Secure Multi-Party Computation in Large Networks[*]

Varsha Dani[1], Valerie King[2], Mahnush Movahedi[3†], Jared Saia[1], Mahdi Zamani[4†]

[1]University of New Mexico, Albuquerque, NM
[2]University of New Victoria, Victoria, BC
[3]String Labs, Palo Alto, CA
[4]Visa Research, Palo Alto, CA

Abstract

We describe scalable protocols for solving the secure multi-party computation (MPC) problem among a significant number of parties. We consider both the synchronous and the asynchronous communication models. In the synchronous setting, our protocol is secure against a static malicious adversary corrupting less than a $1/3$ fraction of the parties. In the asynchronous environment, we allow the adversary to corrupt less than a $1/8$ fraction of parties. For any deterministic function that can be computed by an arithmetic circuit with m gates, both of our protocols require each party to send a number of messages and perform an amount of computation that is $\tilde{O}(m/n+\sqrt{n})$. We also show that our protocols provide statistical and universally-composable security.

To achieve our asynchronous MPC result, we define the *threshold counting problem* and present a distributed protocol to solve it in the asynchronous setting. This protocol is load balanced, with computation, communication and latency complexity of $O(\log n)$, and can also be used for designing other load-balanced applications in the asynchronous communication model.

1 Introduction

In *secure multi-party computation (MPC)*, a set of parties, each having a secret value, want to compute a common function over their inputs, without revealing any information about their inputs other than what is revealed by the output of the function. Recent years have seen a renaissance in MPC, but unfortunately, the distributed computing community is in danger of missing out. In particular, while new MPC algorithms boast dramatic improvements in latency and communication costs, none of these algorithms offer significant improvements in the highly *distributed* case, where the number of parties is large.

This is unfortunate since MPC holds the promise of addressing many important problems in distributed computing. How can peers in BitTorrent auction off resources without hiring an auctioneer? How can we design a decentralized Twitter that enables provably anonymous broadcast of messages? How can we create deep learning algorithms over data spread among large clusters of machines?

Most large-scale distributed systems are composed of nodes with limited resources. This makes it of extreme importance to *balance* the protocol load across all parties involved. Also, large networks tend to have weak admission control mechanisms, which makes them likely to contain malicious nodes. Thus, a key variant of the MPC problem that we consider will be when a certain hidden fraction of the nodes are controlled by a malicious adversary.

1.1 Our Contribution

In this paper, we describe general MPC protocols for computing arithmetic circuits when the number of parties is large. In terms of communication and computation costs per party, our protocols scale sublinearly with the number of parties

[*]This research was partially supported by NSF CAREER Award 0644058 and NSF grants CCR-0313160 and CCF-1320994.
[†]The work was done when this author was a student at the University of New Mexico.

and linearly with the size of the circuit.

For achieving sublinear communication and computation costs, our protocols critically rely on the notion of *quorums*. A quorum is a set of $O(\log n)$ parties, where the number of corrupted parties in each quorum is guaranteed not to exceed a certain fraction. We describe an efficient protocol for creating a sufficient number of quorums in the asynchronous setting.

To adapt to the asynchronous setting, we introduce the general problem of *threshold counting*. We show how this problem relates to the issue of dealing with arbitrarily-delayed inputs in our asynchronous MPC protocol, and then propose an efficient protocol for solving it.

When a protocol is concurrently executed alongside other protocols (or with other instances of the same protocol), one must ensure this composition preserves the security of the protocol. We show that our protocols are secure under such concurrent compositions by proving its security in the *universal composability (UC) framework* of Canetti [Can01].

1.2 Model

Consider n parties $P_1, \ldots, P_n$ in a fully-connected network with private and authenticated channels. In our asynchronous protocol, we assume communication is via asynchronous message passing so that sent messages may be arbitrarily and adversarially delayed. Latency (or running time) of a protocol in this model is defined as the maximum length of any chain of messages sent/received throughout the protocol (see [CD89, AW04]).

We assume a *malicious* adversary who controls an unknown subset of parties. We refer to these parties as *corrupted* and to the remaining as *honest*. The honest parties always follow our protocol, but the corrupted parties not only may share information with other corrupted parties but also can deviate from the protocol in any arbitrary manner, e.g., by sending invalid messages or remaining silent.

We assume the adversary is *static* meaning that it must select the set of corrupted parties at the start of the protocol. We assume that the adversary is computationally unbounded; thus, we make no cryptographic hardness assumptions.

1.3 Problem Statement

Multi-Party Computation. In the MPC problem, n parties, each holding a private input, want to jointly evaluate a deterministic n-ary function f over their inputs while ensuring:

1. Each party learns the correct output of f; and
2. No party learns any information about other parties' inputs other than what is revealed from the output.

Constraints for the Asynchronous Model. Consider a simple setting, where, the n parties send their inputs to a trusted party P who then locally computes f and sends the result back to every party. In the asynchronous setting, the MPC problem is challenging even with such a trusted party. In particular, since the t corrupted parties can refrain from sending their inputs to P, it can only wait for $n-t$ inputs rather than n inputs. Then, it can compute f over n inputs consisting of $n-t$ values received from the parties and t dummy (default) values for the missing inputs. Finally, the trusted party sends the output back to the parties. The goal of asynchronous MPC is to achieve the same functionality as the above scenario but without the trusted party P.

Quorum Building. Consider n parties connected pairwise via authenticated and private channels, where up to t of the parties are controlled by a malicious adversary. In the *quorum building problem*, the parties want to agree on a set of n quorums.

Definition 1 (Quorum)**.** *A* quorum *is a set of* $\Theta(\log n)$ *parties, where the fraction of corrupted parties in this set is at most* $t/n + \epsilon$*, for a small positive constant* ϵ*.*

Threshold Counting. In this problem, there are n honest parties each with a flag bit initially set to 0. At least $\tau < n$ of the parties will eventually set their bits to 1. The goal is for all the parties to learn when the number of bits set to 1 becomes greater than or equal to τ.

Definition 2 (Threshold Counting)**.** *Consider n honest parties connected to each other via an asynchronous network with authenticated pairwise channels. Each party is holding a bit that is initially set to* 0 *and will be permanently set to* 1 *upon some external event independently observed by the party. In the* threshold counting problem*, all parties want to learn eventually if the number of* 1 *bits has become greater than or equal to τ.*

1.4 Our Results

The main results of this paper are summarized by the following theorems. We consider an n-ary function, f, represented as an arithmetic circuit of depth d with m gates. We say an event occurs *with high probability (w.h.p)*, if on problems of size n, the event occurs with probability $1 - O(1/n^c)$, for any constant $c > 0$.

Theorem 1. *There exists a universally-composable protocol that with high probability solves the synchronous MPC problem and has the following properties:*

- *It is secure against $t < (1/3 - \epsilon)n$ corrupted parties, for any fixed $\epsilon > 0$;*
- *Each party sends $\tilde{O}(m/n + \sqrt{n})$ bits;*
- *Each party performs $\tilde{O}(m/n + \sqrt{n})$ computations;*
- *The latency is $O(d\,\mathsf{polylog}(n))$.*

We prove Theorem 1 in Section 5.3.

Theorem 2. *There exists a universally-composable protocol that with high probability solves the asynchronous MPC problem and has the following properties:*

- *It is secure against $t < (1/8 - \epsilon)n$ corrupted parties, for any fixed $\epsilon > 0$;*
- *Each party sends $\tilde{O}(m/n + \sqrt{n})$ bits;*
- *Each party performs $\tilde{O}(m/n + \sqrt{n})$ computations;*
- *The expected latency is $O(d\,\mathsf{polylog}(n))$.*

We prove Theorem 2 in Section 5.4.

Theorem 3. *There exists a protocol that solves the threshold counting problem among n parties with probability $1 - \frac{1}{7n}$ while ensuring:*

1. *Each party sends at most $O(\log n)$ messages of constant size;*
2. *Each party receives at most $O(\log n)$ messages;*
3. *Each party performs $O(\log n)$ computations;*
4. *Total latency is $O(\log n)$.*

We prove Theorem 3 in Section 6.

Theorem 4. *There exists a protocol that can solve the quorum building problem with probability $1 - 1/n^c$ (for some fixed $c > 0$) among n parties in the asynchronous communication model with the following properties:*

1. *The protocol is secure against $t < (1/4 - \epsilon)n$ corrupted parties, for any fixed $\epsilon > 0$;*
2. *The protocol builds n quorums each of size at most $c \log n$;*
3. *Each party sends at most $\tilde{O}(\sqrt{n})$ bits;*
4. *Each party performs $\tilde{O}(\sqrt{n})$ computations; and*
5. *The latency is $O(\mathsf{polylog}(n))$.*

We prove Theorem 1 in Section 7.

Paper Organization. In Section 2, we discuss related work. In Section 3, we define our notation and discuss the building blocks used in our protocols. We present our MPC protocols in Section 4. In Section 5, we prove the security of our MPC protocols. Section 6 is a self-contained presentation of the threshold counting problem and our solution to this problem. In Section 7, we describe an asynchronous protocol for the quorum building problem. Finally, we conclude in Section 8 and discuss future directions.

2 Related Work

Due to the large body of work, we do not attempt a comprehensive review of the MPC literature here, but rather focus on seminal work and, in particular, schemes that achieve sublinear per-party communication costs.

2.1 Traditional MPC

The MPC problem was first described by Yao [Yao82]. He described an algorithm for MPC with two parties in the presence of a semi-honest adversary. Goldreich et al. [GMW87] propose the first MPC protocol that is secure against a malicious adversary. This work along with [CDG88, GHY88] are all based on cryptographic hardness assumptions. These were later followed by several cryptographic improvements [BMR90, GRR98, CFGN96].

In a seminal work, Ben-Or et al. [BGW88] show that every function can be computed with information-theoretic security in the presence of a semi-honest adversary controlling less than half of the parties and in the presence of a malicious adversary controlling less than a third of the parties. They describe a protocol for securely evaluating an arithmetic circuit that represents the function.

This work was later improved in terms of both communication and computation costs in [CCD88, Bea91, GRR98]. Unfortunately, these methods all have poor communication scalability. In particular, if there are n parties involved in the computation, and the function f is represented by a circuit with m gates, then these algorithms require each party to send a number of messages and perform a number of computations that is $\Omega(nm)$.

These were followed by several improvements to the cost of MPC, when m (i.e., the circuit size) is much larger than n [DI06, DN07, DIK+08]. For example, the protocol of Damgård et al. [DIK+08] incurs computation and communication costs that are $\tilde{O}(m)$ plus a polynomial in n. Unfortunately, the additive polynomial in these algorithms is large (at least $\Omega(n^6)$) making them impractical for large n. One may argue that the circuit-dependent complexity dominates the polynomial complexity for large circuits. However, we believe there are many useful circuits such as the ones used in [MSZ15, HKI+12], which have relatively small number of gates.

2.2 Asynchronous MPC

Foundational work in asynchronous MPC was presented by Ben-Or et al. [BCG93]. They adapt the protocol of [BGW88] to the asynchronous setting and show that asynchronous MPC is possible for up to $n/3$ fail-stop faults and up to $n/4$ malicious faults. Improvements were made by Srinathan and Rangan [SR00] and Prabhu et al. [PSR02] with a final communication cost of $O(n^3)$ per multiplication. Beerliová-Trubíniová and Hirt [BTH07] achieved perfectly-secure asynchronous MPC with the optimal resiliency bound of up to $n/4$.

Damgård et al. [DGKN09] describe a perfectly secure MPC that guarantees termination only when the adversary allows a preprocessing phase to terminate. However, their protocol is not entirely asynchronous, as they assume a few synchronization points; hence, they can achieve a resiliency bound of up to $n/3$.

Choudhury et al. [CHP13] propose an amortized asynchronous MPC protocols with linear communication complexity per multiplication gate meaning that the communication done by an individual party for each gate does not grow with the number of parties. This protocol is unconditionally secure against up to $n/4$ corrupted parties with a small failure probability. In our paper, we are directly addressing the third open problem of [CHP13] as we quote here:

"If one is willing to reduce the resilience t from the optimal resilience by a constant fraction, then by using additional techniques like packed secret sharing, committee election and quorum forming, one can achieve further efficiency in the synchronous MPC protocols, as shown in [...]. It would be interesting to see whether such techniques can be used in the asynchronous settings to gain additional improvements."

