Scalable Multiparty Computation with Nearly Optimal Work and Resilience

Ivan Damgård[1], Yuval Ishai[2⋆], Mikkel Krøigaard[1], Jesper Buus Nielsen[1], and Adam Smith[3]

[1] University of Aarhus, Denmark. Email: {ivan,mk,buus}@daimi.au.dk
[2] Technion and UCLA. Email: yuvali@cs.technion.ac.il
[3] Pennsylvania State University, USA. Email: asmith@cse.psu.edu

Abstract. We present the first general protocol for secure multiparty computation in which the *total* amount of work required by n players to compute a function f grows only polylogarithmically with n (ignoring an additive term that depends on n but not on the complexity of f). Moreover, the protocol is also nearly optimal in terms of resilience, providing computational security against an active, adaptive adversary corrupting a $(1/2 - \epsilon)$ fraction of the players, for an arbitrary $\epsilon > 0$.

1 Introduction

Secure multiparty computation (MPC) allows n mutually distrustful players to perform a joint computation without compromising the privacy of their inputs or the correctness of the outputs. Following the seminal works of the 1980s which established the *feasibility* of MPC [35, 22, 4, 9], significant efforts have been invested into studying the *complexity* of MPC. When studying how well MPC scales to a large network, the most relevant goal minimizing the growth of complexity with the number of players, n. This is motivated not only by distributed computations involving inputs from many participants, but also by scenarios in which a (possibly small) number of "clients" wish to distribute a joint computation between a large number of untrusted "servers".

The above question has been the subject of a large body of work [19–21, 26, 24, 29, 11, 14, 27, 2, 12, 28, 15, 3]. In most of these works, the improvement over the previous state of the art consisted of either reducing the multiplicative overhead depending on n (say, from cubic to quadratic) or, alternatively, maintaining the same asymptotic overhead while increasing the fraction of players that can be corrupted (say, from one third to one half).

The current work completes this long sequence of works, at least from a crude asymptotic point of view: We present a general MPC protocol which is simultaneously optimal, up to lower-order terms, with respect to both efficiency and resilience. More concretely, our protocol allows n players to evaluate an

⋆ Supported in part by ISF grant 1310/06, BSF grant 2004361, and NSF grants 0430254, 0456717, 0627781.

arbitrary circuit C on their joint inputs, with the following efficiency and security features.

COMPUTATION. The *total* amount of time spent by all players throughout the execution of the protocol is $\mathrm{poly}(k, \log n, \log |C|) \cdot |C| + \mathrm{poly}(k, n)$, where $|C|$ is the size of C and k is a cryptographic security parameter. Thus, the protocol is strongly *scalable* in the sense that the amount of work involving each player (amortized over the computation of a large circuit C) vanishes with the number of players. We write the above complexity as $\widetilde{\mathcal{O}}(|C|)$, hiding the low-order multiplicative $\mathrm{poly}(k, \log n, \log |C|)$ and additive $\mathrm{poly}(k, n)$ terms.[1]

COMMUNICATION. As follows from the bound on computation, the total number of bits communicated by all n players is also bounded by $\widetilde{\mathcal{O}}(|C|)$. This holds even in a communication model that includes only point-to-point channels and no broadcast. Barring a major breakthrough in the theory of secure computation, this is essentially the best one could hope for. However, unlike the case of computation, here a significant improvement cannot be completely ruled out.

RESILIENCE. Our protocol is computationally UC-secure [6] against an active, adaptive adversary corrupting at most a $(1/2 - \epsilon)$ fraction of the players, for an arbitrarily small constant $\epsilon > 0$. This parameter too is essentially optimal since robust protocols that guarantee output delivery require honest majority.

ROUNDS. The round complexity of the basic version of the protocol is $\mathrm{poly}(k, n)$. Using a pseudorandom generator that is "computationally simple" (e.g., computable in NC^1), the protocol can be modified to run in a constant number of rounds. Such a pseudorandom generator is implied by most standard concrete intractability assumptions in cryptography [1]. Unlike our main protocol, the constant-round variant only applies to functionalities that deliver outputs to a small (say, constant) number of players. Alternatively, it may apply to arbitrary functionalities but provide the weaker guarantee of "security with abort".

The most efficient previous MPC protocols from the literature [12, 28, 15, 3] have communication complexity of $\widetilde{\mathcal{O}}(n \cdot |C|)$, and no better complexity even in the semi-honest model. The protocols of Damgård and Nielsen [15] and Beerliova and Hirt [3] achieve this complexity with unconditional security. It should be noted that the protocol of Damgård and Ishai [12] has a variant that matches the asymptotic complexity of our protocol. However, this variant applies only to functionalities that receive inputs from and distribute outputs to a small number of players. Furthermore, it only tolerates a small fraction of corrupted players.

Techniques. Our protocol borrows ideas and techniques from several previous works in the area, especially [12, 28, 15, 3]. Similarly to [12], we combine the

[1] Such terms are to some extent unavoidable, and have also been ignored in previous works along this line. Note that the additive term becomes insignificant when considering complex computations (or even simple computations on large inputs), whereas the multiplicative term can be viewed as polylogarithmic under exponential security assumptions. The question of minimizing these lower order terms, which are significant in practice, is left for further study.

efficient secret sharing scheme of Franklin and Yung [20] with Yao's garbled circuit technique [35]. The scheme of Franklin and Yung generalizes Shamir's secret sharing scheme [33] to efficiently distribute a whole block of ℓ secrets, at the price of decreasing the security threshold. Yao's technique can be used to transform the circuit C into an equivalent, but very shallow, *randomized* circuit $C_{\mathtt{Yao}}$ of comparable size. The latter, in turn, can be evaluated "in parallel" on blocks of inputs and randomness that are secret-shared using the scheme of [20].

The main efficiency bottleneck in [12] is the need to distribute the blocks of randomness that serve as inputs for $C_{\mathtt{Yao}}$. The difficulty stems from the fact that these blocks should be arranged in a way that reflects the structure of C. That is, each random secret bit may appear in several blocks according to a pattern determined by C. These blocks were generated in [12] by adding contributions from different players, which is not efficient enough for our purposes. More efficient methods for distributing many random secrets were used in [28, 15, 3]. However, while these methods can be applied to cheaply generate many blocks of the same pattern, the blocks we need to generate may have arbitrary patterns.

To get around this difficulty, we use a pseudorandom function (PRF) for reducing the problem of generating blocks of an arbitrary structure to the problem of generating independent random blocks. This is done by applying the PRF (with a key that is secret-shared between the servers) to a sequence of public labels that specifies the required replication pattern, where identical labels are used to generate copies of the same secret.

Another efficiency bottleneck we need to address is the cost of delivering the outputs. If many players should receive an output, we cannot afford to send the entire output of $C_{\mathtt{Yao}}$ to these players. To get around this difficulty, we propose a procedure for securely distributing the decoding process between the players without incurring too much extra work. This also has the desirable effect of dividing the work equally between the players.

Finally, to boost the fractional security threshold of our protocol from a small constant δ to a nearly optimal constant of $(1/2-\epsilon)$, we adapt to our setting a technique that was introduced by Bracha [5] in the context of Byzantine Agreement. The idea is to compose our original protocol $\pi_{\mathtt{out}}$, which is efficient but has a low security threshold $(t < n/c)$, with another known protocol $\pi_{\mathtt{in}}$, which is inefficient but has an optimal security threshold $(t < n/2)$ in a way that will give us essentially the best of both worlds. The composition uses $\pi_{\mathtt{in}}$ to distribute the local computations of each player in $\pi_{\mathtt{out}}$ among a corresponding committee that includes a constant number of players. The committees are chosen such that any set including at most $1/2-\epsilon$ of the players forms a majority in less than δn of the committees. Bracha's technique has been recently applied in the cryptographic contexts of secure message transmission [17] and establishing a network of OT channels [23]. We extend the generality of the technique by applying it as a method for boosting the security threshold of general MPC protocols with only a minor loss of efficiency.

2 Preliminaries

In this section we present some useful conventions.

Client-server model. Similarly to previous works, it will be convenient to slightly refine the usual MPC model as follows. We assume that the set of players consists of a set of *input clients* that hold the inputs to the desired computation, a set of n *servers*, $\mathcal{S} = \{S_1, \ldots, S_n\}$, that execute the computation, and a set of *output clients* that receive outputs. Since one player can play the role of both client(s) and a server, this is a generalization of the standard model. The number of clients is assumed to be at most linear in n, which allows us to ignore the exact number of clients when analyzing the asymptotic complexity of our protocols.

