ bits, where D_M denotes the multiplicative depth of the circuit.*

Theorem 1. *The MPC protocol consisting of* PreparationPhase *and* ComputationPhase *evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, with communicating $\mathcal{O}\big((c_I n + c_R n + c_M n + c_O n + D_M n^2)\kappa + (c_I + n)\,\mathcal{BA}(\kappa)\big)$ bits, which amounts to $\mathcal{O}\big((c_I n^2 + c_R n + c_M n + c_O n + D_M n^2)\kappa + n^3\kappa\big)$ bits, where D_M denotes the multiplicative depth of the circuit. The protocol is perfectly secure against an active adversary corrupting $t < n/3$ players.*

The communication complexity for giving input can be improved from $\mathcal{O}(n^2\kappa)$ per input to $\mathcal{O}(n\kappa)$. Details can be found in Appendix A.

Theorem 2. *The MPC protocol given in Appendix A evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, with communicating $\mathcal{O}\big((c_I n + c_R n + c_M n + c_O n + D_M n^2)\kappa + n\,\mathcal{BA}(\kappa)\big)$ bits, which amounts to $\mathcal{O}\big((c_I n + c_R n + c_M n + c_O n + D_M n^2)\kappa + n^3\kappa\big)$ bits, where D_M denotes the multiplicative depth of the circuit. The protocol is perfectly secure against an active adversary corrupting $t < /n/3$ players.*

8 Conclusions

We have presented a perfectly secure multi-party computation protocol with optimal security ($t < n/3$), which communicates only $\mathcal{O}(n)$ field elements per multiplication.

Compared with the previously most efficient perfectly-secure MPC protocol [HMP00], this is a speedup of $\theta(n^2)$ with the same level of security.

Compared with the previously "most secure" MPC protocol with linear communication complexity [DN07], this improves the security from unconditional to perfect, and at the same time slightly improves the communication overhead (from $\mathcal{O}(n^4\kappa)$ in [DN07] to $\mathcal{O}(n^3\kappa)$ here).

This speed-up was possible due to a new technique, so-called hyper-invertible matrices. Such matrices allow to detectably generate $\Omega(n)$ random sharings at costs $\mathcal{O}(n^2)$, with perfect security (i.e., without any probabilistic checks as used in all previous highly-efficient MPC protocols). We believe that this approach is much more natural than the previous approach with probabilistic checks (for example, [DN07] needs to work in an extension field to keep the error-probability small).

References

[Bea91a] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, 1991.

[Bea91b] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, pages 75–122, 1991.

[BGP92] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Bit optimal distributed consensus. *Computer Science Research*, pages 313–322, 1992. Preliminary version has appeared in Proc. 21st STOC, 1989.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th ACM Symposium on the Theory of Computing (STOC)*, pages 1–10, 1988.

[BH06] Zuzana Beerliova-Trubiniova and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography — TCC 2006*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328. Springer-Verlag, March 2006.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. 20th ACM Symposium on the Theory of Computing (STOC)*, pages 11–19, 1988.

[CDvdG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *Advances in Cryptology — CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 87–119. Springer-Verlag, 1987.

[CW92] Brian A. Coan and Jennifer L. Welch. Modular construction of a Byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, March 1992. Preliminary version has appeared in Proc. 8th PODC, 1989.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Alfred Menezes, editor, *Advances in Cryptology — CRYPTO 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[GHY87] Zvi Galil, Stuart Haber, and Moti Yung. Cryptographic computation: Secure fault-tolerant protocols and the public-key model. In *Advances in Cryptology — CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 135–155. Springer-Verlag, 1987.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proc. 19th ACM Symposium on the Theory of Computing (STOC)*, pages 218–229, 1987.

[HMP00] Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multiparty computation. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 143–161. Springer-Verlag, December 2000.

[HN06] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *Advances in Cryptology — CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482. Springer-Verlag, August 2006.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proc. 21st ACM Symposium on the Theory of Computing (STOC)*, pages 73–85, 1989.