2.3 MPC with Sublinear Overhead

We first introduced the notion of using quorums to decrease message cost in MPC in a brief announcement [DKMS12]. In that paper, we described a synchronous protocol with bit complexity of $\tilde{O}(m/n + \sqrt{n})$ per party that can tolerate a computationally unbounded adversary who controls up to $(1/3 - \epsilon)$ fraction of the parties for any fixed positive ϵ.

In a later paper [DKMS14], we extended our synchronous MPC result [DKMS12] to the asynchronous communication setting by requiring at most a $(1/8 - \epsilon)$ fraction corrupted parties. A major challenge in the asynchronous model is to ensure that at least $n - t$ inputs are committed to before the circuit is evaluated. To address this issue, we introduce and solve the *threshold counting problem*, where the parties jointly count the number of inputs submitted to protocol and proceed to the circuit evaluation step once this number exceeds a certain threshold. We later describe the details of this scheme and how it allows us to deal with the asynchrony of channels in MPC.

Boyle et al. [BGT13] also used the notion of quorums to design a synchronous MPC protocol with sublinear per-party communication cost. Their protocol is secure against a static and computationally bounded adversary corrupting up to $(1/3 - \epsilon)$ fraction of parties, for any fixed positive ϵ. Interestingly, the communication cost of their protocol is independent of circuit size. This is achieved by evaluating the circuit over encrypted values using a *fully-homomorphic encryption (FHE)* scheme [Gen09]. Unfortunately, this protocol is not fully load-balanced as it evaluates the circuit using only one quorum (called the supreme committee). The protocol requires each party to send $\mathsf{polylog}(n)$ messages of size $\tilde{O}(n)$ bits and requires $\mathsf{polylog}(n)$ rounds.

Chandran et al. [CCG+14] address two limitations of the protocol of [BGT13]: tolerating an adaptive adversary and achieving optimal resiliency (i.e., $t < n/2$ malicious parties). They replace the common reference string assumption of [BGT13] with a different setup assumption called symmetric-key infrastructure, where every pair of parties shares a uniformly-random key that is unknown to other parties. The authors also show how to remove the SKI assumption at a cost of increasing the communication locality by $O(\sqrt{n})$. Although this protocol provides small communication locality, the bandwidth cost seems to be super-polynomial due to large message sizes.

Boyle et al. [BCP15] also used the notion of quorums to achieve a sublinear MPC protocol for computing RAM programs [GO96] in large networks. For securely evaluating a RAM program Π, their protocol incurs a total communication and computation of $\tilde{O}(\mathsf{Comp}(\Pi) + n)$ while requiring $\tilde{O}(|x| + \mathsf{Space}(\Pi)/n)$ memory per party, where $\mathsf{Comp}(\Pi)$ and $\mathsf{Space}(\Pi)$ are computational and space complexity of Π respectively, and $|x|$ denotes the input size. While evaluation of RAM programs can in principle allow for more efficient function evaluation than evaluation of arithmetic circuits[1], the construction of [BCP15] incurs a memory overhead on every party that depends on the memory complexity of the program. Also, the protocol of [BCP15] only works in synchronous networks.

In Table 1, we review recent MPC results that provide sublinear communication locality. All of these results rely on some quorum building technique for creating a set of quorums each with honest majority.

The current paper is the complete version of our previous papers on synchronous [DKMS12] and asynchronous [DKMS14] MPC. We not only improve the presentation of our protocols but also provide techniques for facilitating their implementation in practice (see Section 4.1.3). We also completely modify our proofs to show the security of our protocols in the universal composability framework. We also describe a protocol for quorum building in the asynchronous setting, which is an extension of our previous work on quorum building [KLST11] (see Section 7).

2.4 Counting Networks

The threshold counting problem can be solved in a load-balanced way using *counting networks* that were first introduced by Aspnes et al. [AHS91]. Counting networks are constructed from simple two-input two-output computing elements called *balancers* connected to one another by wires. A counting network can count any number of inputs even if they arrive at arbitrary times, are distributed unevenly among the input wires, and propagate through the network asynchronously.

Aspnes et al. [AHS91] establish an $O(\log^2 n)$ upper bound on the depth complexity of counting networks. Since the latency of counting is dependent on the depth of the network, minimizing this depth has been the goal of many papers in this area. A simple explicit construction of an $O(c^{\log^* n} \log n)$-depth counting network (for some positive constant c), and a randomized construction of an $O(\log n)$-depth counting network that works with high probability are described by

[1]See [GKP+13] for a discussion about this.

Table 1: Recent MPC results with sublinear communication costs

Protocol	Adversary	Resiliency Bound	Async?	Total Message Complexity	Total Computation Complexity	Latency	Msg Size	LB?[1]
[BGT13][2]	Bounded	$(1/3-\epsilon)n$	No	$\tilde{O}(n)$	$\tilde{\Omega}(n)+\tilde{\Omega}(\kappa m d^3)$	$\tilde{O}(1)$	$O(n\ell \cdot \mathsf{polylog}(n))$	No
[BCP15]	Unbounded	$(1/3-\epsilon)n$	No	$\tilde{O}(\mathsf{Comp}(\Pi)+n)$	$\tilde{O}(\mathsf{Comp}(\Pi)+n)$	$\tilde{O}(\mathsf{Time}(\Pi))$	$O(\ell)$	Yes
[CCG+14]	Bounded[3]	$n/2$	No	$O(n\log^{1+\epsilon} n)$ or $O(n\sqrt{n}\log^{1+\epsilon} n)$	$\Omega(n\log^{1+\epsilon} n)$ or $\Omega(n\sqrt{n}\log^{1+\epsilon} n)$	$O(\log^{\epsilon'} n)$	$\Omega(\log^{\log n} n)$ or $\Omega(\sqrt{n}^{\log n})$	Yes
This paper (sync)	Unbounded	$(1/3-\epsilon)n$	No	$\tilde{O}(m+n\sqrt{n})$	$\tilde{O}(m+n\sqrt{n})$	$O(d+\mathsf{polylog}(n))$	$O(\ell)$	Yes
This paper (async)	Unbounded	$(1/8-\epsilon)n$	Yes	$\tilde{O}(m+n\sqrt{n})$	$\tilde{O}(m+n\sqrt{n})$	$O(d+\mathsf{polylog}(n))$	$O(\ell)$	Yes

Parameters: n is the number of parties; ℓ is the size of a field element; d is the depth of the circuit; κ is the the security parameter; ϵ, ϵ' are the positive constants; $\mathsf{Comp}(\Pi)$ is the computational complexity of RAM program Π; $\mathsf{Time}(\Pi)$ is the worst-case (parallel) running time of RAM program Π.

Notes:
[1]Is the protocol load-balanced?
[2]The total computation complexity is calculated based on the FHE scheme of [BGV12].
[3]Assumes a symmetric-key infrastructure. However, unlike the rest, this protocol is secure against an adaptive adversary.

Klugerman and Plaxton in [KP92, Klu95]. These constructions use the AKS sorting network [AKS83] as a building block. While this sorting network and the resulting counting networks have $O(\log n)$ depth and require each party (or gate in their setting) to send $O(\log n)$ messages, large hidden constants render them impractical.

3 Preliminaries

In this section, we define standard terms, notation, and known building blocks used throughout this paper.

Notation. We denote the set of integers $\{1,\ldots,n\}$ by $[n]$. A protocol is called *t-resilient* if no set of t or fewer parties can influence the correctness of the outputs of the remaining parties. We assume that all arithmetic operations in the circuit are carried out over a finite field $\mathbb{F}$. The size of $\mathbb{F}$ depends on the particular function to be computed and is always $\Omega(\log n)$. All of the messages transmitted by our protocol are logarithmic in $\mathbb{F}$ and n. We assume the inputs of each party is exactly one element of $\mathbb{F}$.

Let r be a value chosen uniformly at random from $\mathbb{F}$ and $\widehat{x} = x + r$, for any $x \in \mathbb{F}$. In this case, we say *x is masked with r* and we refer to r and $\widehat{x}$ as the *mask* and the *masked value*, respectively. We let $\mathsf{Majority}(x_1,\ldots,x_n)$ refer to a local procedure that returns the most common element in the list $x_1,\ldots,x_n$.

Universal Composability Framework. When a protocol is executed several times possibly concurrently with other protocols, one requires ensuring this composition preserves the security of the protocol. This is because an adversary attacking several protocols that run concurrently can cause more harm than by attacking a *stand-alone* execution, where only a single instance of one of the protocols is executed.

One way to ensure this is to show the security of the protocol under concurrent composition using the *universal composability (UC)* framework of Canetti [Can01]. A protocol that is secure in the UC framework is called *UC-secure*. We describe this framework in Section 5.

Verifiable Secret Sharing. An (n,t)-*secret sharing* scheme is a protocol in which a dealer who holds a secret value shares it among n parties such that any set of $t < n$ parties cannot gain any information about the secret, but any set of at least $t+1$ parties can reconstruct it. An (n,t)-*verifiable secret sharing (VSS)* scheme is an (n,t)-secret sharing scheme with the additional property that after the sharing stage, a dishonest dealer is either disqualified, or the honest parties can reconstruct the secret, even if shares sent by dishonest parties are spurious. When we say a set of shares of a secret are *valid*, we mean the secret can be uniquely reconstructed solely from the set of shares distributed among the parties.

In this paper, we use the VSS schemes of Ben-Or et al. in [BGW88] and [BCG93] for the synchronous and asynchronous settings, respectively. A secret sharing scheme is *linear* if given two shares a_i and b_i of secrets a and b, $c_i = a_i + b_i$ is a valid share of $c = a + b$. The VSS protocols of [BGW88] and [BCG93] are based on Shamir's linear secret sharing [Sha79] scheme. In this scheme, the dealer shares a secret s among n parties by choosing a random polynomial $f(x)$ of degree t such that $f(0) = s$. For all $i \in [n]$, the dealer sends $f(i)$ to the i-th party. Since at least $t+1$ points are required to reconstruct $f(x)$, no coalition of t or less parties can reconstruct s.

Theorem 5 ([BGW88])**.** *There exists a linear $(n,n/3)$-VSS scheme that is perfectly-secure against a static adversary in the synchronous communication model.*

Theorem 6 ([BCG93])**.** *There exists a linear $(n,n/4)$-VSS scheme that is perfectly-secure against a static adversary in the asynchronous communication model.*

We refer to the sharing stages of these VSS protocols as VSS and AVSS, respectively. The reconstruction algorithm requires a Reed-Solomon decoding algorithm [RS60] to correct up to t invalid shares sent by corrupted users. In our protocols, we use the error correcting algorithm of Berlekamp and Welch [BW86] described in the following. Let $\mathbb{F}_p$ denote a finite field of prime order p, and $S = \{(x_1, y_1) \mid x_i, y_i \in \mathbb{F}_p\}_{i=1}^{\eta}$ be a set of η points, where $\eta - \varepsilon$ of them are on a polynomial $y = P(x)$ of degree τ, and the rest $\varepsilon < (\eta - \tau + 1)/2$ points are erroneous.

Given the set of points S, the goal is to find the polynomial $P(x)$. The algorithm proceeds as follows. Consider two polynomials $E(x) = e_0 + e_1 x + \ldots + e_\varepsilon x^\varepsilon$ of degree ε, and $Q(x) = q_0 + q_1 x + \ldots + q_k x^k$ of degree $k \le \varepsilon + \tau - 1$ such that $y_i E(x_i) = Q(x_i)$ for all $i \in [\eta]$. This defines a system of η linear equations with $\varepsilon + k < \eta$ variables $e_0, \ldots, e_\varepsilon, q_0, \ldots, q_k$ that can be solved efficiently using Gaussian elimination technique to get the coefficients of $E(x)$ and $Q(x)$. Finally, calculate $P(x) = Q(x)/E(x)$.

Classic MPC. Our protocols rely on the classic MPC protocols of Ben-Or et al. [BGW88] and Ben-Or et al. [BCG93] for the synchronous and asynchronous settings, respectively.

Theorem 7 ([BGW88, AL11])**.** *There exists an MPC protocol that is perfectly secure under concurrent composition against a static adversary corrupting $t < n/3$ of the parties in the synchronous communication model. The protocol proceeds in $O(d)$ rounds, and each party sends $\mathsf{poly}(m,n)$ bits, where d is the depth of the circuit to be computed and m is the number of gates in the circuit.*

Theorem 8 ([BCG93])**.** *There exists an MPC protocol that is perfectly secure against a static adversary corrupting $t < n/4$ of the parties in the asynchronous communication model. The expected running time of the protocol is $O(d \log n)$, where d is the depth of the circuit to be computed. Each party sends $\mathsf{poly}(m,n)$ bits, where m is the number of gates in the circuit.*

When run among n parties to compute a circuit with m gates, both protocols send $O(m\,\mathrm{poly}(n))$ bits and incur a latency of $O(m)$. We refer to the former protocol as CMPC and to the latter as ACMPC.