Complexity conventions. We will represent the functionality which we want to securely realize by a boolean circuit C with bounded fan-in, and denote by $|C|$ the number of gates in C. We adopt the convention that every input gate in C is labeled by the input client who should provide this input (alternatively, labeled by "random" in the case of a randomized functionality) and every output gate in C is labeled by a name of a single output client who should receive this output. In particular, distributing an output to several clients must be "paid for" by having a larger circuit. Without this rule, we could be asked to distribute the entire output $C(x)$ to all output clients, forcing the communication complexity to be more than we can afford. We denote by k a cryptographic security parameter, which is thought of as being much smaller than n (e.g., $k = O(n^\epsilon)$ for a small constant $\epsilon > 0$, or even $k = \text{polylog}(n)$).

Security conventions. By default, when we say that a protocol is "secure" we mean that it realizes in the UC model [6] the corresponding functionality with computational t-security against an *active* (malicious) and *adaptive* adversary, using synchronous communication over secure point-to-point secure channels. Here t denotes the maximal number of corrupted server; there is no restriction on the number of corrupted clients. (The threshold t will typically be of the form δn for some constant $0 < \delta < 1/2$.) The results can be extended to require only authenticated channels assuming the existence of public key encryption (even for adaptive corruptions, cf. [7]). We will sometimes make the simplifying assumption that outputs do not need to be kept private. This is formally captured by letting the ideal functionality leak $C(x)$ to the adversary. Privacy of outputs can be achieved in a standard way by having the functionality mask the output of each client with a corresponding input string picked randomly by this client.

3 Building Blocks

In this section, we will present some subprotocols that will later be put together in a protocol implementing a functionality F_{CP}, which allows to evaluate the same circuit in parallel on multiple inputs. We will argue that each subprotocol is *correct*: every secret-shared value that is produced as output is consistently shared, and *private*: the adversary learns nothing about secrets shared by uncorrupted parties. While correctness and privacy alone do not imply UC-security, when combined with standard simulation techniques for honest-majority MPC protocols they will imply that our implementation of F_{CP} is UC-secure.

Packed Secret-Sharing. We use a variant of the packed secret-sharing technique by Franklin and Yung [20]. We fix a finite field $\mathbb{F}$ of size $\mathcal{O}(\log(n)) = \widetilde{\mathcal{O}}(1)$ and share together a vector of field elements from $\mathbb{F}^\ell$, where ℓ is a constant fraction of n. We call $s = (s^1, \dots, s^\ell) \in \mathbb{F}^\ell$ a *block*. Fix a generator α of the multiplicative group of $\mathbb{F}$ and let $\beta = \alpha^{-1}$. We assume that $|\mathbb{F}| > 2n$ such that $\beta^0, \dots, \beta^{c-1}$ and $\alpha^1, \dots, \alpha^n$ are distinct elements. Given $x = (x_0, \dots, x_{c-1}) \in \mathbb{F}^c$, compute the unique polynomial $f(\mathtt{X}) \in \mathbb{F}[\mathtt{X}]$ of degree $\le c-1$ for which $f(\beta^i) = x_i$ for $i = 0, \dots, c-1$, and let $M_{c\to n}(x) = (y_1, \dots, y_n) = (f(\alpha^1), \dots, f(\alpha^n))$. This map is clearly linear, and we use $M_{c\to n}$ to denote both the mapping and its matrix. Let $M_{c\to r}$ consist of the top r rows of $M_{c\to n}$.

Since the mapping consists of a polynomial interpolation followed by a polynomial evaluation, one can use the fast Fourier transform (FFT) to compute the mapping in time $\widetilde{\mathcal{O}}(c) + \widetilde{\mathcal{O}}(n) = \widetilde{\mathcal{O}}(n)$. In [3] it is shown that $M_{c\to n}$ is *hyper-invertible*. A matrix M is hyper-invertible if the following holds: Let R be a subset of the rows, and let M_R denote the sub-matrix of M consisting of rows in R. Likewise, let C be a subset of columns and let M^C denote the sub-matrix consisting of columns in C. Then we require that M_R^C is invertible whenever $|R| = |C| > 0$. Note that from $M_{c\to n}$ being hyper-invertible and computable in $\widetilde{\mathcal{O}}(n)$ time, it follows that all $M_{c\to r}$ are hyper-invertible and computable in $\widetilde{\mathcal{O}}(n)$ time.

Protocol $\textsc{Share}(D, d)$:

1. Input to dealer D: $(s^1, \dots, s^\ell) \in \mathbb{F}^\ell$. Let $M = M_{\ell+t\to n}$, where $t = d - \ell + 1$.
2. D: Sample $r^1, \dots, r^t \in_R \mathbb{F}$, let $(s_1, \dots, s_n) = M(s^1, \dots, s^\ell, r^1, \dots, r^t)$, and send s_i to server S_i, for $i = 1, \dots, n$.

The sharing protocol is given in Protocol $\textsc{Share}(D, d)$. Note that $(s_1, \dots, s_n)$ is just a t-private packed Shamir secret sharing of the secret block $(s^1, \dots, s^\ell)$ using a polynomial of degree $\le d$. We therefore call $(s_1, \dots, s_n)$ a *d-sharing* and write $[s]_d = [s^1, \dots, s^\ell]_d = (s_1, \dots, s_n)$. In general we call a vector $(s_1, \dots, s_n)$ a *consistent d-sharing* (over $\mathcal{S} \subseteq \{S_1, \dots, S_n\}$) if the shares (of the servers in $\mathcal{S}$) are consistent with some d-sharing. For $a \in \mathbb{F}$ we let $a[s]_d = (as_1, \dots, as_n)$ and for $[s]_d = (s_1, \dots, s_n)$ and $[t]_d = (t_1, \dots, t_n)$ we let $[s]_d + [t]_d = (s_1 + t_1, \dots, s_n + t_n)$. Clearly, $a[s]_d + b[t]_d$ is a d-sharing of $as + bt$; We write $[as + bt]_d = a[s]_d + b[t]_d$. We let $[st]_{2d} = (s_1t_1, \dots, s_nt_n)$. This is a $2d$-sharing of the block $st = (s^1t^1, \dots, s^\ell t^\ell)$.

Below, when we instruct a server to check if $y = (y_1, \dots, y_n)$ is d-consistent, it interpolates the polynomial $f(\alpha^i) = y_i$ and checks that the degree is $\le d$. This can be done in $\widetilde{\mathcal{O}}(n)$ time using FFT.

To be able to reconstruct a sharing $[s]_{d_1}$ given t faulty shares, we need that $n \ge d_1 + 1 + 2t$. We will only need to handle up to $d_1 = 2d$, and therefore need $n = 2d + 1 + 2t$. Since $d = \ell + t - 1$ we need $n \ge 4t + 2\ell - 1$ servers. To get the efficiency we are after, we will need that ℓ, $n - 4t$ and t are $\Theta(n)$. Concretely we could choose, for instance, $t = n/8, \ell = n/4$.

Random Monochromatic Blocks. In the following, we will need a secure protocol for the following functionality:

> Functionality MONOCHROM:
> Takes no input.
> Output: a uniformly random sharing $[b]_d$, where the block b is $(0, \ldots, 0)$ with probability $\frac{1}{2}$ and $(1, \ldots, 1)$ with probability $\frac{1}{2}$.

We only call the functionality k times in total, so the complexity of its implementation does not matter for the amortized complexity of our final protocol.

Semi-Robust VSS. To get a verifiable secret sharing protocol guaranteeing that the shares are d-consistent we adapt to our setting a VSS from [3].[2] Here and in the following subprotocols, several non-trivial modifications have to be made, however, due to our use of packed secret sharing, and also because directly using the protocol from [3] would lead to a higher complexity than we can afford.