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

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 160–164. IEEE, 1982.

Appendix

A Totally Linear Protocol

To construct a totally linear MPC protocol we propose a more efficient input protocol. For the sake of simpler presentation we assume that all inputs are given at the beginning of the computation stage.

We first present the input protocol LinearInput that allows a set of dealers $D \subset \mathcal{P} \cup \mathcal{U}$ each having T inputs to (robustly) share these inputs among the players in $\mathcal{P}'$ (using pre-computed t-sharings of random values). If there is a user with more than T inputs, he plays a role of more dealers.

Protocol LinearInput **(every $D_k \in D$ having inputs $s_k^{(1)}, \ldots, s_k^{(T)}$ with associated random t-sharings $[r_k^{(1)}]_t, \ldots, [r_k^{(T)}]_t$).**

1. Reconstruct: For every $D_k \in D$ and every $l = 1, \ldots, T$ invoke ReconsPriv$(D_k, [r_k^{(l)}]_t)$ to reconstruct the secret random value $r_k^{(l)}$ towards D_k.
2. Compute Difference: Every $D_k \in D$ computes for every $l = 1, \ldots, T$ the difference $d_k^{(l)} = s_k^{(l)} - r_k^{(l)}$.
3. Broadcast: Invoke Broadcast to let every dealer $D_k \in D$ broadcast (towards the players in $\mathcal{P}'$) the T computed differences $d_k^{(1)}, \ldots, d_k^{(T)}$.
4. Compute Locally and Output: For every $D_k \in D$ and every $l = 1, \ldots, T$ the players in $\mathcal{P}'$ (locally) compute the sharing of the input $s_k^{(l)}$ as $[s_k^{(l)}]_t = [d_k^{(l)}]_0 + [r_k^{(l)}]_t$.

The robust protocol Broadcast is constructed in three steps.

We first present a non-robust broadcast protocol for $\mathcal{P}'$ PE − Broadcast.

Note, that broadcasting a value can be interpreted as sharing this value with degree zero, thus checking whether every player distributed his value consistently is the same as checking the correctness of sharings with degree zero, which we can easily do applying HIM.

Protocol PE − Broadcast**(every $P_i \in \mathcal{P}'$ has input x_i).**

1. Distribute Values: Every P_i shares his input with Share $(P_i, x_i, d = 0)$, i.e. sends x_i to every $P_j \in \mathcal{P}'$. Resulting in n' (supposed) 0-sharings

$$[x_1]_0, \ldots, [x_{n'}]_0$$

2. Apply HIM M: The players in $\mathcal{P}'$ compute locally the 0-sharings $[\widehat{x}_1]_0, \ldots, [\widehat{x}_{n'}]_0$ as

$$([\widehat{x}_1]_0, \ldots, [\widehat{x}_{n'}]_0) = M([x_1]_0, \ldots, [x_{n'}]_0)$$

3. Check: Every $P_i \in \mathcal{P}'$ checks the correctness of $[\widehat{x}_i]_0$. For this every $P_j \in \mathcal{P}'$ sends his share of $\widehat{x}_i$ to P_i. If the values received by P_i are not 0-consistent (equal), P_i gets unhappy.
4. Output: Every $P_j \in \mathcal{P}'$ outputs the values received in Step 1.)

Now we construct a robust broadcast protocol for $\mathcal{P}'$ $\mathsf{BroadcastFor}\mathcal{P}'$ using $\mathsf{PE-Broadcast}$, player elimination and segmentation. $\mathsf{BroadcastFor}\mathcal{P}'$ allows the players in $\mathcal{P}'$, each holding ℓ values $x_i^{(1)}, \ldots, x_i^{(\ell)}$ to broadcast this values among the players in $\mathcal{P}'$.