In this paper, we use the above VSS and classic MPC protocols only among logarithmic-size groups of parties and only for computing logarithmic-size circuits. Thus, the communication overhead per invocation of these protocols will be $\mathsf{polylog}(n)$.

Byzantine Agreement. In the *Byzantine agreement* problem, each party is initially given an input bit. All honest parties must agree on a bit which coincides with at least one of their input bits.

When parties only have access to secure pairwise channels, a Byzantine agreement protocol is required to ensure reliable broadcast. This guarantees all parties receive the same message even if the broadcaster (dealer) is dishonest and sends different messages to different parties. Every time a broadcast is required in our protocols, we use the protocols of Ben-Or and El-Yaniv [BE03].

Theorem 9 ([BE03])**.** *There exist Byzantine agreement protocols that are perfectly secure under concurrent composition against a static adversary corrupting $t < n/3$ of the parties in the synchronous and asynchronous communication models. The protocol proceeds in constant rounds. Each party sends $\mathsf{poly}(n)$ bits.*

When all parties participating in a run of the broadcast protocol receive the same message, we say the parties have received *consistent* messages.

4 Our Protocols

We now describe our protocols for scalable MPC in large networks. Throughout this section, we consider the network model defined in Section 1.2. We first describe our synchronous protocol and then adapt this protocol to the asynchronous setting.

We assume that the parties have an arithmetic circuit C computing f; the circuit consists of m addition and multiplication gates. For convenience of presentation, we assume each gate has in-degree and out-degree 2.[1] For any two gates x and y in C, if the output of x is input to y, we say that x is a *child* of y and that y is a *parent* of x. We assume the gates of C are numbered $1, 2, \ldots, m$, where the gate numbered 1 is the output (root) gate.

4.1 Synchronous MPC

The high-level idea behind our protocols is first to create a sufficient number of quorums and assign to each gate in the circuit one of these quorums. Then, for each party P_i holding an input $x_i \in \mathbb{F}$, P_i secret-shares x_i among all parties in the quorum associated with the i-th input gate. We refer to such a quorum as an *input quorum*.

Next, the protocol evaluates the circuit gate-by-gate starting from input gates. Each gate is jointly evaluated by parties of the quorum associated with this gate over the secret-shared inputs provided by its children. In a similar way, the result of the gate is then used as the input to the computation of the parent gate. Finally, the quorum associated with the root gate constructs the final result and sends it to all parties via a binary tree of quorums.

This high-level idea relies on solutions to the following main problems.

Quorum Building. Creating a sufficient number of quorums. In Section 7, we describe a randomized protocol called Build-Quorums that achieves this goal with high probability.

Circuit Evaluation. Securely evaluating each gate over secret-shared inputs by the parties inside a quorum. In Section 4.1.2, we describe a protocol called Circuit-Eval that achieves this goal.

Share Renewal. Sending the result of one quorum to another without revealing any information to any individual party or to any coalition of corrupted parties in both quorums. We solve this as part of our gate evaluation protocol described in Section 4.1.2.

Protocol 1 is our main protocol. When we say a party *VSS-shares* (or *secret-shares*) a value s in a quorum Q (or among a set of parties), we mean the party participates as the dealer with input s in the protocol VSS with all parties in Q (or in the set of parties).

The protocol starts by running Build-Quorums to create n quorums $Q_1, \ldots, Q_n$. Then, it assigns the gates of C to these quorums in the following way. The output gate of C is assigned to Q_1; then, every gate in C numbered i (other than the output gate) is assigned to $Q_{(i \bmod n)}$. For each gate $u \in C$, we let Q_u denote the quorum associated with u, y_u denote the output of u, r_u be a random element from $\mathbb{F}$, and $\widehat{y}_v$ denote the masked output of u, where $\widehat{y}_u = y_u + r_u$.

[1]Our protocol works, with minor modifications, for gates with arbitrary constant fan-in and fan-out.

Protocol 1 MPC

1. **Quorum Building.** All parties run Build-Quorums to agree on n good quorums $Q_1,\ldots,Q_n$. The i-th gate of C is assigned to $Q_{(i \bmod n)}$, for all $i \in [m]$.
2. **Input Commitment.** For all $i \in [n]$, party P_i holding an input value $x_i \in \mathbb{F}$ runs the following steps:
   (a) Pick a uniformly random element $r_i \in \mathbb{F}$, set $\widehat{x} = x_i + r_i$, and broadcast $\widehat{x}$ to Q_i.
   (b) Run VSS to secret-share r_i in Q_i.
3. **Circuit Evaluation.** All parties participate in a run of Circuit-Eval to securely evaluate C.
4. **Output Reconstruction.** For the output gate z, each party in Q_z performs the following steps:
   (a) Send his share of r_z to all other parties in Q_z.
   (b) Upon receiving all the shares of r_z, run the Reed-Solomon decoding procedure to correct invalid shares. Then, reconstruct r_z from the corrected shares.
   (c) Set the circuit output message: $y \leftarrow \widehat{y}_z - r_z$.
   (d) Send y to all parties in the Q_2 and Q_3.
5. **Output Propagation.** For every $i \in \{2,\ldots,n\}$, each party in Q_i performs the following steps:
   (a) For each $j \in [q]$, receive $y^{(j)}$ from $P_j \in Q_{\lfloor i/2 \rfloor}$, and calculate $y \leftarrow \mathsf{Majority}\left(y^{(1)},\ldots,y^{(q)}\right)$.
   (b) If $2i \le n$, then send y to all parties in Q_{2i}. If $2i+1 \le n$, then send y to all parties in Q_{2i+1}.

4.1.1 Input Commitment

Let Q_i be the quorum associated with party P_i who holds input x_i. At the start of our protocol, P_i samples a value r_i uniformly at random from $\mathbb{F}$, sets $\widehat{x} = x_i + r_i$, and broadcasts $\widehat{x}$ to all parties in Q_i. Next, P_i runs VSS to secret-share r_i among all parties in Q_i.

4.1.2 Circuit Evaluation

The main idea for reducing the amount of communication required in evaluating the circuit is quorum-based gate evaluation. If each party participates in the computation of the whole circuit, it must communicate with all other parties. Instead, in quorum-based gate evaluation, each gate of the circuit is computed by a *gate gadget*. A gate gadget (see Figure 1) consists of three quorums: two *input quorums* and one *output quorum*. Input quorums are associated with the gate's children who serve inputs to the gate. The output quorum is associated with the gate itself and is responsible for creating a shared random mask and maintaining the output of the quorum for later use in the circuit. As depicted in Figure 1, these gate gadgets connect to form the entire circuit. In particular, for any gate u, the output quorum of u's gate gadget is the input quorum of the gate gadget for all of u's parents.

The parties in each gate gadget run CMPC among themselves to compute the gate operation. To ensure privacy is preserved, each gate gadget maintains the invariant that the value computed by the gadget is the value that the corresponding gate in the original circuit would compute, masked by a uniformly random element of the field. This random element is not known to any individual party. Instead, shares of it are held by the members of the output quorum. Thus, the output quorum can participate as an input quorum for the evaluation of any parent gate and provide both the masked version of the inputs and shares of the mask. The gate gadget computation is performed in the same way for all gates in the circuit until the final output of the whole circuit is computed. After the input commitment step, for each input gate v, parties in Q_v know the masked input $\widehat{y}_v$, and each has a share of the mask r_v.

Before proceeding to the circuit evaluation algorithm, let us discuss why we mask inputs/output of each quorum rather than just holding them as secret-shared values inside the quorum. Once the computation of the gate associated

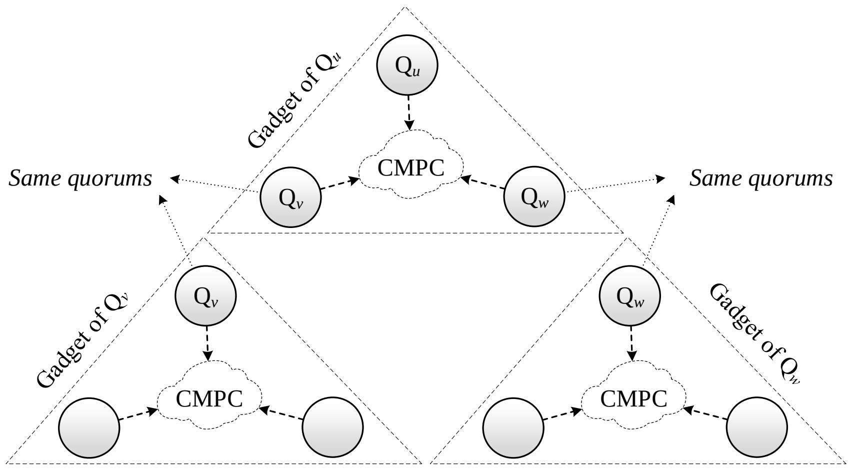

Figure 1: The gate gadgets for gate u and its left and right children

with Q_v is finished over secret-shared inputs, assume the parties in Q_v hold shares of y_v. The parties in Q_v need to forward the shared result to the parties in the next quorum in the circuit to serve as an input to the next gate. Let Q_u be the next quorum as shown in Figure 2. Since at most a $1/3$ fraction of the parties in each quorum are corrupted, if the parties in Q_v directly send their shares to the parties in Q_u, then the adversary can learn more than a $1/3$ fraction of the shares of y_v using a coalition of corrupted parties in Q_v and Q_u, and thus gain some information about y_v. Therefore, we need a fresh secret sharing of y_v as the input to Q_u that is independent of the sharing in Q_v. This is done by masking y_v using a fresh random value r_v to form the masked value $\widehat{y_v} = y_v + r_v$, which serves as an input to the computation of the next gate u. This random value is generated in Step 1 of Protocol 2. This mechanism prevents the adversary from learning more than a $1/3$ fraction of shares via a coalition of corrupted parties in two or more quorums.

The first step of the circuit evaluation is to generate shares of uniformly random field elements for all gates. If a party is in a quorum at gate u, it generates shares of r_u, a uniformly random field element, by participating in the Gen-Rand protocol. These shares are needed as inputs to the subsequent run of CMPC.

Next, parties form the gadget for each gate u to evaluate the functionality of the gate using Circuit-Eval. Let v and w be the left and right children of u respectively. The gate evaluation process is shown in Figure 2. The values y_v and y_w are the inputs to u, and y_u is its output as it would be computed by a trusted party. Each party in Q_u has a share of the random element r_u via Gen-Rand. Every party in Q_v has the masked value $y_v + r_v$ and a share of r_v (respectively for Q_w).

As shown in Part (b) of Figure 2, all parties in the three quorums participate in a run of CMPC, using their inputs, in order to compute $\widehat{y_u} = y_u + r_u$. Part (c) of the figure shows the output of the gate evaluation after participating in CMPC. Each party in Q_u now learns $\widehat{y_u}$ as well a share of r_u. Therefore, parties in Q_u now have the input required for performing the computation of parents of u (if any). Note that both y_u and r_u remain unknown to any individual.

The gate evaluation is performed for all gates in C starting from the bottom to the top. The output of the quorum associated with the output gate in C is the output of the entire algorithm. This quorum will unmask the output via the output reconstruction step. The last step of the algorithm is to send this output to all parties. We do this via a complete binary tree of quorums, rooted at the output quorum.

4.1.3 Implementing the Gate Circuit

For every gate $u \in C$, the Circuit-Eval protocol requires a circuit (as we denote by C_u) for unmasking the masked inputs $\widehat{y_v}$ and $\widehat{y_w}$, computing u's functionality f_u over the unmasked inputs, and masking the output with the gate's random value r_u. This circuit is securely evaluated using the CMPC protocol by the quorum associated with u.

Protocol 2 Circuit-Eval

Goal. Given a circuit C and n input quorums $Q_1, \ldots, Q_n$ holding inputs $x_1, \ldots, x_n$ respectively in secret-shared format, this protocol securely evaluates C. The protocol terminates with each party in the output quorum Q_z holding $\widehat{y}_z$ and a share of r_z.