> Protocol SEMIROBUSTSHARE(d):
>
> 1. For each dealer D and each group of blocks $(x_1, \ldots, x_{n-3t}) \in (\mathbb{F}^\ell)^{n-3t}$ to be shared by D, the servers run the following in parallel:
>   (a) D: Pick t uniformly random blocks $x_{n-3t+1}, \ldots, x_{n-2t}$ and deal $[x_i]_d$ for $i = 1, \ldots, n-2t$, using SHARE(D, d).
>   (b) All servers: Compute $([y_1]_d, \ldots, [y_n]_d) = M([x_1]_d, \ldots, [x_{n-2t}]_d)$ by locally applying M to the shares.
>   (c) Each S_j: Send the share y_i^j of $[y_i]_d$ to S_i.
>   (d) D: send the shares y_i^j of $[y_i]_d$ to S_i.
> 2. Now conflicts between the sent shares are reported. Let $\mathcal{C}$ be a set of subsets of $\mathcal{S}$, initialized to $\mathcal{C} := \emptyset$. Each S_i runs the following in parallel:
>   (a) If S_i sees that D for some group sent shares which are not d-consistent, then S_i broadcasts $(\texttt{J'accuse}, D)$, and all servers add $\{D, S_i\}$ to $\mathcal{C}$.
>   (b) Otherwise, if S_i sees that there is some group dealt by D and some S_j which for this group sent y_i^j and D sent $y_i^{j\prime} \neq y_i^j$, then S_i broadcasts $(\texttt{J'accuse}, D, S_j, g, y_i^{j\prime}, y_i^j)$ for all such S_j, where g identifies the group for which a conflict is claimed. At most one conflict is reported for each pair (D, S_j).
>   (c) If D sees that $y_i^{j\prime}$ is not the share it sent to S_j for group g, then D broadcasts $(\texttt{J'accuse}, S_j)$, and all servers add $\{D, S_j\}$ to $\mathcal{C}$.
>   (d) At the same time, if S_i sees that y_i^j is not the share it sent to S_j for group g, then S_i broadcasts $(\texttt{J'accuse}, S_j)$, and all servers add $\{S_i, S_j\}$ to $\mathcal{C}$.
>   (e) If neither D nor S_i broadcast $(\texttt{J'accuse}, S_j)$, they acknowledge to have sent different shares to S_j for group g, so one of them is corrupted. In this case all servers add $\{D, S_i\}$ to $\mathcal{C}$.
> 3. Now the conflicts are removed by *eliminating* some players:
>   (a) As long as there exists $\{S_1, S_2\} \in \mathcal{C}$ such that $\{S_1, S_2\} \subseteq \mathcal{S}'$, let $\mathcal{S}' := \mathcal{S}' \setminus \{S_1, S_2\}$.
>   (b) The protocol outputs the $[x_i]_d$ created by non-eliminated dealers.

[2] This protocol has an advantage over previous subprotocols with similar efficiency, e.g. from [12], in that it has *perfect* (rather than statistical) security. This makes it simpler to analyze its security in the presence of adaptive corruptions.

The protocol uses $M = M_{n-2t\to n}$ to check consistency of sharings. For efficiency, all players that are to act as dealers will deal at the same time. The protocol can be run with all servers acting as dealers. Each dealer D shares a *group* of $n - 3t = \Theta(n)$ blocks, and in fact, D handles a number of such groups in parallel. Details are given in Protocol SemiRobustShare. Note that SemiRobustShare(d) may not allow all dealers to successfully share their blocks, since some can be eliminated during the protocol. We handle this issue later in Protocol RobustShare.

At any point in our protocol, $\mathcal{S}'$ will be the set of servers that still participate. We set $n' = |\mathcal{S}'|$ and $t' = t - e$ will be the maximal number of corrupted servers in $\mathcal{S}'$, where e is the number of pairs eliminated so far.

To argue correctness of the protocol, consider any surviving dealer $D \in \mathcal{S}'$. Clearly D has no conflict with any surviving server, i.e., there is no $\{D, S_i\} \in \mathcal{C}$ with $\{D, S_i\} \subset \mathcal{S}'$. In particular, all $S_i \in \mathcal{S}'$ saw D send only d-consistent sharings. Furthermore, each such S_i saw each $S_j \in \mathcal{S}'$ send the same share as D during the test, or one of $\{D, S_j\}$, $\{S_i, S_j\}$ or $\{D, S_i\}$ would be in $\mathcal{C}$, contradicting that they are all subsets of $\mathcal{S}'$.

Since each elimination step $\mathcal{S}' := \mathcal{S}' \setminus \{S_1, S_2\}$ removes at least one new corrupted server, it follows that at most t honest servers were removed from $\mathcal{S}'$. Therefore there exists $H \subset \mathcal{S}'$ of $n - 2t$ honest servers. Let $([y_i]_d)_{S_i \in H} = M_H([x_1]_d, \ldots, [x_{n-2t}]_d)$. By the way conflicts are removed, all $[y_i]_d$, $S_i \in H$ are d-consistent on $\mathcal{S}'$. Since M_H is invertible, it follows that all $([x_1]_d, \ldots, [x_{n-t}]_d) = M_H^{-1}([y_i]_d)_{S_i \in H}$ are d-consistent on $\mathcal{S}'$.

The efficiency follows from $n - 3t = \Theta(n)$, which implies a complexity of $\widetilde{\mathcal{O}}(\beta n) + \text{poly}(n)$ for sharing β blocks (here poly(n) covers the $\mathcal{O}(n^3)$ broadcasts). Since each block contains $\Theta(n)$ field elements, we get a complexity of $\widetilde{\mathcal{O}}(\phi)$ for sharing ϕ field elements.

As for privacy, let $I = \{1, \ldots, n - 3t\}$ be the indices of the data blocks and let $R = \{n - 3t + 1, \ldots, n - 2t\}$ be the indices of the random blocks. Let $C \subset \{1, \ldots, n\}$, $|C| = t$ denote the corrupted servers. Then $([y_i]_d)_{i \in C} = M_C([x_1]_d, \ldots, [x_{n-2t}]_d) = M_C^I([x_i]_d)_{i \in I} + M_C^R([x_i]_d)_{i \in R}$. Since $|C| = |R|$, M_C^R is invertible. So, for each $([x_i]_d)_{i \in D}$, exactly one choice of random blocks $([x_i]_d)_{i \in R} = (M_C^R)^{-1}(([y_i]_d)_{i \in C} - M_C^I([x_i]_d)_{i \in I})$ are consistent with this data, which implies perfect privacy.

Double Degree VSS. We also use a variant SemiRobustShare(d_1, d_2), where each block x_i is shared both as $[x_i]_{d_1}$ and $[x_i]_{d_2}$ (for $d_1, d_2 \leq 2d$). The protocol executes SemiRobustShare(d_1) and SemiRobustShare(d_2), in parallel, and in Step 2a in SemiRobustShare the servers also accuse D if the d_1-sharing and the d_2-sharing is not of the same value. It is easy to see that this guarantees that all $D \in \mathcal{S}'$ shared the same x_i in all $[x_i]_{d_1}$ and $[x_i]_{d_2}$.

Reconstruction. We use the following procedure for reconstruction towards a server R.

Protocol $\textsc{Reco}(R, d_1)$:

1. The servers hold a sharing $[s]_{d_1}$ which is d_1-consistent over $\mathcal{S}'$ (and $d_1 \leq 2d$). The server R holds a set $\mathcal{C}_i$ of servers it knows are corrupted. Initially $\mathcal{C}_i = \emptyset$.
2. Each $S_i \in \mathcal{S}'$: Send the share s_i to R.
3. R: If the shares s_i are d_1-consistent over $\mathcal{S}' \setminus \mathcal{C}_i$, then compute s by interpolation. Otherwise, use error correction to compute the nearest sharing $[s']_{d_1}$ which is d_1-consistent on $\mathcal{S}' \setminus \mathcal{C}_i$, and compute s from this sharing using interpolation. Furthermore, add all S_j for which $s'_j \neq s_j$ to $\mathcal{C}_i$.

Computing the secret by interpolation can be done in time $\widetilde{\mathcal{O}}(n)$. For each invocation of the poly(n)-time error correction, at least one corrupted server is removed from $\mathcal{C}_i$, bounding the number of invocations by t. Therefore the complexity for reconstructing β blocks is $\widetilde{\mathcal{O}}(\beta n) + \text{poly}(n) = \widetilde{\mathcal{O}}(\phi)$, where ϕ is the number of field elements reconstructed.

At the time of reconstruction, some e eliminations have been performed to reach $\mathcal{S}'$. For the error correction to be possible, we need that $n' \geq d_1 + 1 + 2t'$. In the worst case one honest party is removed per elimination. So we can assume that $n' = n - 2e$ and $t' = t - e$. So, it is sufficient that $n \geq d_1 + 1 + 2t$, which follows from $n \geq 2d + 1 + 2t$ and $d_1 \leq 2d$.

Robust VSS. Protocol $\textsc{RobustShare}$ guarantees that all dealers can secret share their blocks, and can be used by input clients to share their inputs. Privacy follows as for $\textsc{SemiRobustShare}$. Correctness is immediate. Efficiency follows directly from $n - 4t = \mathcal{O}(n)$, which guarantees a complexity of $\widetilde{\mathcal{O}}(\phi)$ for sharing ϕ field elements.

Protocol $\textsc{RobustShare}(d)$:

1. Each dealer D shares groups of $n - 4t$ blocks $x_1, \ldots, x_{n-4t}$. For each group it picks t random blocks $x_{n-4t+1}, \ldots, x_{n-3t}$, computes n blocks $(y_1, \ldots, y_n) = M(x_1, \ldots, x_{n-3t})$ and sends y_i to S_i. Here $M = M_{n-4t \to n}$.
2. The parties run $\textsc{SemiRobustShare}(d)$, and each S_i shares y_i.[a] This gives a reduced server set $\mathcal{S}'$ and a d-consistent sharing $[y'_i]_d$ for each $S_i \in \mathcal{S}'$.
3. The parties run $\textsc{Reco}(D, d)$ on $[y'_i]_d$ for $S_i \in \mathcal{S}'$ to let D learn y'_i for $S_i \in \mathcal{S}'$.
4. D picks $H \subset \mathcal{S}'$ for which $|H| = n - 3t$ and $y'_i = y_i$ for $S_i \in H$, and broadcasts H, the indices of these parties.[b]
5. All parties compute $([x_1]_d, \ldots, [x_{n-3t}]_d) = M_H^{-1}([y_i]_d)_{i \in H}$. Output is $[x_1]_d, \ldots, [x_{n-4t}]_d$.