Protocol $\mathsf{BroadcastFor}\mathcal{P}'$**.**
For each segment $k = 1, \ldots, t$ (of length $\ell' = \lceil \frac{\ell}{t} \rceil$) do:

0. Every $P_i \in \mathcal{P}'$ sets his happy-bit to happy.
1. PE-Broadcast: Invoke $\mathsf{PE-Broadcast}$ $\ell' = \lceil \frac{\ell}{t} \rceil$ times in parallel, i.e. for $l = 1, \ldots, \ell'$ invoke $\mathsf{PE-Broadcast}$ to let every $P_i \in \mathcal{P}'$ broadcast his input $x_i = x_i^{(l+(k-1)\ell')}$.
2. Fault Detection: Reach agreement whether or not at least one player is unhappy:
   2.1 Every $P_i \in \mathcal{P}'$ sends his happy-bit to every $P_j \in \mathcal{P}'$, who gets unhappy if at least one P_i claims to be unhappy.
   2.2 The players in $\mathcal{P}'$ run a consensus protocol on their respective happy-bits. If the consensus outputs "happy", then the generated triples are outputted and the segment is finished. Otherwise, the following Fault-Localization step is executed.
3. Fault Localization: Localize $E \subseteq \mathcal{P}'$ with $|E| = 2$ and at least one player in E being corrupted:
   3.0 Denote the player $P_r \in \mathcal{P}'$ with the smallest index r as the referee.[7]
   3.1 Every $P_i \in \mathcal{P}'$ sends everything he received and all random values he chose during the computation of the actual segment (including fault detection) to P_r.
   3.2 Given the values received in Step 3.1, P_r can reproduce every message that should have been sent (by applying the respective protocol instructions of the sender), and compare it with the value that the recipient claims to have received. Then P_r broadcasts (l, i, j, x, x'), where l is the index of a message where P_i should have sent x to P_j, but P_j claims to have received $x' \neq x$.

[7] The communication can be balanced by selecting a player who has not yet been referee in a previous segment.

3.3 The accused players broadcast whether they agree with P_r. If P_i disagrees, set $E = \{P_r, P_i\}$, if P_j disagrees, set $E = \{P_r, P_j\}$, otherwise set $E = \{P_i, P_j\}$.

4. Player Elimination: Set $\mathcal{P}' \leftarrow \mathcal{P}' \setminus E$, $n' \leftarrow n' - 2$, $t' \leftarrow t' - 1$, and repeat the segment.

Finally we present the protocol Broadcast that enables a set of dealers D (players or users), each holding T values to robustly broadcast this values, among the players in $\mathcal{P}'$.

The idea of the protocol is to let every dealer expand his T values to n' values (using an error-correcting code tolerating t' errors) and to send each of these values to one player in $\mathcal{P}'$. Then the players in $\mathcal{P}'$ invoke BroadcastFor$\mathcal{P}'$ to broadcast the received values and final (locally) compute the original values from the broadcasted values using error-correction.

Protocol Broadcast(**every dealer** D_k **holding** $a_k^{(0)}, \ldots, a_k^{(T-1)}$)**.**

1. Expand and Distribute: For every dealer D_k denote the polynomial defined by the values $a_k^{(0)}, \ldots, a_k^{(T-1)}$ as $p_k(x)$, i.e.

$$p_k(x) = a_k^{(0)} + a_k^{(1)} x + \ldots + a_k^{(T-1)} x^{T-1}$$

. The dealer D_k computes for every player $P_i \in \mathcal{P}'$ the point $p_k(\alpha_i)$ and sends it to P_i.

2. Broadcast: The players in $\mathcal{P}'$ invoke BroadcastFor$\mathcal{P}'$ with P_i having input $p_1(\alpha_i), \ldots, p_{|D|}(\alpha_i)$.
3. Compute and Output: For every dealer D_k every $P_i \in \mathcal{P}'$ locally computes the values $a_k^{(0)}, \ldots, a_k^{(T-1)}$ from the broadcasted values $p_k(\alpha_1), \ldots, p_k(\alpha_{n'})$ (using error-correction).