For every level in C starting from level 1 and for every gate gadget rooted at $u \in C$ at the current level with children $v, w \in C$, perform the following steps:

1. **Mask Generation.** Parties in Q_u run Gen-Rand to jointly generate a secret-shared random value $r_u \in \mathbb{F}$.

2. **CMPC in Quorums.** The following parties participate in a run of CMPC with their corresponding inputs to compute the function defined below:

   - Every party in Q_u with his share of r_u.
   - Every party in Q_v with his input $(\widehat{y}_v,$ his share of $r_v)$.
   - Every party in Q_w with his input $(\widehat{y}_w,$ his share of $r_w)$.

   CMPC function (see Figure 3):

   (a) Reconstruct r_u, r_v and r_w.

   (b) $y_v \leftarrow \widehat{y}_v - r_v$

   (c) $y_w \leftarrow \widehat{y}_w - r_w$

   (d) $\widehat{y}_u \leftarrow f_u(y_v, y_w) + r_u$

Protocol 3 Gen-Rand

Goal. A set of parties $P_1, \ldots, P_q$ in a quorum want to agree on a secret-shared value r chosen uniformly at random from $\mathbb{F}$.

1. For all $i \in [q]$, party P_i chooses $\rho_i \in \mathbb{F}$ uniformly at random and VSS-shares it among all q parties.

2. For every $j \in [q]$, let q' be the number of shares P_j receives from the previous step, and $\rho_{1j}, \ldots, \rho_{q'j}$ be these shares. P_j computes $r_j = \sum_{k=1}^{q'} \rho_{kj}$.

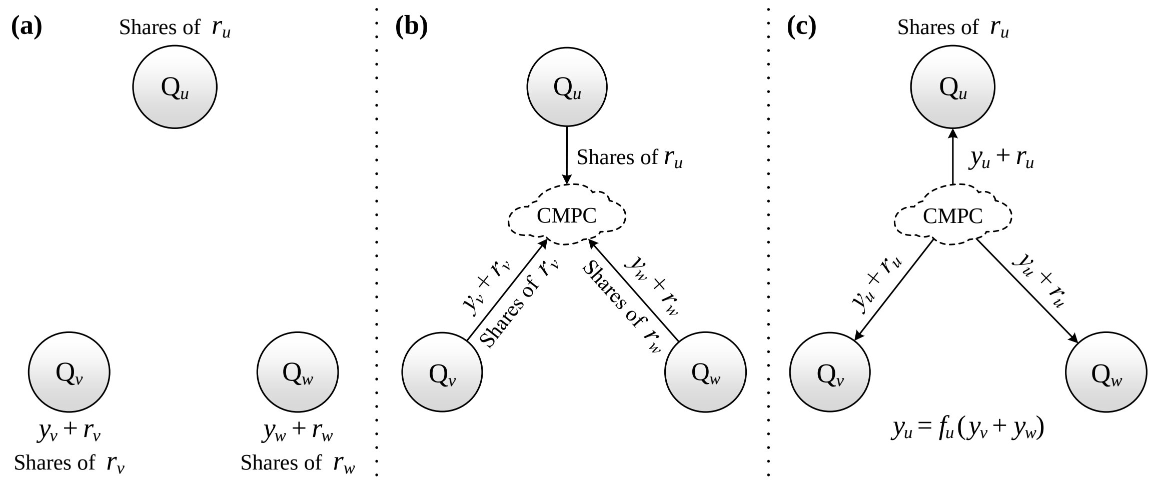

Figure 2: Evaluation of gate u: (a) generating r_u, (b) providing inputs to CMPC, (c) receiving the masked outputs

For unmasking an input, C_u requires a *reconstruction circuit*, which given a set of shares, outputs the corresponding secret. Since dishonest parties may send spurious shares, the circuit implements the error-correcting algorithm of Berlekamp and Welch [BW86] to fix such corruptions. Then, the resulting shares are given to an *interpolation circuit*, which implements a simple polynomial interpolation. Figure 3 depicts the circuit for gate u.

Following the decoding algorithm of [BW86], since the Gaussian elimination algorithm over finite fields has $O(n^3)$ arithmetic complexity [Far88], the corresponding circuit has at most $O(n^3)$ levels. Since the interpolation circuit consists of at most $O(n^2)$ arithmetic operations (using the Lagrange's method [Abr74]), the overall depth of the reconstruction circuit will be $O(n^3)$.

4.2 Asynchronous MPC

We now adapt our synchronous protocol to the asynchronous communication model. In the new model, we assume at most a $1/8 - \epsilon$ fraction of the parties is corrupted. We do this by modifying the following parts in Protocol 1:

1. We replace the synchronous subprotocols VSS and CMPC with their corresponding asynchronous versions AVSS and ACMPC respectively. In Section 7, we describe a technique for adapting Build-Quorums to the asynchronous setting. The new quorum building algorithm generates a set of n quorums each with at most a $1/8$ fraction of corrupted parties.

2. At the end of the Input Commitment stage, the protocol should wait for at least $n - t$ inputs before proceeding to the Circuit Evaluation stage. To this end, we introduce a new subprotocol called Wait-For-Inputs and invoke it right after step (b) of the Input Commitment stage. This protocol is described in Section 4.2.1.

3. Although the protocol ACMPC terminates with probability one, its actual running time (i.e., the number of rounds until it terminates) is a random variable with expected value $O(D \log q)$, where q is the number of parties participating in the MPC, and D is the circuit depth [BCG93]. Since we run m instances of ACMPC (one for each gate of C), we need a method that allows us to bound the running time of each gate, and thus to bound the expected running time of our asynchronous MPC protocol. We describe a simple method for achieving this in Section 4.2.2.

4. The Gen-Rand protocol will be replaced with a run of ACMPC, where each party participates with a value chosen uniformly at random from $\mathbb{F}$ to compute the sum of their inputs. The parties only execute the ACMPC protocol up to the output reconstruction step so that each party at the end of this protocol will hold a share of the random value. We call this asynchronous version of generating a shared random value Protocol Async-Gen-Rand.

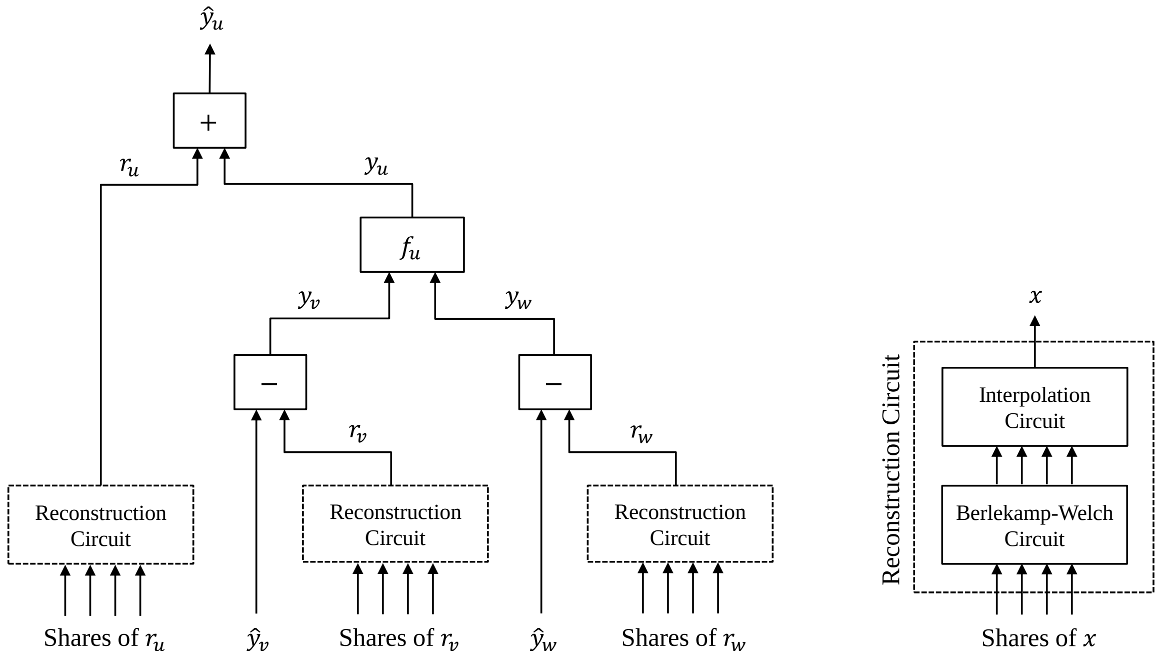

Figure 3: Circuit of gate u

4.2.1 Implementing Wait-For-Inputs

The protocol Wait-For-Inputs counts the number of inputs that are successfully received by their corresponding input quorums. This can be achieved using a solution to the threshold counting problem: Count the number of inputs successfully received by each input quorum and return once this number becomes greater than or equal to $n-t$. As a result of returning from Wait-For-Inputs, the main protocol resumes and starts the circuit evaluation procedure.

In Section 6, we provide a solution to the threshold counting problem. We refer to this protocol as Thresh-Count. This protocol creates a distributed tree structure called the *count tree*, which is known to all parties and determines how the parties communicate with each other to count the number of inputs.

Protocol 4 implements Wait-For-Inputs using our Thresh-Count algorithm. In Wait-For-Inputs, the role of each party in Thresh-Count (i.e., each node in the count tree) is played by a quorum of parties. Once Thresh-Count terminates, the parties in each input quorum decide whether or not the corresponding inputs are part of the computation.

When running among quorums, Thresh-Count requires the quorums to communicate with each other. We say a quorum Q sends a message M to quorum Q', when every (honest) party in Q sends M to every party in Q'. A party in Q' is said to have received M from Q if it receives M from at least $7/8$ of the parties in Q. When we say a party *broadcasts* a message M to a quorum Q, we mean the party sends M to every party in Q, and then, all parties in Q run a Byzantine agreement protocol over their messages to ensure they all hold the same message.

4.2.2 Bounding the Expected Running Time

For some $\gamma > 0$, let $O(n^\gamma)$ be the number of gates in the circuit C our MPC protocol wants to evaluate. Now, consider q parties in a quorum who want to compute a circuit of depth D jointly using the protocol ACMPC. Let X denote the random variable corresponding to the number of rounds until an instance of ACMPC terminates. From [BCG93], we have

$$\mathbf{E}[X] = O(D \log q).$$

Instead of running only one instance of ACMPC, we run $\lambda \log n$ instances sequentially each for $2\mathbf{E}[X]$ rounds, for any $\lambda > \gamma + 1$. The output corresponding to the first instance that terminates will be returned as the output of the gate. Note

Protocol 4 Wait-For-Inputs

Goal. For every input quorum Q, all parties in a quorum Q wait until $n-t$ inputs are received by the input quorums. For each party $P_i \in Q$, P_i is initially holding two values $\widehat{x}$ and r_i, the i-th share of a random value r.

Each party $P_i \in Q$ does the following:

1. Start Thresh-Count asynchronously.
2. $b_i \leftarrow 0$.
3. If $\widehat{x}$ and r_i's are consistent and valid (based on the Byzantine agreement protocol and the verification stage of AVSS respectively), set $b_i \leftarrow 1$ and send an event $\langle \text{Ready} \rangle$ to Thresh-Count.
4. When Thresh-Count terminates, run ACMPC jointly with other parties $P_1, \ldots, P_q$ to compute $\sum_{i=1}^{q} b_i$.
5. If the result is less than $5q/8$, then $\widehat{x} \leftarrow$ Default and $r_i \leftarrow 0$.

that since our MPC functionality is deterministic, it is secure to run multiple instances of ACMPC with the same inputs.[1] Using the Markov's inequality,

$$\Pr(X \geq 2\mathbf{E}[X]) \leq 1/2.$$

In each gate of C, each party also participates in a run of Gen-Rand, which invokes AVSS a total of q times. Similar to ACMPC, for each instance of AVSS, we run $\lambda \log n$ instances sequentially each for $2\mathbf{E}[X]$ rounds. The sharing corresponding to the first instance that terminates will be accepted by the parties.