[a] In the main protocol, many copies of $\textsc{RobustShare}$ will be run in parallel, and S_i can handle the y_i's from all copies in parallel, putting them in groups of size $n - 3t$.

[b] $\mathcal{S}'$ has size at least $n - 2t$, and at most the t corrupted parties did not share the right value. When many copies of $\textsc{RobustShare}(d)$ are run in parallel, only one subset H is broadcast, which works for all copies.

Sharing Bits. We also use a variant RobustShareBits(d), where the parties are required to input bits, and where this is checked. First RobustShare(d) is run to do the actual sharing. Then for each shared block $[x^1, \ldots, x^\ell]$ the parties compute $[y^1, \ldots, y^\ell]_{2d} = ([1, \ldots, 1]_d - [x^1, \ldots, x^\ell])[x^1, \ldots, x^\ell] = [(1 - x^1)x^1, \ldots, (1 - x^\ell)x^\ell]$. They generate $[1, \ldots, 1]_d$ by all picking the share 1. Note that $[y]_{2d} = [0, \ldots, 0]_{2d}$ if and only if all x^i were in $\{0, 1\}$.

For each dealer D all $[y]_{2d}$ are checked in parallel, in groups of $n' - 2t'$. For each group $[y_1]_{2d}, \ldots, [y_{n'-2t'}]_{2d}$, D makes sharings $[y_{n'-2t'+1}]_{2d}, \ldots, [y_{n'-t'}]_{2d}$ of $y_i = (0, \ldots, 0)$, using RobustShare($2d$). Then all parties compute $([x_1]_{2d}, \ldots, [x_{n'}]_{2d}) = M([y_1]_{2d}, \ldots, [y_{n'-t'}]_{2d})$, where $M = M_{n'-t' \to n}$. Then each $[x_i]_{2d}$ is reconstructed towards S_i. If all $x_i = (0, \ldots, 0)$, then S_i broadcasts `ok`. Otherwise S_i for each cheating D broadcasts (`J'accuse`, D, g), where D identifies the dealer and g identifies a group $([x_1]_{2d}, \ldots, [x_{n'}]_{2d})$ in which it is claimed that $x_i \neq (0, \ldots, 0)$. Then the servers publicly reconstruct $[x_i]_d$ (i.e., reconstruct it towards each server using Reco($2d, \cdot$)). If $x_i = (0, \ldots, 0)$, then S_i is removed from $\mathcal{S}'$; otherwise, D is removed from $\mathcal{S}'$, and the honest servers output the all-zero set of shares.

Let H denote the indices of $n' - t'$ honest servers. Then $([x_i]_{2d})_{i \in H} = M_H([y_1]_{2d}, \ldots, [y_{n'-t'}]_{2d})$. So, if $x_i = (0, \ldots, 0)$ for $i \in H$, it follows from $([y_1]_{2d}, \ldots, [y_{n'-t'}]_{2d}) = M_H^{-1}([x_i]_{2d})_{i \in H}$ that all $y_i = (0, \ldots, 0)$. Therefore D will pass the test if and only if it shared only bits. The privacy follows using the same argument as in the privacy analysis of Protocol SemiRobustShare. The efficiency follows from $\Theta(n)$ blocks being handled in each group, and the number of broadcasts and public reconstructions being independent of the number of blocks being checked.

Resharing with a Different Degree. We need a protocol which given a d_1-consistent sharing $[x]_{d_1}$ produces a d_2-consistent sharing $[x]_{d_2}$ (here $d_1, d_2 \leq 2d$). For efficiency all servers R act as resharer, each handling a number of groups of $n' - 2t' = \Theta(n)$ blocks. The protocol is not required to keep the blocks x secret. We first present a version in which some R might fail.

Protocol SemiRobustReshare(d_1, d_2):

- For each $R \in \mathcal{S}'$ and each group $[x_1]_{d_1}, \ldots, [x_{n'-2t'}]_{d_1}$ (all sharings are d_1-consistent on $\mathcal{S}'$) to be reshared by R, the servers proceed as follows:
- Run Reco(R, d_1) on $[x_1]_{d_1}, \ldots, [x_{n'-2t'}]_{d_1}$ to let R learn $x_1, \ldots, x_{n'-2t'}$.
- Run SemiRobustShare(d_2), where each R inputs $x_1, \ldots, x_{n'-2t'}$ to produce $[x_1]_{d_2}, \ldots, [x_{n'-2t'}]_{d_2}$ (step 1a is omitted as we do not need privacy). At the same time, check that R reshared the same blocks, namely in Step 1b we also apply M to the $[x_1]_{d_1}, \ldots, [x_{n'-2t'}]_{d_1}$, in Step 2a open the results to the servers and check for equality. Conflicts are removed by elimination as in SemiRobustShare.

Now all groups handled by $R \in \mathcal{S}'$ were correctly reshared with degree d_2. To deal with the fact that some blocks might not be reshared, we use the same idea as when we turned SemiRobustShare into RobustShare, namely the servers first apply $M_{n'-2t' \to n'}$ to each group of blocks to reshare, each of the resulting n' sharings are assigned to a server. Then each server does SemiRobustReshare on all his assigned sharings. Since a sufficient number of servers will complete this successfully, we can reconstruct d_2-sharings of the x_i's. This protocol is called RobustReshare.

Random Double Sharings. We use the following protocol to produce double sharings of blocks which are uniformly random in the view of the adversary.

Protocol RanDouSha(d):

1. Each server S_i: Pick a uniformly random block $R_i \in_R \mathbb{F}^\ell$ and use SemiRobustShare($d, 2d$) to deal $[R_i]_d$ and $[R_i]_{2d}$.
2. Let $M = M_{n' \to n'-t'}$ and let $([r_1]_d, \ldots, [r_{n'-t'}]_d) = M([R_i]_d)_{S_i \in \mathcal{S}'}$ and $([r_1]_{2d}, \ldots, [r_{n'-t'}]_{2d}) = M([R_i]_{2d})_{S_i \in \mathcal{S}'}$. The output is the pairs $([r_i]_d, [r_i]_{2d})$, $i = 1, \ldots, n'-t'$.

Security follows by observing that when $M = M_{n' \to n'-t'}$, then $M^H : \mathbb{F}^{n'-t'} \to \mathbb{F}^{n'-t'}$ is invertible when $|H| = n'-t'$. In particular, the sharings of the (at least) $n'-t'$ honest servers fully randomize the $n'-t'$ generated sharings in Step 2.

In the following, RanDouSha(d) is only run once, where a large number, β, of pairs $([r]_d, [r]_{2d})$ are generated in parallel. This gives a complexity of $\widetilde{\mathcal{O}}(\beta n) + \mathrm{poly}(n) = \widetilde{\mathcal{O}}(\phi)$, where ϕ is the number of field elements in the blocks.

Functionality $F_{CP}(A)$

The functionality initially chooses a random bitstring $K_1, .., K_k$ where k is the security parameter. It uses gm blocks of input bits $z_1^1, \ldots, z_m^1, ..., z_1^g, \ldots z_m^g$. Each block z_u^v can be:

- owned by an input client. The client can send the bits in z_u^v to F_{CP}, but may instead send "refuse", in which case the functionality sets $z_i^j = (0, ...0)$.
- Random, of type w, $1 \leq w \leq k$, then the functionality sets $z_u^v = (K_w, ..., K_w)$.
- Public, in which case some arbitrary (binary string) value for z_u^w is hardwired into the functionality.

The functionality works as follows:

1. After all input clients have provided values for the blocks they own, compute $A(z_1^v, ..., z_m^v)$ for $v = 1..g$.
2. On input "open v to server S_a" from all honest servers, send $A(z_1^v, ..., z_m^v)$ to server S_a.