Since $\lambda \log n$ instances of ACMPC and $\lambda q \log n$ instances of AVSS are executed in each gate, the computation of the gate terminates after at most

$$2\lambda \mathbf{E}[X] \log n = O(D \log q \log n)$$

rounds with error probability at most

$$(q+1)(1/2)^{\lambda \log n} = \frac{q+1}{n^{\lambda}}.$$

Since C has $O(n^{\gamma})$ gates, $q = O(\log n)$, and $\lambda > \gamma + 1$, by union bound over all gates of C, the protocol terminates with high probability. Finally, since C has depth d, the expected running time of our MPC protocol is $O(dD \log q \log n)$. In Section 4.1.2, we argued that the circuit computed by Circuit-Eval has depth $D = \text{polylog}(n)$. Thus, the expected running time of our asynchronous protocol is $O(d\,\text{polylog}(n))$.

4.3 Remarks

As described in the introduction, the goal of MPC is to simulate a trusted third party in the computation of the circuit, and then send back the computation result to the parties. Let S denote the set of parties from whom input is received by the (simulated) trusted party. Recall that $|S| \geq n-t$.[2] Thus, for an arbitrary S, a description of S requires $\Omega(n)$ bits, and cannot be sent back to the parties using only a scalable amount of communication. Therefore, we relax the standard requirement that S be sent back to the parties. Instead, we require that each honest party learns the output of f at the end of the protocol; whether or not their own input was included in S; and the *size* of S.

Also note that although we have not explicitly included this in the input commitment step, it is very easy for the parties to compute the size of the computation set S. Once each input quorum Q_i has performed the third step of Wait-For-Inputs and has agreed on the flag $b_i = 1$, they can only use an addition circuit to add these bits together, and then disperse the result. This is an MPC, all of whose inputs are held by honest parties since each input flag b_i is jointly owned by the entire quorum Q_i, and all the quorums are good. Thus, the computation can afford to wait for all n inputs and computes the correct sum.

[1]If the functionality was non-deterministic, the adversary could learn multiple samples from the secret input distributions when the MPC algorithm runs multiple times over the same inputs.

[2]We allow $|S| > n-t$ because the adversary is not limited to delivering one message at a time; two or more messages may be received simultaneously.

In our both protocols, a party P may participate in more than one quorum that is running a single instance of the classic MPC. In this case, we allow P to play the role of more than one different parties in CMPC and ACMPC, one for each quorum to which P belongs. This ensures that the fraction of corrupted parties in any instance of the classic MPC always remains less than $1/3$ for the synchronous case and $1/4$ for the asynchronous case. Also, note that CMPC and ACMPC both maintain privacy guarantees when a constant number of parties collude. Thus, each party will learn no information beyond the output and his own inputs when playing three different roles in each run of these protocols.

5 Security Proofs

We first describe the UC framework in Section 5.1 and then give a sketch of our proof in Section 5.2. We prove the UC-security of Protocol 1 in Section 5.3. We then prove the UC-security of our asynchronous protocol in Section 5.4. Finally, we calculate the resource costs of these protocols in Section 5.5.

5.1 The UC Framework

The UC framework is based on the *simulation paradigm* [Gol00], where the protocol is considered in two models: *ideal* and *real*. In the ideal model, the parties send their inputs to a trusted party who computes the function and sends the outputs to the parties. We refer to the algorithm run by the trusted party in the ideal model as the *functionality* of the protocol. In the real model, parties run the actual protocol that assumes no trusted party. We refer to a run of the protocol in one of these models as the *execution* of the protocol in that model.

A protocol $\mathcal{P}$ securely computes a functionality $F_{\mathcal{P}}$ if for every adversary $\mathcal{A}$ in the real model, there exists an adversary $\mathcal{S}$ in the ideal model, such that the result of a real execution of $\mathcal{P}$ with $\mathcal{A}$ is indistinguishable from the result of an ideal execution with $\mathcal{S}$. The adversary in the ideal model, $\mathcal{S}$, is called the *simulator*.

The simulation paradigm provides security only in the stand-alone model. For proving security under composition, the UC framework introduces an adversarial entity called the *environment*, denoted by $\mathcal{Z}$, which generates the inputs to all parties, reads all outputs, and interacts with the adversary in an arbitrary way throughout the computation. The environment also chooses inputs for the honest parties and gets their outputs when the protocol is finished.

A protocol is said to *UC-securely* compute an ideal functionality if for any adversary $\mathcal{A}$ that interacts with the protocol there exists a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can tell whether it is interacting with a run of the protocol and $\mathcal{A}$, or with a run of the ideal model and $\mathcal{S}$.

Now, consider a protocol $\mathcal{P}$ that has calls to ℓ subprotocols $\mathcal{P}_1, \ldots, \mathcal{P}_\ell$ already proved to be UC-secure. To facilitate the security proof of $\mathcal{P}$, we can make use of the *hybrid model*, where the subprotocols are assumed to be ideally computed by a trusted third-party. In other words, we replace each call to a subprotocol with a call to its corresponding functionality. This hybrid model is usually called the *$(F_{\mathcal{P}_1}, \ldots, F_{\mathcal{P}_\ell})$-hybrid* model. We say $\mathcal{P}$ is *UC-secure in the hybrid model* if $\mathcal{P}$ in the hybrid model is indistinguishable by the adversary from $\mathcal{P}$ in the ideal model. The *modular composition theorem* [Can00] states that if $\mathcal{P}_1, \ldots, \mathcal{P}_\ell$ are all UC-secure, and $\mathcal{P}$ is UC-secure in the hybrid model, then $\mathcal{P}$ is UC-secure in the real model.

5.2 Proof Sketch

Before proceeding to the proof, we remark that the error probabilities in Theorem 1 and Theorem 2 come entirely from the possibility that Build-Quorums or Thresh-Count fail to output correct results. All other components of our protocol are perfectly secure and always give correct results.[1] We also assume that, at the beginning of our MPC protocol, the parties have already agreed on n good quorums, and the threshold counting procedure is performed successfully.

As in [Gol04], we refer to the security in the presence of a malicious adversary controlling t parties *t-security*. For every gate $u \in C$, let I_u denote the set of corrupted parties in the quorum associated with u. Also, let I denote the set of all corrupted parties, where $|I| < t$.

Our goal is to prove the UC-security of our MPC protocols. To do this, we must show two steps. The first step is to demonstrate that each of our subprotocols is UC-secure. The second step is to show that our protocols are UC-secure in

[1]The running times of the AVSS and ACMPC protocols are random variables. We bounded the expected running time of our asynchronous MPC protocol in Section 4.2.2.

Table 2: Ideal functionalities for our synchronous (left) and asynchronous (right) MPC protocols

Functionality	Realized by
F_{VSS}	Protocol VSS
F_{CMPC}	Protocol CMPC
$F_{\mathsf{Gen\text{-}Rand}}$	Protocol Gen-Rand
F_{Input}	Input Commitment stage
$F_{\mathsf{Circuit\text{-}Eval}}$	Protocol Circuit-Eval
F_{Output}	Output Propagation stage

Functionality	Realized by
F_{AVSS}	Protocol AVSS
F_{ACMPC}	Protocol ACMPC
$F_{\mathsf{Async\text{-}Gen\text{-}Rand}}$	Protocol Async-Gen-Rand
$F_{\mathsf{Async\text{-}Input}}$	Input Commitment stage
$F_{\mathsf{Async\text{-}Circuit\text{-}Eval}}$	Circuit Evaluation stage
$F_{\mathsf{Async\text{-}Output}}$	Output Propagation stage

the hybrid model. Once we show these two steps, then by the modular composition theorem, we conclude that our protocols are UC-secure in the real model. In Lemma 11, we show the second step, that the adversary cannot distinguish the execution within the hybrid model from the ideal model.

We next describe our approach to the first step which is more challenging. For this step, we make use of a theorem that will help us show that our subprotocols are UC-secure. Kushilevitz et al. [KLR10] show Theorem 10. This theorem targets perfectly-secure protocols that are shown secure using a straight-line black-box simulator. A black-box simulator is a simulator that is given only oracle access to the adversary (see [Gol00] Section 4.5 for a detailed definition). Such a simulator is straight-line if it interacts with the adversary in the same way as real parties, meaning that it proceeds round by round without ever going back.

Theorem 10 ([KLR10])**.** *Every protocol that is perfectly secure in the stand-alone model and has a straight-line black-box simulator is UC-secure.*

We first define the ideal functionalities shown in Table 2 that correspond to the subprotocols used in our MPC protocols. We then prove that our synchronous MPC protocol is t-secure in the (F_{VSS}, F_{CMPC}, $F_{\mathsf{Gen\text{-}Rand}}$, F_{Input}, $F_{\mathsf{Circuit\text{-}Eval}}$, F_{Output})-hybrid model and that our asynchronous MPC protocol is t-secure in the (F_{AVSS}, F_{ACMPC}, $F_{\mathsf{Async\text{-}Gen\text{-}Rand}}$, $F_{\mathsf{Async\text{-}Input}}$, $F_{\mathsf{Async\text{-}Circuit\text{-}Eval}}$, $F_{\mathsf{Async\text{-}Output}}$)-hybrid model. Finally, we use Theorem 10 to infer the UC-security of this protocol. We first show that all of our subprotocols are UC-secure. Similar to the above approach, we first prove t-security of every subprotocol in its corresponding hybrid model using a straight-line black-box simulator, and then use Theorem 10 to infer its UC-security.

We stress that Theorem 10 holds only for perfectly-secure protocols, and in fact, is known not to hold for statistically-secure protocols, i.e., protocols with negligible probability of error [KLR10]. Therefore, we must be careful in addressing the error probability of our subprotocols when using Theorem 10.

The only components of our asynchronous protocol that have probabilities of error and thus make our protocols statistically-secure are Build-Quorums and Thresh-Count. Both of these protocols do not receive any information about parties' inputs, and therefore only require proofs of correctness. If Build-Quorums fails, which happens with negligible probability, then it may create quorums with more than a $t/n+\epsilon$ corrupted parties, which can compromise user input privacy later in MPC. If Build-Quorums is used to create quorums of size $c\log n$, then the error probability is $\frac{1}{n^c}$ (see Theorem 4). If Thresh-Count fails, which happens with probability $\frac{1}{7n}$ (see Theorem 3), then the MPC protocol may proceed to the circuit evaluation stage before or after $n-t$ inputs have been counted. This failure, however, does not compromise input privacy at all. If Build-Quorums and Thresh-Count do not fail, which happens in our asynchronous case with probability $1-\frac{1}{n^c}-\frac{1}{7n}$, then the protocol is perfectly secure in the stand-alone model and by Theorem 10 is UC-secure. By using a similar argument, our synchronous protocol is statistically secure with probability $1-\frac{1}{n^c}$ and is UC-secure.

To prove the t-security of a protocol Π, we describe a simulator $\mathcal{S}_\Pi$ that simulates the real protocol execution by running a copy of Π in the ideal model. For each call to a secure subprotocol π, the simulator calls the corresponding ideal functionality F_π. A *view* of a corrupted party from the execution of a protocol is defined as the set of all messages it receives during the execution of that protocol plus the inputs and the outputs it receives. At every stage of the simulation process, $\mathcal{S}_\Pi$ adds the messages received by every corrupted party in that stage to its view of the simulation. This is achieved by running a copy of Π for each corrupted party with his actual input as well as by running a copy of Π for each honest party with a dummy input.[1] The view of the adversary is then defined as the combined view of all corrupted parties.

5.3 Proof of Theorem 1

In this section, we first describe the functionalities F_{VSS} and F_{CMPC}. For each functionality, we refer the reader to an existing protocol that implements the functionality and shows its security.

Security of VSS. We use the proof of Asharov and Lindell [AL11] for the VSS protocol of Ben-Or et al. [BGW88], where the corresponding functionality is denoted by F_{VSS} (see Functionality 5.5 and Protocol 5.6 of [AL11]). In F_{VSS}, the dealer provides a polynomial $q(x)$ of degree t to the functionality, and each party P_i receives a share $q(i)$. We restate Theorem 5.7 of [AL11] here without its proof.

Theorem 11 (Theorem 5.7 of [AL11])**.** *Let $t < n/3$. Protocol VSS is perfectly t-secure for F_{VSS} in the presence of a synchronous static malicious adversary.*

Security of CMPC. We use the proof of Asharov and Lindell [AL11] for the CMPC protocol (see Protocol 7.1 of [AL11]) and denote the corresponding functionality by F_{CMPC} as defined in Protocol 5.

Protocol 5 F_{CMPC}

Goal. The functionality is parametrized by an arbitrary q-input function $g : \mathbb{F}^q \to \mathbb{F}^q$. The functionality interacts with q parties and the adversary.

Functionality:

1. For every input x_i of g, the functionality receives x_i from party P_i, for all $i \in [q]$.
2. The functionality computes $y = g(x_1, x_2, ..., x_q)$ and sends y to all parties and adversary.

Theorem 12 (Corollary 7.3 of [AL11])**.** *For every function $g : \mathbb{F}^q \to \mathbb{F}^q$ and $T < q/3$, CMPC is perfectly T-secure for F_{CMPC} parameterized by g over private synchronous channels and in the presence of a static malicious adversary.*

5.3.1 Security of Input Commitment

This section defines the functionality of the input commitment stage of Protocol 1.

Protocol 6 F_{Input}

Goal. The functionality guarantees valid inputs are received by all input quorums.

Functionality:

1. For every input x_i, the functionality receives $\widehat{x}_i = x_i + r_i$ and r_i from party P_i, for all $i \in [n]$.
2. For each $i \in [n]$, the functionality broadcast $\widehat{x}_i$ to Q_i.
3. For each $i \in [n]$, the functionality shares r_i among q parties of Q_i by choosing a random polynomial $h(x)$ of degree T such that $h(0) = r_i$.
4. For all $j \in [q]$, the functionality sends $h(j)$ to the j-th party in Q_i. This guarantees that honest parties in Q_i can reconstruct the correct r_i later, but dishonest parties cannot.

Note that F_{Input} is necessary to build the base case for starting the computation of the circuit. In Lemma 1, we show that the input commitment stage securely implements F_{Input}.

Lemma 1. *If the quorum formation stage of Algorithm 1 finished successfully, the Input Commitment stage of Protocol 1 securely UC-realizes functionality F_{Input} in the synchronous model with $1/3 - \epsilon$ malicious corruptions.*

[1] $\mathcal{S}_\Pi$ learns neither the actual inputs nor the actual outputs of the honest parties.

Proof. We prove the t-security of the Input Commitment stage in the F_{VSS}-hybrid model, which is similar to the Input Commitment stage of Protocol 1 except that every call to its subprotocol VSS is replaced with a call to F_{VSS}. We define the corresponding simulator $\mathcal{S}_{\mathsf{Input}}$ in Protocol 7.

Protocol 7 $\mathcal{S}_{\mathsf{Input}}$

For every $i \in [n]$, party P_i holds an input $x_i \in \mathbb{F}$. Associated with this input, we consider a quorum Q_i. Let I_i denote the set of corrupted parties in Q_i, and let I denote the set of all corrupted parties among $P_1, \ldots, P_n$.

Inputs. $\{r_i\}_{i\in[n]}$ and $\{\widehat{x_i}\}_{i\in[n]}$ from parties in I (the set of all corrupted parties).

Simulation:
For every $i \in [n]$,

1. If $P_i \in I$, obtain from the adversary x_i and the polynomial that it instructs P_i to send to the F_{VSS} as a dealer of r_i.
2. Interaction with the trusted party: the simulator send the inputs of corrupted parties to the trusted party and receives the output values $\widehat{x_i}$ and shares of r_i for corrupted parties.
3. If $P_i \notin I$, choose r_i and x_i uniformly at random from $\mathbb{F}$ and with constraint that $\widehat{x_i} = x_i + r_i$.
4. Broadcast $\widehat{x_i}$ to all parties in Q_i.
5. Run F_{VSS} to secret-share r_i in Q_i such that for each corrupted party, the share of r_i is compatible with his output (F_{VSS} outputs evaluations of a polynomial).
6. For every party in I_i, add his share of r_i (as it expects from the F_{VSS} invocations) and $\widehat{x_i}$ to his view.

Before proceeding to the proof, note that we assume the quorum formation stage of Algorithm 1 finished successfully. Based on Theorem 4, for each input gate $i \in [n]$, at most $1/3$ fraction of the parties in Q_i are malicious.

Let V_1 denote the view of the adversary from the hybrid execution, and V_2 be its view from the simulation. We first show that V_2 is indistinguishable from the view that the adversary might get from running the simulator with real inputs for honest parties. For each party P_i, V_2 (and similarly V_2 from actual inputs of honest parties) contains the masked input, $\widehat{x_i}$, and at most $1/3$ fraction of the shares for its random mask r_i received from F_{VSS}, since Q_i has only $1/3$ fraction malicious parties (See Theorem 4). The masked inputs convey no information about the inputs since r_i are chosen randomly by the simulator or an honest party. Moreover, a $1/3$ fraction of the shares is not enough to reconstruct the random number r_i based on the properties of VSS. Moreover, the values for $\widehat{x_i}$ and shares of r_i for different, honest parties are independent to each other. Thus, the adversary cannot distinguish the two views.

Next, we show that the hybrid execution of the Input Commitment stage correctly computes F_{Input} and its output has the desired properties as F_{Input} while it is indistinguishable from the simulator execution with actual inputs for honest parties. Since at least $2/3$ of the parties in Q_i are honest, they have the correct shares from P_i for r_i and have the proper value for $\widehat{x_i}$. This is enough information to reconstruct r_i and find $x_i = \widehat{x_i} - r_i$ later.

Based on the UC-security of VSS (Theorem 11), the modular composition theorem, and the fact that our simulator is straight-line and black-box, and our protocol is perfectly secure, it follows from Theorem 10 that the Input Commitment stage is UC-secure. □ □

5.3.2 Security of Circuit Evaluation

We first prove the security of Gen-Rand. The ideal functionality $F_{\mathsf{Gen\text{-}Rand}}$ is given in Protocol 8. Based on Theorem 4, at least $2q/3$ of parties in each quorum are honest, thus $2q/3$ of the inputs $\rho_1, \ldots, \rho_q$ are sent by honest parties and are chosen uniformly and independently at random from $\mathbb{F}$. Hence, $r = \sum_{i=1}^{q} \rho_i$ is also a uniform and independent random element of $\mathbb{F}$. This is because the sum of elements of $\mathbb{F}$ is uniformly random if at least one of them is uniformly random.

Lemma 2. *If the quorum formation stage of Algorithm 1 finished successfully, the protocol Gen-Rand securely UC-realizes functionality $F_{\mathsf{Gen\text{-}Rand}}$ in the synchronous model, with $1/3 - \epsilon$ malicious corruptions.*

Proof. We prove the t-security of Gen-Rand in the F_{VSS}-hybrid model, which is similar to Protocol 3 except that every

Protocol 8 $F_{\textsf{Gen-Rand}}$

Goal. For a gate $u \in C$, generate a random value $r \in \mathbb{F}$ and secret share it among parties $P_1, \ldots, P_q$ in the quorum associated with u.

Functionality:

1. Receive inputs $\rho_1, \ldots, \rho_q \in \mathbb{F}$ from $P_1, \ldots, P_q$ respectively. For every $i \in [q]$, if P_i does not send an input, then define $\rho_i = 0$.
2. Calculate $r = \sum_{i=1}^{q} \rho_i$.The functionality shares r among q parties of Q_u by choosing a random polynomial $h(x)$ of degree T such that $h(0) = r$. For all $j \in [q]$, the functionality sends $h(j)$ to the j-th party in Q_u. .

call to VSS is replaced with a call to the ideal functionality $F_{\textsf{VSS}}$. The corresponding simulator $\mathcal{S}_{\textsf{Gen-Rand}}$ is given in Protocol 9.

Protocol 9 $\mathcal{S}_{\textsf{Gen-Rand}}$

Inputs. For a gate $u \in C$, the inputs $\{\rho_j\}_{P_j \in I_u}$ of the corrupted parties $P_1, \ldots, P_q$ in the quorum associated with u.

Simulation:

1. For every $P_i \in (Q_u - I_u)$ (i.e., for every honest party P_i), call $F_{\textsf{VSS}}$ with dummy input 0. Let $s_1^i, \ldots, s_q^i$ denote the outputs.
2. For every $P_j \in I_u$,
   (a) Run $F_{\textsf{VSS}}$ with input ρ_j. Let $\rho_1^j, \ldots, \rho_q^j$ denote the outputs. For every $k \in [q]$, add ρ_j^k to the view of P_j.
   (b) Compute $r_j = \sum_{k=1}^{q} \rho_j^k$ and add r_j to the view of P_j.

The views of the corrupted parties in the hybrid execution and the simulation are indistinguishable because the only difference between the two views is that $\mathcal{S}_{\textsf{Gen-Rand}}$ generates the shares from dummy input 0 instead of actual inputs. Since $F_{\textsf{VSS}}$ creates uniform and independent random shares from any input, the two views are identically distributed. Based on the UC-security of VSS (Theorem 11) and since our simulator is straight-line and black-box, and our protocol is perfectly-secure, Gen-Rand is UC-secure. □ □

We now proceed to the security proof of Circuit-Eval. The ideal functionality $F_{\textsf{Circuit-Eval}}$ is given in Protocol 10.

Lemma 3. *If the quorum formation stage of Algorithm 1 finished successfully, the protocol Circuit-Eval securely UC-realizes functionality $F_{\textsf{Circuit-Eval}}$ in the synchronous model, with $1/3 - \epsilon$ malicious corruptions.*

Proof. We now prove the t-security of Circuit-Eval in the ($F_{\textsf{Gen-Rand}}$,$F_{\textsf{CMPC}}$)-hybrid model, which is similar to Protocol 2 except that every call to CMPC and Gen-Rand is replaced with a call to $F_{\textsf{CMPC}}$ and $F_{\textsf{Gen-Rand}}$ respectively. The corresponding simulator $\mathcal{S}_{\textsf{Circuit-Eval}}$ is given in Protocol 11.

We now show that the views of the corrupted parties in the hybrid execution and the simulation are indistinguishable. Let $I_\triangle = I_u \cup I_v \cup I_w$. After the evaluation of u, the following information will be added to the view of every corrupted party $P_i \in I_\triangle$: $\widehat{y}_u$ and $\{r_u^{(j)}\}_{P_j \in I_u}$. Recall that $\widehat{y}_u$ is the output of $F_{\textsf{CMPC}}$ during the computation of u, which is equal to $y_u + r_u$, and r_u is a uniformly random element of $\mathbb{F}$ based on $F_{\textsf{Gen-Rand}}$, independent of all other randomness in the algorithm.

First, if a corrupted party P_i is not in any of the quorums associated with u, v, and w, then no additional information will be added to its view during the computation of u; thus, its view will be identically distributed in the hybrid execution and the simulation.

Second, a corrupted party $P_i \in I_\triangle$ may add a share of r_u as well as shares of the individual random elements whose sum is r_u to its view in the computation of $F_{\textsf{Gen-Rand}}$. Also, it adds $y_u + r_u$ to its view. However, P_i cannot learn any additional information about the shares of r_u (and thus about r_u) based on $F_{\textsf{CMPC}}$ and $F_{\textsf{Gen-Rand}}$. In other words, the parties in $I_\triangle$ are unable to determine r_u directly, since the only relevant inputs are the shares of r_u, and they do not have enough of those since they have fewer than half of them.

These parties also do not have enough shares of shares of r_u to reconstruct it. However, they add to their view shares of each of the other shares of r_u multiple times: once during the input stage of $F_{\textsf{CMPC}}$ in which u is involved, and

Protocol 10 $F_{\textsf{Circuit-Eval}}$

Goal. For every level in circuit C starting, and for every gate $u \in C$, the functionality evaluate the function of gate u denoted by f_u.