Parallel Circuit Evaluation. Let $A : \mathbb{F}^m \to \mathbb{F}$ be an arithmetic circuit over $\mathbb{F}$. For m blocks containing binary values $z_1 = (z_{1,1}, \ldots, z_{1,\ell}), \ldots, z_m = (z_{m,1}, \ldots, z_{m,\ell})$ we let $A(z_1, \ldots, z_m) = (A(z_{1,1}, \ldots, z_{m,1}), \ldots, A(z_{1,\ell}, \ldots, z_{m,\ell}))$. We define an ideal functionality F_{CP} which on input that consists of such a group of input blocks will compute $A(z_1, ..., z_m)$. To get an efficient implementation, we will handle g groups of input blocks, denoted $z_1^1, \ldots, z_m^1, ..., z_1^g, \ldots z_m^g$ in parallel. Some of these bits will be chosen by input clients, some will be random, and some are public values, hardwired into the functionality. See the figure for details. The subsequent protocol CompPar securely implements F_{CP}. As for its efficiency, let γ denote the number of gates in A, and let M denote the multiplicative depth of the circuit (the number of times Step 2b is executed). Assume that $M = \text{poly}(k)$, as will be the case later. Then the complexity is easily seen to be $\widetilde{\mathcal{O}}(\gamma g n) + M\text{poly}(n) = \widetilde{\mathcal{O}}(\gamma g n)$. Let μ denote the number of inputs on which A is being evaluated. Clearly $\mu = g\ell = \Theta(gn)$, giving a complexity of $\widetilde{\mathcal{O}}(\gamma\mu)$. If we assume that $\gamma = \text{poly}(k)$, as will be the case later, we get a complexity of $\widetilde{\mathcal{O}}(\gamma\mu) = \widetilde{\mathcal{O}}(\mu)$, and this also covers the cost of sharing the inputs initially.

Protocol CompPar(A):

1. The servers run RanDouSha(d) to generate a pair $([r]_d, [r]_{2d})$ for each multiplication to be performed in the following.
2. Input: for each input client D, run RobustShareBits(d) in parallel for all blocks owned by D. Run Monochrom k times to get $[K_t, \ldots, K_t]_d$, for $t = 1...k$, and let $[z_u^v]_d = [K_w, \ldots, K_w]_d$ if z_u^v is random of type w. Finally, for all public z_u^v, we assume that default sharings of these blocks are hardwired into the programs of the servers.
   The servers now hold packed sharings $[z_u^v]_d$, all of which are d-consistent on $\mathcal{S}'$. Now do the following, for each of the g groups, in parallel:
   (a) For all addition gates in A, where sharings $[x]_d$ and $[y]_d$ of the operands are ready, the servers compute $[x+y]_d = [x]_d + [y]_d$ by locally adding shares. This yields a d-consistent sharing on $\mathcal{S}'$.
   (b) Then for all multiplication gates in A, where sharings $[x]_d$ and $[y]_d$ of the operands are ready, the servers execute:
      i. Compute $[xy+r]_{2d} = [a]_d[b]_d + [r]_{2d}$, by local multiplication and addition of shares. This is a $2d$-consistent sharing of $xy+r$ on $\mathcal{S}'$.
      ii. Call RobustReshare($2d, d$) to compute $[xy+r]_d$ from $[xy+r]_{2d}$. This is a d-consistent sharing of $xy+r$ on the reduced server set $\mathcal{S}'$. Note that all resharings are handled by one invocation of RobustReshare. Finally compute $[xy]_d = [xy+r]_d - [r]_d$.
   (c) If there are still gates which were not handled, go to Step 2a.
3. Output: When all gates have been handled, the servers hold for each group a packed sharing $[A(z_1^v, \ldots, z_m^v)]_d$ which is d-consistent over the current reduced server set $\mathcal{S}'$. To open group v to server S_a, run Reco(S_a, d).

Lemma 1. *Protocol* CompPar *securely implements* F_{CP}.

Sketch of proof: The simulator will use standard techniques for protocols based on secret sharing, namely whenever an honest player secret-shares a new block, the simulator will hand random shares to the corrupt servers. When a corrupted player secret-shares a value, the simulator gets all shares intended for honest servers, and follows the honest servers' algorithm to compute their reaction to this. In some cases, a value is reconstructed towards a corrupted player as part of a subprotocol. Such values are always uniformly random and this is therefore trivial to simulate. The simulator keeps track of all messages exchanged with corrupt players in this way. The perfect correctness of all subprotocols guarantees that the simulator can compute, from its view of RobustShareBits, the bits shared by all corrupt input clients, it will send these to F_{CP}. When an input client or a server is corrupted, the simulator will get the actual inputs of the client, respectively the outputs received by the server. It will then construct a random, complete view of the corrupted player, consistent with the values it just learned, and whatever messages the new corrupted player has exchanged with already corrupted players. This is possible since all subprotocols have perfect privacy. Furthermore the construction can be done efficiently by solving a system of linear equations, since the secret sharing scheme is linear. Finally, to simulate an opening of an output towards a corrupted server, we get the correct value from the functionality, form a complete random set of shares consistent with the shares the adversary has already and the output value, and send the shares to the adversary. This matches what happens in a real execution: since all subprotocols have perfect correctness, a corrupted server would also in real life get consistent shares of the correct output value from all honest servers. It is straightforward but tedious to argue that this simulation is *perfect*. □

4 Combining Yao Garbled Circuits and Authentication

To compute a circuit C securely, we will use a variant of Yao's garbled circuit construction [34, 35]. It can be viewed as building from an arbitrary circuit C together with a pseudorandom generator a new (randomized) circuit $C_{\mathtt{Yao}}$ whose depth is only $\mathrm{poly}(k)$ and whose size is $|C| \cdot \mathrm{poly}(k)$. The output of $C(x)$ is equivalent to the output of $C_{\mathtt{Yao}}(x, r)$, in the sense that given $C_{\mathtt{Yao}}(x, r)$ one can efficiently compute $C(x)$, and given $C(x)$ one can efficiently sample from the output distribution $C_{\mathtt{Yao}}(x, r)$ induced by a uniform choice of r (up to computational indistinguishability). Thus, the task of securely computing $C(x)$ can be reduced to the task of securely computing $C_{\mathtt{Yao}}(x, r)$, where the randomness r should be picked by the functionality and remain secret from the adversary.

In more detail, $C_{\mathtt{Yao}}(x, r)$ uses for each wire w in C two random encryption keys K_0^w, K_1^w and a random wire mask γ_w. We let $E_K()$ denote an encryption function using key K, based on the pseudorandom generator used. The construction works with an encrypted representation of bits, concretely $\mathrm{garble}_w(y) = (K_y^w, \gamma_w \oplus y)$ is called a *garbling of* y. Clearly, if no side information on keys or wire masks is known, $\mathrm{garble}_w(y)$ gives no information on y.

The circuit $C_{\mathtt{Yao}}(x, r)$ outputs for each gate in C a table with 4 entries, indexed by two bits (b_0, b_1). We can assume that each gate has two input wires l, r and output wire *out*. If we consider a circuit C made out of only NAND gates, $\dot\wedge$, a single entry in the table looks as follows:

$$(b_0, b_1): \quad E_{K^l_{b_0 \oplus \gamma_l}} \left(E_{K^r_{b_1 \oplus \gamma_r}} \left(\mathrm{garble}_{out} \left([b_0 \oplus \gamma_l] \dot\wedge [b_1 \oplus \gamma_r] \right) \right) \right) .$$

The tables for the output gates contain encryptions of the output bits without garbling, i.e., $[b_0 \oplus \gamma_l] \dot\wedge [b_1 \oplus \gamma_r]$ is encrypted. Finally, for each input wire w_i, carrying input bit x_i, the output of $C_{\mathtt{Yao}}(x, r)$ includes $\mathrm{garble}_{w_i}(x_i)$.

It is straightforward to see that the tables are designed such that given $\mathrm{garble}_l(b_l), \mathrm{garble}_r(b_r)$, one can compute $\mathrm{garble}_{out}(b_l \dot\wedge b_r)$. One can therefore start from the garbled inputs, work through the circuit in the order one would normally visit the gates, and eventually learn (only) the bits in the output $C(x)$. We will refer to this as *decoding* the Yao garbled circuit.

In the following, we will need to share the work of decoding a Yao garbling among the servers, such that one server only handles a few gates and then passes the garbled bits it found to other servers. In order to prevent corrupt servers from passing incorrect information, we will augment the Yao construction with digital signatures in the following way.

The authenticated circuit $C_{\texttt{AutYao}}(x,r)$ uses a random input string r and will first generate a key pair $(sk,pk) = \text{gen}(r')$, for a digital signature scheme, from some part r' of r. It makes pk part of the output. Signing of message m is denoted $S_{sk}(m)$. It will then construct tables and encrypted inputs exactly as before, except that a table entry will now look as follows:

$$G(b_0,b_1) = \ E_{K^l_{b_0\oplus\gamma_l}}\left(E_{K^r_{b_1\oplus\gamma_r}}(\text{garble}_{out}([b_0\oplus\gamma_l]\dot{\wedge}[b_1\oplus\gamma_r]), S_{sk}(e,b_0,b_1,L))\right)\ ,$$

where $e = \text{garble}_{out}\,[b_0\oplus\gamma_l]\dot{\wedge}[b_1\oplus\gamma_r]$ and L is a unique identifier of the gate. In other words, we sign exactly what was encrypted in the original construction, plus a unique label (b_0,b_1,L).