Input. For every $i \in [n]$, the functionality receives inputs from the parties in input quorum Q_i as the based case for starting its computation. For every gate $u \in C$ with children $v, w \in C$, if v, w are input quorums Q_u and Q_v, the functionality receives $\widehat{y}_v$ and $r_v^{(i)}$ from parties in Q_u, and $\widehat{y}_w$ and $r_w^{(i)}$ from parties in Q_v respectively.
Functionality:
For every level in circuit C starting from level 1, and for every gate $u \in C$ with children $v, w \in C$ perform the following:

1. For every $i \in [q]$, receive ρ_i from P_i in Q_u and calculate $r_u = \sum_{i=1}^{q} \rho_i$.
2. The functionality shares r_u among q parties of Q_u by choosing a random polynomial $h(x)$ of degree T such that $h(0) = r_u$. For all $j \in [q]$, the functionality stores $r_u^{(j)} = h(j)$ and sends it to the j-th party in Q_u.
3. The functionality Computes the following:
   (a) $r_u \leftarrow$ The result of running a Reed-Solomon error-correcting algorithm to fix possible invalid shares of $r_u^{(1)}, \ldots, r_u^{(q)}$, and then reconstruct the secret using a polynomial interpolation technique.
   (b) $r_v \leftarrow$ The result of running a Reed-Solomon error-correcting algorithm to fix possible invalid shares of $r_v^{(1)}, \ldots, r_v^{(q)}$, and then reconstructs the secret using a polynomial interpolation technique.
   (c) $r_w \leftarrow$ The result of running a Reed-Solomon error-correcting algorithm to fix possible invalid shares of $r_w^{(1)}, \ldots, r_w^{(q)}$, and then reconstructs the secret using a polynomial interpolation technique.
   (d) $y_v \leftarrow \widehat{y}_v - r_v$
   (e) $y_w \leftarrow \widehat{y}_w - r_w$
   (f) $\widehat{y}_u \leftarrow f_u(y_v, y_w) + r_u$
4. The functionality stores $\widehat{y}_u$ and sends it to all parties in Q_u, Q_v and Q_w.

once during the computation of the parent of u. Each time, they do not get enough shares of shares r_u to reconstruct any shares of r_u. But, can they combine the shares of shares from different runs for the same secret to gain some information? Since fresh and independent randomness was used by the dealers creating these shares on each run, the shares from each run are independent of the other runs, and so they do not collectively give any more information than each of the runs give separately. Since each run does not give the parties in $I_\triangle$ enough shares to reconstruct anything, it follows that they do not learn any information about r_u.

Second, parties in $I_\triangle$ add shares of shares for r_v and r_w to their views. However, with a similar argument as r_u, they cannot reconstruct r_v and r_w as well even if these parties participate in one or more of the instances of F_{CMPC} which involve v or w: the computation of v or w themselves or the computations of u as their parents.

Moreover, $\widehat{y}_u$ is also a random element in the field since r_u is uniformly random and $\widehat{y}_u = y_u + r_u$. Thus, $\widehat{y}_u$ holds no information about y_u, and the corrupted parties cannot learn any information about y_u except what is implicit in his input and the circuit output. This means that the corrupted parties cannot distinguish if they are participating in a run of the hybrid model or the simulation.

To prove the outputs added to the view of the adversary in real execution and simulation are equivalent, we show that protocol Circuit-Eval correctly computes $\widehat{y}_u = y_u + r_u$. Based on F_{Input} and $F_{\mathsf{Gen\text{-}Rand}}$, for each gate $u \in C$, the inputs of the honest parties in Q_u are enough to reconstruct r_u. If u is an input gate not included in the computation from the Input Commitment stage, then r_u and its shares are 0. Thus, all three values of r_u, r_v, and r_w can be correctly reconstructed by running a Reed-Solomon error-correcting algorithm to fix possible invalid shares of $r_w^{(1)}, \ldots, r_w^{(q)}$, which can tolerate up to a $1/3$ fraction of the inputs being invalid, and then reconstructs the secret using a polynomial interpolation.

We prove $\widehat{y}_u = y_u + r_u$ by induction on the height of u, where y_u is the correct output of the gate u. The base case is correct because based on the correctness of F_{Input}, for each input gate v', we have $\widehat{y}_{v'} = y_{v'} + r_{v'}$ and $r_{v'}$ can correctly be reconstructed from the inputs received from honest parties in $Q_{v'}$. Suppose that for all gates u' whose height is less than the height of u, the functionality can compute $\widehat{y}_{u'} = y_{u'} + r_{u'}$ and $r_{u'}$. This induction hypothesis is valid for v and w.

We now describe the induction step. In the computation of u, the functionality runs F_{CMPC}. We now argue based on the definition of the function computed by F_{CMPC} that the output of F_{CMPC} is $\widehat{y}_u = r_u + y_u$. By the induction hypothesis, the functionality can reconstruct correct r_v and r_w and consequently it can correctly find y_v and y_w even if a $1/3$ fraction of the inputs are missing. It is because the majority of the parties in Q_v and Q_w hold correct values of $\widehat{y}_v$ and $\widehat{y}_w$. Thus, the functionality can correctly compute $f_u(y_v, y_w) + r_u$.

Protocol 11 $\mathcal{S}_{\mathsf{Circuit\text{-}Eval}}$

For every gate $u \in C$ with children $v, w \in C$, consider three groups of parties Q_u, Q_v, and Q_w, each of whom has q parties. In each group, up to $q/3$ parties are corrupted.

Inputs. For each gate u, $\{\rho_i\}_{P_i \in I_u}$, and for each input gate v, values $\{\widehat{y}_v, r_v^{(i)}\}_{P_i \in I_v}$ from parties in I_u and I_v respectively.

Simulation: For every level in circuit C starting from level 1, and for every gate $u \in C$ with children $v, w \in C$ perform the following:

1. Run $F_{\mathsf{Gen\text{-}Rand}}$ with the following inputs: ρ_i for every $P_i \in I_u$ and a dummy input for every party in $Q_u - I_u$. Let $\{r_u^{(i)}\}_{P_i \in Q_u}$ denote the outputs. For every $P_i \in I_u$, store $r_u^{(i)}$ and add it to the view of P_i.

2. Let $Q_\triangle = Q_u \cup Q_v \cup Q_w$ and $I_\triangle = I_u \cup I_v \cup I_w$. Run F_{CMPC} to compute the functionality defined in Line 3 of $F_{\mathsf{Circuit\text{-}Eval}}$ with the following inputs: the input of every party in $I_\triangle$ (which is stored as the result of the computation for the child gate or input stage) as described in $F_{\mathsf{Circuit\text{-}Eval}}$, and a dummy input for every party in $Q_\triangle - I_\triangle$. Let $\widehat{y}_u$ denote the output. For every party in $I_\triangle$, store $\widehat{y}_u$ and add it to the view of the adversary.

Finally, since $\mathcal{S}_{\mathsf{Circuit\text{-}Eval}}$ is straight-line and black-box, and the protocol is perfectly-secure, Circuit-Eval is UC-secure. □ □

5.3.3 Security of Output Propagation

The ideal functionality for the Output Propagation stages of Protocol 1 are given in Protocol 12.

Lemma 4. *If the quorum formation stage of Algorithm 1 finished successfully, the Output Propagation stages of Protocol 1 securely UC-realize functionality F_{Output} in the synchronous model, with $1/3 - \epsilon$ malicious corruptions.*

Protocol 12 F_{Output}

Goal. For the output gate z, the functionality receives output y_z, and sends it to all parties $P_1, ..., P_n$.

Functionality:

1. Receive y_z as the input to the functionality.
2. Send y_z to all parties $P_1, ..., P_n$.

Proof. We first show by induction that after output propagation stage of Protocol 1, all honest parties eventually learn y_z. Since Q_1 is assigned to the output gate, it provides a base case. For $i > 1$, consider the parties in Q_i, and for all $j < i$ assume the correct output is learned by all parties in Q_j. During the Output Propagation stage, the parties in Q_i receive putative values for the output from the parties at $Q_{\lfloor i/2 \rfloor}$. Since $Q_{\lfloor i/2 \rfloor}$ is good, and by induction hypothesis all honest parties in it have learned the correct output, it follows that all honest parties in quorum $Q_{\lfloor i/2 \rfloor}$ send the same message, which is the correct output. By induction, all the parties learn the correct value.

To prove the t-security of the output propagation, the corresponding simulator $\mathcal{S}_{\mathsf{Output}}$ is given in Protocol 13.

Protocol 13 $\mathcal{S}_{\mathsf{Output}}$

Inputs. For the output gate z and the corresponding quorum Q_z, the inputs of the simulator is y_z.

Simulation:

1. For every $i \in \{2, ..., n\}$, parties in Q_i perform the following steps:
   (a) Receive y from $Q_{\lfloor i/2 \rfloor}$ and add it to the view of every parties in $I_{\lfloor i/2 \rfloor}$.
   (b) If $2i \leq n$, then send y to all parties in Q_{2i}. If $2i + 1 \leq n$, then send y to all parties in Q_{2i+1}.

The views of the corrupted parties in the hybrid execution and the simulation are indistinguishable since the only message that is added to the view of the adversary is the output y_z. Based on the security definition of MPC, the adversary is allowed to learn the output. □ □

5.3.4 Security of Protocol 1

We are now ready to put everything together and show that our main protocol can securely compute any n-ary function f. As the first step of the proof, we refer to Theorem 4 that shows quorum building stage of our MPC protocol can successfully build n quorums with high probability. For the remaining parts of the proof, we assume the quorum formation stage of Algorithm 1 finished successfully, i.e. all our quorums have more than $2q/3$ honest parties in them. The idea functionality of Protocol 1, F_{MPC} is described in Protocol 14.

Protocol 14 F_{MPC}

Goal. The functionality is parametrized by an arbitrary n-input function $f : \mathbb{F}^n \rightarrow \mathbb{F}^n$. The functionality interacts with n parties and the adversary.

Functionality:

1. For every input x_i of f, the functionality receives x_i from party P_i, where $i \in [n]$.
2. The functionality computes $y = f(x_1, x_2, ..., x_n)$ and sends y to all parties and adversary.

Lemma 5. *For every function $f : \mathbb{F}^n \rightarrow \mathbb{F}^n$ and $t < n/3 - \epsilon$, Protocol 1 is statically t-secure for F_{MPC} parameterized by f over private synchronous channels and in the presence of a static malicious adversary.*

Proof. We first prove the t-security of the Protocol 1 in the (F_{Input},$F_{\mathsf{Circuit\text{-}Eval}}$, F_{Output})-hybrid model, which is similar to Protocol 1 except that every subprotocols stages is replaced with a call to their functionalities respectively. The corresponding simulator $\mathcal{S}_{\mathsf{MPC}}$ is given in Protocol 15.

Protocol 15 $\mathcal{S}_{\mathsf{MPC}}$

For every $i \in [n]$, party P_i holds an input $x_i \in \mathbb{F}$. For every quorum Q_i Let I_i denote the set of corrupted parties in Q_i, and let I denote the set of all corrupted parties among $P_1, ..., P_n$.

Inputs. $\{r_i\}_{i\in[n]}$, and $\{\widehat{x_i}\}_{i\in[n]}$ from parties in I (set of all corrupted parties).

Simulation:

1. **Quorum Building.** Run Build-Quorums.
2. **Input Commitment.** (Equivalent to running $\mathcal{S}_{\mathsf{Input}}$)

   Run F_{Input} with the following inputs for each party $i \in n$,

   (a) If $P_i \in I$, obtain from the adversary x_i and the polynomial that it instructs P_i to send to the F_{Input} as a dealer of r_i.

   (b) If $P_i \notin I$, choose r_i and x_i uniformly at random from $\mathbb{F}$ and $\widehat{x_i} \leftarrow x_i + r_i$.

   Receive the outputs as the adversary expects from F_{Input}:

   (a) Send $\widehat{x_i}$ and a share of r_i to each party in Q_i.

   (b) For every party $P_j \in I_i$, add his share of r_i and $\widehat{x_i}$ to his view.

   (c) For each corrupted party $p_j \in I_i$, store x_i and his share of r_i he receives as the output of F_{Input}.
3. **Interaction with trusted party.** The simulator sends the trusted party computing f the inputs of the corrupted parties and receives from him the outputs $y_z = f(x_1, x_2, ..., x_n)$.
4. **Circuit Evaluation.** (Equivalent to running $\mathcal{S}_{\mathsf{Circuit\text{-}Eval}}$)

   Run $F_{\mathsf{Circuit\text{-}Eval}}$ with the following inputs:

   (a) For each input gate i and its corresponding quorum Q_i, use the stored value from input stage for corrupted parties $P_j \in I_i$. Use random values for other parties in Q_i.

   (b) For each gate u, obtain from the adversary the inputs of the corrupted party for generation of the random mask.

   Receive the outputs as the adversary expects from F_{Input}:

   (a) For each gate u, receive $\widehat{y_u}$ and a share of r_u. Add these values to the view of the adversary.

   (b) Let z be the output gate and Q_z its associated quorum, store $\widehat{y_z}$ and $r_z^{(i)}$ (a share of r_z) for corrupted party $P_i \in I_z$.
5. **Output reconstruction.**

   Let z be the output gate and Q_z its associated quorum. For every $i \in I_z$, let $\widehat{y_z}$ and $r_z^{(i)}$ be the stored values from circuit evaluation stage for corrupted party $P_i \in I_z$. Chooses a random polynomial for r_z under the constraint that the shares of it be compatible with $r_z^{(i)}$ and $y_z = \widehat{y_z} - r_z$, where y_z is the output of the evaluation received by the simulator from the trusted party computing f. Send shares of r_z to all parties in Q_z.
6. **Output Propagation**

   Run $F_{\mathsf{Output.}}$ to send y_z to all the parties p_i where $i \in n$.