For each input wire w_i, it also signs $\text{garble}_{w_i}(x_i)$ along with some unique label, and makes $\text{garble}_{w_i}(x_i)$ and the signature σ_i part of the output. Since the gates in the Yao circuit are allowed to have fan-out,[3] we can assume that each input bit x_i to C appears on just one input wire w_i. Then the single occurrence of $(\text{garble}_{w_i}(x_i),\sigma_i)$ is the only part of the output of $C_{\texttt{AutYao}}(x,r)$ which depends on x_i. We use this below.

5 Combining Authenticated Yao Garbling and a PRF

Towards using CompPar for generating $C_{\texttt{AutYao}}(x,r)$ we need to slightly modify it to make it more uniform.

The first step is to compute not $C_{\texttt{AutYao}}(x,r)$, but $C_{\texttt{AutYao}}(x,\text{prg}(K))$, where $\text{prg} : \{0,1\}^k \to \{0,1\}^{|r|}$ is a PRG and $K \in \{0,1\}^k$ a uniformly random seed. The output distributions $C_{\texttt{AutYao}}(x,r)$ and $C_{\texttt{AutYao}}(x,\text{prg}(K))$ are of course computationally indistinguishable, so nothing is lost by this change. In fact, we use a very specific PRG: Let ϕ be a PRF with k-bit key and 1-bit output. We let $\text{prg}(K) = (\phi_K(1),\ldots,\phi_K(|r|))$, which is well known to be a PRG. Below we use $C_{\texttt{AutYao}}(x,K)$ as a short hand for $C_{\texttt{AutYao}}(x,\text{prg}(K))$ with this specific PRG.

The j's bit of $C_{\texttt{AutYao}}(x,K)$ depends on at most one input bit $x_{i(j)}$, where we choose $i(j)$ arbitrarily if the j'th bit does not depend on x. The uniform structure we obtain for the computation of $C_{\texttt{AutYao}}(x,K)$ is as follows.

Lemma 2. *There exists a circuit A of size $\text{poly}(k,\log|C|)$ such that the j'th bit of $C_{\texttt{AutYao}}(x,K)$ is $A(j,x_{i(j)},K)$.*

[3] For technical reasons, explained below, we assume that no gate has fan-out higher than 3, which can be accomplished by at most a constant blow-up in circuit size.

This follows easily from the fact that Yao garbling treats all gates in C the same way and that gates can be handled in parallel. The proof can be found in [16].

It is now straightforward to see that we can set the parameters of the functionality F_{CP} defined earlier so that it will compute the values $A(j, x_{i(j)}, K)$ for all j. We will call F_{CP} with A as the circuit and we order the bits output by $C_{\texttt{AutYao}}(x, K)$ into blocks of size ℓ. The number of such blocks will be the parameter g used in F_{CP}, and m will be the number of input bits to A. Blocks will be arranged such that the following holds for for any block given by its bit positions $(j_1, ..., j_\ell)$: either this block does nor depend on x or all input bits contributing to this output block, namely $(x_{i(j_1)}, \ldots, x_{i(j_\ell)})$, are given by one input client. This is possible as any input bit affects the same number of output bits, namely the bits in $\text{garble}_{w_i}(x_i)$ and the corresponding signature σ_i.

We then just need to define how the functionality should treat each of the input blocks z_u^v that we need to define. Now, z_u^v corresponds to the v'th output block and to position u in the input to A. Suppose that the v'th output block has the bit positions $(j_1, .., j_\ell)$. Then if u points to a position in the representation of j, we set z_u^v to be the public value $(j_1^u, \ldots, j_\ell^u)$, namely the u'th bit in the binary representations of $j_1, ..., j_\ell$. If u points to the position where $x_{i(j)}$ is placed and block v depends on x, we define z_u^v to be owned by the client supplying $(x_{i(j_1)}, \ldots, x_{i(j_\ell)})$ as defined above. And finally if u points to position w in the key K, we define z_u^v to be random of type w.

This concrete instantiation of F_{CP} is called $F_{\texttt{CompYao}}$, a secure implementation follows immediately from Lemma 1. From the discussion on CompPar, it follows that the complexity of the implementation is $\widetilde{\mathcal{O}}(|C|)$.

6 Delivering Outputs

Using $F_{\texttt{CompYao}}$, we can have the string $C_{\texttt{AutYao}}(x, K)$ output to the servers (ℓ bits at a time). We now need to use this to get the the results to the output clients efficiently. To this end, we divide the garbled inputs and encrypted gates into (small) subsets $\mathcal{G}_1, \ldots, \mathcal{G}_G$ and ask each server to handle only a fair share of the decoding of these.

We pick $G = n + (n - 2t)$ and pick the subsets such that no gate in $\mathcal{G}_g$ has an input wire w which is an output wire of a gate in $\mathcal{G}_{g'}$ for $g' > g$. We pick the subsets such that $|\mathcal{G}_g| = \widetilde{\mathcal{O}}(|C|/G)$, where $|\mathcal{G}_g|$ is the number of gates in $\mathcal{G}_g$. We further ensure that only the last $n - 2t$ subsets contain output wire carrying values that are to be sent to output clients. Furthermore, we ensure that all the L bits in the garbled inputs and encrypted gates for gates in $\mathcal{G}_g$ can be found in $\widetilde{\mathcal{O}}(L/\ell)$ blocks of $C_{\texttt{AutYao}}(x, K)$. This is trivially achieved by ordering the bits in $C_{\texttt{AutYao}}(x, K)$ appropriately during the run of CompPar.

We call a wire (name) w an *input wire* to $\mathcal{G}_g$ if there is a gate in $\mathcal{G}_g$ which has w as input wire, and the gate with output wire w (or the garbled input x_i for wire w) is not in $\mathcal{G}_g$. We call w an *output wire* from $\mathcal{G}_g$ if it is an output wire from a gate in $\mathcal{G}_g$ and is an input wire to another set $\mathcal{G}_{g'}$. We let the *weight* of $\mathcal{G}_g$, denoted $\|\mathcal{G}_g\|$, be the number of input wires to $\mathcal{G}_g$ plus the number of gates in $\mathcal{G}_g$ plus the number of output wires from $\mathcal{G}_g$. By the assumption that all gates have fan-out at most 3, $\|\mathcal{G}_g\| \leq 5|\mathcal{G}_g|$, where $|\mathcal{G}_g|$ is the number of gates in $\mathcal{G}_g$.

Protocol CompOutput:

1. All servers (in $\mathcal{S}'$): mark all $\mathcal{G}_g$ as unevaluated and let $c_i := 0$ for all S_i.[a]
2. All servers: let $\mathcal{G}_g$ be the lowest indexed set still marked as unevaluated, let $c = \min_{S_i \in \mathcal{S}'} c_i$ and let $S_i \in \mathcal{S}'$ be the lowest indexed server for which $c_i = c$.
3. All servers: execute open commands of $F_{\texttt{CompYao}}$ such that S_i receives $\mathcal{G}_g$ and pk.
4. Each $S_j \in \mathcal{S}'$: for each input wire to $\mathcal{G}_g$, if it comes from a gate in a set handled by S_j, send the garbled wire value to S_i along with the signature.
5. S_i: If some S_j did not send the required values, then broadcast $(\texttt{J'accuse}, S_j)$ for one such S_j. Otherwise, broadcast `ok` and compute from the garbled wire values and the encrypted gates for $\mathcal{G}_g$ the garbled wire values for all output wires from $\mathcal{G}_g$.
6. All servers: if S_i broadcasts $(\texttt{J'accuse}, S_j)$, then mark all sets $\mathcal{G}_{g'}$ previously handled by S_i or S_j as unevaluated and remove S_i and S_j from $\mathcal{S}'$. Otherwise, mark $\mathcal{G}_g$ as evaluated and let $c_i := c_i + 1$.
7. If there are $\mathcal{G}_g$ still marked as unevaluated, then go to Step 2.
8. Now the ungarbled, authenticated wire values for all output wires from C are held by at least one server. All servers send pk to all output clients, which adopt the majority value pk. In addition all servers send the authenticated output wire values that they hold to the appropriate output clients, which authenticate them using pk.

[a] c_i is a count of how many $\mathcal{G}_g$ were handled by S_i.

The details are given in Protocol CompOutput. We call a run from Step 2 through Step 6 *successful* if $\mathcal{G}_g$ became marked as evaluated. Otherwise we call it *unsuccessful*. For each successful run one set is marked as evaluated. Initially G sets are marked as unevaluated, and for each unsuccessful run, at most $2\lceil G/n' \rceil$ sets are marked as unevaluated, where $n' = |\mathcal{S}'|$. Each unsuccessful run removes at least one corrupted party from $\mathcal{S}'$. So, it happens at most $G + t2\lceil G/n' \rceil$ times that a set is marked as evaluated, and since $n' \geq n - 2t \geq 2t$, there are at most $2G + 2t$ successful runs. There are clearly at most t unsuccessful runs, for a total of at most $2G + 4t \leq 2G + n \leq 3G$ runs. It is clear that the complexity of one run from Step 2 through Step 6 is $\|\mathcal{G}_g\| \cdot \mathrm{poly}(k) + \mathrm{poly}(n, k) = \widetilde{\mathcal{O}}(\|\mathcal{G}_g\|) = \widetilde{\mathcal{O}}(|\mathcal{G}_g|) = \widetilde{\mathcal{O}}(|C|/G)$. From this it is clear that the communication and computational complexities of CompOutput are $\widetilde{\mathcal{O}}(|C|)$.

The CompOutput protocol has the problem that t corrupted servers might not send the output values they hold. We handle this in a natural way by adding robustness to these output values, replacing the circuit C by a circuit C' derived from C as follows. For each output client, the output bits from C intended for this client are grouped into blocks, of size allowing a block to be represented as $n - 3t$ field elements $(x_1, \ldots, x_{n-3t})$. For each block, C' then computes $(y_1, \ldots, y_{n-2t}) = M(x_1, \ldots, x_{n-3t})$ for $M = M_{n-3t \to n-2t}$, and outputs the y-values instead of the x-values. The bits of $(y_1, \ldots, y_{n-2t})$ are still considered as output intended for the client in question. The output wires for the bits of $y_1, \ldots, y_{n-2t}$ are then added to the sets $\mathcal{G}_{n+1}, \ldots, \mathcal{G}_{n+n-2t}$, respectively. Since $|\mathcal{S}'| \geq n-2t$ each of these $\mathcal{G}_g$ will be held by different servers at the end of CompOutput. So the output client will receive y_i-values from at least $n-3t$ servers, say in set H, and can then compute $(x_1, \ldots, x_{n-3t}) = M_H^{-1}(y_i)_{S_i \in H}$. Since $|C'| = \widetilde{\mathcal{O}}(|C|)$ and the interpolation can be done in time $\widetilde{\mathcal{O}}(n)$ we maintain the required efficiency.

Our overall protocol $\pi_{\texttt{out}}$ now consists of running (the implementation of) $F_{\texttt{CompYao}}$ using C' as the underlying circuit, and then CompOutput. We already argued the complexity of these protocols.

A sketch of the proof of security: we want to show that $\pi_{\texttt{out}}$ securely implements a functionality F_C that gets inputs for C from the input clients, leaks $C(x)$ to the adversary, and sends to each output client its part of $C(x)$.

We already argued that we have a secure implementation of $F_{\texttt{CompYao}}$, so it is enough to argue that we implement F_C securely by running $F_{\texttt{CompYao}}$ and then CompOutput. First, by security of the PRG, we can replace $F_{\texttt{CompYao}}$ by a functionality that computes an authenticated Yao-garbling $C_{\texttt{AutYao}}(x, r)$ using genuinely random bits, and otherwise behaves like $F_{\texttt{CompYao}}$. This will be indistinguishable from $F_{\texttt{CompYao}}$ to any environment.

Now, based on $C(x)$ that we get from F_C, a simulator can construct a simulation of $C_{\texttt{AutYao}}(x, r)$ that will decode to $C'(x)$, by choosing some arbitrary x' and computing $C_{\texttt{AutYao}}(x', r)$, with the only exception that the correct bits of $C'(x)$ are encrypted in those entries of output-gate tables that will eventually be be decrypted. By security of the encryption used for the garbling, this is indistinguishable from $C_{\texttt{AutYao}}(x, r)$.

The simulator then executes CompOutput with the corrupted servers and clients, playing the role of both the honest servers and $F_{\texttt{CompYao}}$ (sending appropriate ℓ-bit blocks of the simulated $C_{\texttt{AutYao}}(x, r)$ when required). By security of the signature scheme, this simulated run of CompOutput will produce the correct values of $C'(x)$ and hence $C(x)$ as output for the clients, consistent with F_C sending $C(x)$ to the clients in the ideal process. Thus we have the following:

Lemma 3 (Outer Protocol). *Suppose one-way functions exist. Then there is a constant $0 < \delta < 1/2$ such that for any circuit C there is an n-server δn-secure protocol $\pi_{\texttt{out}}$ for C which requires only $\mathrm{poly}(k, \log n, \log |C|) \cdot |C| + \mathrm{poly}(k, n)$ total computation (let alone communication) with security parameter k.*

We note that, assuming the existence of a PRG in NC^1, one can obtain a constant-round version of Lemma 3 for the case where there is only a constant number of output clients. The main relevant observation is that in such a case we can afford to directly deliver the outputs of $C_{\texttt{Yao}}$ to the output clients, avoiding use of CompOutput. The round complexity of the resulting protocol is proportional to the *depth* of $C_{\texttt{Yao}}(x, K)$, which is $\mathrm{poly}(k)$.[4] To make the

[4] Note that $C_{\texttt{Yao}}(x, K)$ cannot have constant depth, as it requires the computation of a PRF to turn K into randomness for $C_{\texttt{Yao}}$.

round complexity constant, we use the fact that a PRG in NC^1 allows to replace $C_{\mathtt{Yao}}(x, K)$ by a similar randomized circuit $C'_{\mathtt{Yao}}(x, K; \rho)$ whose depth is *constant* [1]. Applying CompPar to $C'_{\mathtt{Yao}}$ and delivering the outputs directly to the output clients yields the desired constant-round protocol. If one is content with a weaker form of security, namely "security with abort", then we can accommodate an arbitrary number of output clients by delivering all outputs to a single client, where the output of client i is encrypted and authenticated using a key only known to this client. The selected client then delivers the outputs to the remaining output clients, who broadcast an abort message if they detect tampering with their output.

7 Improving the Security Threshold Using Committees

In this section, we bootstrap the security of the protocol developed in the previous sections to resist coalitions of near-optimal size $(\frac{1}{2} - \epsilon)n$, for constant ϵ.

Theorem 1 (Main Result). *Suppose one-way functions exist. Then for every constant $\epsilon > 0$ and every circuit C there is an n-server $(\frac{1}{2} - \epsilon)n$-secure protocol Π for C, such that Π requires at most* $\mathrm{poly}(k, \log n, \log |C|) \cdot |C| + \mathrm{poly}(k, n)$ *total computation (and, hence, communication) with security parameter k.*

Moreover, if there exists a pseudorandom generator in NC^1 and the outputs of C are delivered to a constant number of clients, the round complexity of Π can be made constant with the same asymptotic complexity.

The main idea is to use *player virtualization* [5] to emulate a run of the previous sections' protocol among a group of n "virtual servers". Each virtual server is emulated by a committee of d real participants, for a constant d depending on ϵ, using a relatively inefficient SFE subprotocol that tolerates $\frac{d-1}{2}$ cheaters. The n (overlapping) committees are chosen so that an adversary corrupting $(\frac{1}{2} - \epsilon)n$ real players can control at most δn committees, where "controlling" a committee means corrupting at least $d/2$ of its members (and thus controlling the emulated server). As mentioned earlier (and by analogy which concatenated codes) we call the subprotocol used to emulate the servers the "inner" protocol, and the emulated protocol of the previous sections the "outer" protocol. For the inner protocol, we can use the protocol of Cramer, Damgård, Dziembowski, Hirt and Rabin [10] or a constant-round variant due to Damgård and Ishai [13].

The player virtualization technique was introduced by Bracha [5] in the context of Byzantine agreement to boost resiliency of a particular Byzantine agreement protocol to $(\frac{1}{3} - \epsilon)n$. It was subsequently used in several other contexts of distributed computing and cryptography, e.g. [25, 17, 23]. The construction of the committee sets below is explicit and implies an improvement on the parameters of the PSMT protocol of Fitzi *et al.* [17] for short messages.

We use three tools: the outer protocol from Lemma 3, the inner protocol and the construction of committee sets. The last two are encapsulated in the two lemmas below. The inner protocol will emulate an ideal, reactive functionality

$\mathcal{F}$ which itself interacts with other entities in the protocol. For the general statement, we restrict $\mathcal{F}$ to be "adaptively well-formed" in the sense of Canetti et al. [8] (see Lindell [31, Sec. 4.4.3], for a definition). All the functionalities discussed in this paper are well-formed.

Lemma 4 (Inner Protocol, [10, 13]). *If one-way functions exist then, for every well-formed functionality $\mathcal{F}$, there exists a UC-secure protocol $\pi_{\mathtt{in}}$ among d players that tolerates any $t \leq \frac{d-1}{2}$ adaptive corruptions. For an interactive functionality $\mathcal{F}$, emulating a given round of $\mathcal{F}$ requires* $\mathrm{poly}(comp_{\mathcal{F}}, d, k)$ *total computation, where $comp_{\mathcal{F}}$ is the computational complexity of $\mathcal{F}$ at that round, and a constant number of rounds.*

Strictly speaking, the protocols from [10, 13] are only for general secure function evaluation. To get from this the result above, we use a standard technique that represents the internal state of $\mathcal{F}$ as values that are shared among the players using verifiable secret sharing (VSS) Details can be found in [16].

Definition 1. *A collection $\mathcal{S}$ of subsets of $[n] = \{1, ..., n\}$ is a (d, ϵ, δ)-secure committee collection if all the sets in $\mathcal{S}$ (henceforth "committees") have size d and, for every set $B \subseteq [n]$ of size at most $(\frac{1}{2} - \epsilon)n$, at most a δ fraction of the committees overlap with B in $d/2$ or more points.*

Lemma 5 (Committees Construction). *For any $0 < \epsilon, \delta < 1$, there exists an efficient construction of a (d, ϵ, δ)-secure committee collection consisting of n subsets of $[n]$ of size $d = \mathcal{O}(\frac{1}{\delta\epsilon^2})$. Given an index i, one can compute the members of the i-th committee in time* $\mathrm{poly}(\log(n))$.

The basic idea is to choose a sufficiently good expander graph on n nodes and let the members of the ith committee be the neighbors of vertex i in the graph. The lemma is proved in [16].

We note that the same construction improves the parameters of the perfectly secure message transmission protocol of Fitzi *et al.* [17] for short messages. To send a message of L bits over n wires while tolerating $t = (\frac{1}{2} - \epsilon)n$ corrupted wires, their protocol requires $\mathcal{O}(L)+n^{\Theta(1/\epsilon^2)}$ bits of communication. Plugging the committees construction above into their protocol reduces the communication to $\mathcal{O}(L + n/\epsilon^2)$. A similar construction to that of Lemma 5 was suggested to the authors of [17] by one of their reviewers ([17, Sec. 5]). This paper is, to our knowledge, the first work in which the construction appears explicitly.

The final, composed protocol Π will have the same input and output clients as $\pi_{\mathtt{out}}$ and n virtual servers, each emulated by a committee chosen from the n real servers. These virtual servers execute $\pi_{\mathtt{out}}$. This is done in two steps:

First, we build a protocol Π' where we assume an ideal functionality $\mathcal{F}_i$ used by the i'th committee. $\mathcal{F}_i$ follows the algorithm of the i'th server in $\pi_{\mathtt{out}}$. When $\pi_{\mathtt{out}}$ sends a message from server i to server j, $\mathcal{F}_i$ acts as dealer in the VSS to have members of the jth committee obtain shares of the message, members then give these as input to $\mathcal{F}_j$. See [16] for details on the VSS to be used. Clients exchange messages directly with the $\mathcal{F}_i$'s according to $\pi_{\mathtt{out}}$. $\mathcal{F}_i$ follows its prescribed algorithm, unless a majority of the servers in the i'th committee are corrupted, in which case all its actions are controlled by the adversary, and it shows the adversary all messages it receives.

The second step is to obtain Π by using Lemma 4 to replace the $\mathcal{F}_i$'s by implementations via $\pi_{\mathtt{in}}$.

The proof of security for Π' is a delicate hybrid argument, and we defer it to [16]. Assuming Π' is secure, the lemma below follows from Lemma 4 and the UC composition theorem:

Lemma 6. *The composed protocol Π is a computationally-secure SFE protocol that tolerates $t = (\frac{1}{2} - \epsilon)n$ adaptive corruptions.*

As for the computational and communication complexities of Π, we recall that these are both $\widetilde{\mathcal{O}}(|C|)$ for $\pi_{\mathtt{out}}$. It is straightforward to see that the overhead of emulating players in $\pi_{\mathtt{out}}$ via committees amounts to a multiplicative factor of $O(\mathrm{poly}(k, d))$, where d is the committee size, which is constant. This follows from the fact that the complexity of $\pi_{\mathtt{in}}$ is $\mathrm{poly}(S, k, d)$ where S is the size of the computation done by the functionality emulated by $\pi_{\mathtt{in}}$. Therefore the complexity of Π is also $\widetilde{\mathcal{O}}(|C|)$. This completes the proof of the main theorem.

References

1. B. Applebaum, Y. Ishai, and E. Kushilevitz. Computationally private randomizing polynomials and their applications. In *Proc. CCC 2005*, pages 260-274.
2. Z. Beerliova-Trubiniova and M. Hirt. Efficient Multi-Party Computation with Dispute Control. In *Proc. TCC 2006*, pages 305-328.
3. Z. Beerliova-Trubiniova and M. Hirt. Perfectly-Secure MPC with Linear Communication Complexity. In *Proc. TCC 2008*, to appear.
4. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC 1988*, pages 1-10.
5. G. Bracha. An $O(\log n)$ expected rounds randomized byzantine generals protocol. *Journal of the ACM*, 34(4):910–920, 1987.
6. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proc. FOCS 2001*, pages 136-145.
7. R. Canetti, U. Feige, O. Goldreich, and M. Naor Adaptively Secure Multiparty Computation. In *Proc. STOC 96*, pages 639–648.
8. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation In *Proc. STOC 2002*, pages 494–503.
9. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. STOC 1988*, pages 11-19.
10. R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient Multiparty Computations Secure Against an Adaptive Adversary. In *Proc. EUROCRYPT 1999*, pages 311–326.
11. R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proc. Eurocrypt 2001*, pages 280–299.
12. I. Damgård and Y. Ishai. Scalable Secure Multiparty Computation. In *Proc. CRYPTO 2006*, pages 501–520.

13. I. Damgård and Y. Ishai. Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In *Proc. Crypto 2005*, pages 378–394.
14. I. Damgård, and J. Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Proc. Crypto 2003*, pages 247–264.
15. I. Damgård, and J. Nielsen. Robust multiparty computation with linear communication complexity. In *Proc. Crypto 2007*, pages 572–590.
16. I. Damgård, Y. Ishai, M. Krøigaard, J. Nielsen and A. Smith: *Scalable Multiparty Computation with Nearly Optimal Work and Resilience*, full version of this paper.
17. M. Fitzi, M. Franklin, J. Garay, and H. Vardhan. Towards optimal and efficient perfectly secure message transmission. In *TCC 07'*, pages 311–322, 2007.
18. M. Fitzi and M. Hirt. Optimally Efficient Multi-Valued Byzantine Agreement. In *Proc. PODC 2006*, pages 163–168.
19. M.K. Franklin and S. Haber. Joint Encryption and Message-Efficient Secure Computation. In *Proc. Crypto 1993*, pages 266–277. Full version in *Journal of Cyptoglogy* 9(4): 217–232 (1996).
20. M.K. Franklin and M. Yung. Communication Complexity of Secure Computation. In *Proc. STOC 1992*, pages 699–710.
21. R. Gennaro, M.O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proc. 17th PODC*,pages 101– 111, 1998.
22. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game (extended abstract). In *Proc. STOC 1987*, pages 218–229.
23. D. Harnik, Y. Ishai, and E. Kushilevitz. How many oblivious transfers are needed for secure multiparty computation? In *Proc. CRYPTO '07*, pages 284–302.
24. M. Hirt and U.M. Maurer. Robustness for Free in Unconditional Multi-party Computation. In *Proc. Crypto 2001*, pages 101–118.
25. M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.
26. M. Hirt, U.M. Maurer, and B. Przydatek. Efficient Secure Multi-party Computation. In *Proc. Asiacrypt 2000*, pages 143–161.
27. M. Hirt and J.B. Nielsen. Upper Bounds on the Communication Complexity of Optimally Resilient Cryptographic Multiparty Computation. In *Proc. Asiacrypt 2005*, pages 79–99.
28. M. Hirt and J.B. Nielsen. Robust Multiparty Computation with Linear Communication Complexity. In *Proc. Crypto 2007*, pages 572–590.
29. M. Jakobsson and A. Juels. Mix and Match: Secure Function Evaluation via Ciphertexts. In *Proc. Asiacrypt 2000*, pages 162–177.
30. E. Kushilevitz, Y. Lindell, and T. Rabin Information-theoretically secure protocols and security under composition. In *Proc. STOC 2006*, pages 109–118.
31. Y. Lindell. Composition of Secure Multi-Party Protocols, A Comprehensive Study. Springer 2003.
32. A. Lubotzky, R. Phillips, and P. Sarnak Ramanujan graphs. *Combinatorica* 8(3):261–277, 1988.
33. A. Shamir. How to share a secret. *Commun. ACM*, 22(6):612–613, June 1979.
34. A.C. Yao Theory and Applications of Trapdoor Functions (Extended Abstract). In *Proc. FOCS 1982*, pages 80–91.
35. A. C. Yao. How to generate and exchange secrets. In *Proc. FOCS 1986*, pages 162–167.