The views of the corrupted parties in the hybrid execution and the simulation are indistinguishable for the input stage, and Circuit-Eval stage since the simulator only runs the functionality of F_{Input}, $F_{\mathsf{Circuit\text{-}Eval}}$. We now show that the output stage correctly computes F_{Output}. Let z be the output gate of C. In real execution, by Lemma 3, all parties in the output quorum Q_z eventually agree on $y_z + r_z$ and hold shares of r_z. In the Output Reconstruction stage, these parties reconstruct r_z. Since at least a $2/3$ fraction of them are honest, they correctly reconstruct r_z. Since all honest parties in Q_z know $y_z + r_z$ and subtract from it the reconstructed r_z, they all eventually learn y_z. Thus, all parties in Q_z eventually learn y_z, which is equal to the simulation's output. Finally, the simulator run F_{Output} so that all the parties can learn y_z. Thus, the only value that is added to the view of the adversary for both output stages are y_z and r_z. This completes the proof of t-security of our main protocol in the (F_{Input},$F_{\mathsf{Circuit\text{-}Eval}}$, F_{Output})-hybrid model.

In lemmas 1, 3, and 4 we proved that each stage is securely realize the functionality of that stage. Thus, our main protocol can securely realize F_{MPC}.

□

□

5.4 Proof of Theorem 2

We now show the security of our asynchronous MPC protocol. Recall that the asynchronous model, the adversary controls up to t parties as well as the scheduling of the messages.

Security of AVSS. We use the definition, scheme and proof of Ben-Or, Canetti and Goldreich [BCG93] for the asynchronous verifiable secret sharing and its reconstruction subprotocols (see Definition 8 and the AVSS scheme follows it). The paper introduces two subprotocols (V-Share, V-Recon), which we refer to them as (AVSS, ARecon) in our paper with functionalities F_{AVSS} and F_{ARecon}. Each good party that completes subprotocol AVSS (which is similar to the VSS scheme for sharing a secret but in the asynchronous model) subsequently invokes subprotocol ARecon with its local output of subprotocol AVSS as local input to reconstruct the secret.

Definition 3 (Definition 8 of [BCG93])**.** *(AVSS, ARecon) is a t-resilient AVSS scheme for n parties if the following holds, for every coalition of up to t corrupted parties.*

1. *Termination:*
   - *(a) If the dealer is honest, then every honest party will eventually complete subprotocol AVSS.*
   - *(b) If some honest party has completed subprotocol AVSS, then all the honest parties will finally complete subprotocol AVSS.*
   - *(c) If an honest party has finished subprotocol AVSS, then it will complete subprotocol ARecon.*
2. *Correctness: Once an honest party has completed subprotocol AVSS, then there exists a unique value, r, such that:*
   - *(a) All the honest parties output r. (Namely, r is the retrieved secret.)*
   - *(b) If the dealer is honest, sharing a secret s, then $r = s$.*
3. *Secrecy: If the dealer is honest and no honest party has begun executing subprotocol ARecon, then the corrupted parties have no information about the shared secret.*

We restate Theorem 2 of [BCG93] here without its proof.

Theorem 13 (Theorem 2 of [BCG93])**.** *Let $t < n/4$. Then, the pair (AVSS, ARecon) is a t-resilient AVSS scheme in the presence of a asynchronous malicious adversary.*

Security of ACMPC. We use the protocol and proof of Ben-Or, Canetti and Goldreich [BCG93] for the ACMPC protocol and denote the corresponding functionality by F_{ACMPC} as defined in Protocol 16.

Theorem 14 (Theorem 3 of [BCG93])**.** *For every function $g : \mathbb{F}^q \to \mathbb{F}^q$ and $T < q/4$, ACMPC securely realize F_{ACMPC} parameterized by g over private asynchronous channels and in the presence of a static malicious adversary that corrupt up to T parties.*

Protocol 16 F_{ACMPC}

Goal. The functionality is parametrized by an arbitrary q-input function $g : \mathbb{F}^q \to \mathbb{F}^q$. The functionality interacts with q parties and the adversary.

Functionality:

1. There exists a set S that contains $q - T$ parties.
2. For every input x_i of g where $P_i \in S$, the functionality receives x_i from party P_i, for all $i \in [q]$.
3. For every input x_i of g where $P_i \notin S$, the functionality receives sets x_i as a default value (0), for all $i \in [q]$.
4. The functionality computes $y = g(x_1, x_2, ..., x_q)$ and sends y to all parties and adversary.

5.4.1 Security of Input Commitment

Before proceeding to the proof of security for Input Commitment stage, we show the following auxiliary lemma.

Lemma 6. *If the quorum formation stage of our asynchronous MPC finished successfully, when a quorum Q sends to a quorum Q' a message M, it is eventually received by all honest parties in Q'.*

Proof. Recall that when Q sends M to Q', every honest party in Q sends M to all parties in Q'. A party in Q' considers itself to have received the message M from Q if it receives M from at least $7/8$ of the parties in Q. Since n quorums have successfully been formed, more than $7/8$ of the parties in each quorum are honest. In particular, this is true for Q. Thus, at least $7/8$ of the members of Q send M to each member of Q'. Since the adversary must eventually deliver all the messages that have been sent, albeit with arbitrary delays, it follows that eventually each honest party in Q' receives M from at least $7/8$ of the members of Q. □ □

We now proceed to the proof of the Input Commitment stage. The ideal functionality, $F_{\mathsf{Async\text{-}Input}}$, is given in Protocol 17. This functionality creates a set S containing the index of the parties whose inputs have been accepted (as defined in Step 1 of Protocol 6) by the protocol to be used for the computation. If a party's input is not in S, then the functionality sets this input to the default value. Next, the functionality sends each masked input $\widehat{x_i}$ to quorum Q_i and secret-shares the mask r_i in Q_i. In Lemma 7, we show the Input Commitment stage in Protocol 1 correctly implements this functionality. Thus, the parties in Q_i eventually either have received consistent VSS-shares of x_i and have agreed on $\widehat{x_i} = x_i + r_i$ as well as on i being in S or they have agreed that $i \notin S$ and have set these values to the predefined value and r_v and all its shares to 0. We say that a quorum has come to agreement on X if all honest parties in the quorum agree on X.

Protocol 17 $F_{\mathsf{Async\text{-}Input}}$

Goal. The functionality guarantees valid inputs are received from at least $n - t$ parties. Then, the functionality notifies the parties in all input quorums to proceed to the next stage of the protocol with either a valid input or a default input.

Functionality:

1. Wait to receive at least $n - t$ valid inputs from the set of all n parties. Let S denote the set of parties whose inputs have been accepted.
2. For each $i \in [n]$, if $P_i \in S$, the functionality receives $\widehat{x_i} = x_i + r_i$ and r_i from party P_i. If $P_i \notin S$, then define $\widehat{y_i} = 0$ and $r_i = 0$.
3. For each $i \in [n]$, the functionality broadcasts $\langle \mathsf{Done} \rangle$ and $y_i \leftarrow x_i + r_i$ to all parties in Q_i
4. For each $i \in [n]$, the functionality shares r_i among q parties of Q_i by choosing a random polynomial $h(x)$ of degree T such that $h(0) = r_i$. For all $j \in [q]$, the functionality sends $h(j)$ to the j-th party in Q_i.

Lemma 7. *If algorithms* Build-Quorums *and* Thresh-Count *finished successfully, the Input Commitment stage of our asynchronous MPC securely UC-realizes functionality $F_{\mathsf{Async\text{-}Input}}$ with $1/8 - \epsilon$ malicious corruptions.*

Proof. We prove the t-security of the Input Commitment stage in the (F_{AVSS},F_{ACMPC})-hybrid model, which is similar to the Input Commitment stage of asynchronous MPC protocol except that every call to its subprotocols is replaced with a call to their corresponding functionality. We define the corresponding simulator $\mathcal{S}_{\mathsf{Async\text{-}Input}}$ in Protocol 18.

Protocol 18 $\mathcal{S}_{\mathsf{Async\text{-}Input}}$

For every $i \in [n]$, party P_i holds an input $x_i \in \mathbb{F}$. Associated with this input, we consider a quorum Q_i. Let I_i denote the set of corrupted parties in Q_i, and let I denote the set of all corrupted parties among $P_1, ..., P_n$.

Inputs. $\{r_i\}_{i\in[n]}$, and $\{\widehat{x_i}\}_{i\in[n]}$ from parties in I (set of all corrupted parties).

Simulation:

1. For every $i \in [n]$,
   (a) If $P_i \in I$, obtain from the adversary x_i and the polynomial that it instructs P_i to send to the F_{VSS} as a dealer of r_i.
   (b) If $P_i \notin I$, choose r_i and x_i uniformly at random from $\mathbb{F}$ and $\widehat{x_i} \leftarrow x_i + r_i$.
   (c) Broadcast $\widehat{x_i}$ to all parties in Q_i and run F_{VSS} to secret-share r_i in Q_i.
   (d) For every party in I_i, add his share of r_i (as it expects from the F_{VSS} invocations) and $\widehat{x_i}$ to his view.
2. For every party in Q_i, run Wait-For-Inputs to wait for at least $n-t$ inputs.
   (a) Run $F_{\mathsf{Thresh\text{-}Count}}$ with flag b_i initially set to zero to count the number of received inputs.
   (b) If x_i and r_i are valid and consistent (based on the broadcast protocol and the verification stage of Reconst respectively), raise an event to set $b_i \leftarrow 1$ in $F_{\mathsf{Thresh\text{-}Count}}$.
   (c) Upon receiving $\langle \mathsf{Done} \rangle$ from the parent quorum, run F_{ACMPC} using b_i as the input to count the number 1 inputs. If the result of F_{ACMPC} is less than $5q/8$, then $\widehat{x} \leftarrow \mathsf{Default}$ and $r_i \leftarrow 0$.

Let V_1 denote the view of the adversary from the hybrid execution, and V_2 be its view from the simulation. The inputs to Thresh-Count and Line 5 of Protocol 4 are completely independent of the inputs of Protocol 1. Thus, V_1 contains only the masked inputs, $\widehat{x_i}$'s, and at most 1/8 fraction of the shares for each random mask, r_i's. The masked inputs convey no information about the inputs. Moreover, a 1/8 fraction of the shares are not enough to reconstruct the random number. Since V_2 contains all random elements, the adversary cannot distinguish V_1 from V_2.

Now, we show that corrupted parties cannot do anything but choose their input as they wish; thus, the Input Commitment stage correctly computes $F_{\mathsf{Async\text{-}Input}}$ and its output is as same as the simulation. This means that all honest parties receive the $\langle \mathsf{Done} \rangle$ message. Moreover, there exists a set S such that for every $i \in [n]$, the following statements hold:

1. All parties in Q_i eventually agree whether $i \in S$ or not.
2. At least $n-t$ input quorums agree that their corresponding party's index is in S.
3. All parties in Q_i agree that party $i \in S$ if and only if they collectively hold enough shares to reconstruct P_i's input. If all parties in Q_i agree that $i \in S$, then party P_i's input will be used in the computation. Otherwise, the default value will be used instead.

First, since there are $n-t$ honest parties, at least $n-t$ valid inputs are eventually sent to Thresh-Count (property (1-a) of $t-resilience$ AVSS). Based on Theorem 3, all parties will be notified when $n-t$ inputs are received.

Each party in Q_i has set his flag bit to either 1 or 0 depending on whether it has received a valid input share from P_i. Let $q = |Q_i|$. Upon receiving the $\langle \mathsf{Done} \rangle$ message, the parties in Q_i run the third step of Wait-For-Inputs to decide whether at least $\frac{5q}{8}$ of them have set their flag bit to 1. If they have, they assume $i \in S$.

If $i \in S$, then at least $\frac{7q}{8}$ of the parties in Q_i have received input shares from P_i before they received the $\langle \mathsf{Done} \rangle$ message. Of these, more than $\frac{3q}{4}$ parties in Q_i are honest and have set their flag bit to 1. Since ACMPC in Line 5 starts even if as many as $q/8$ inputs are missing, the parties in Q_i will correctly decide that at least $\frac{5q}{8}$ flag bits among them are set to 1. Thus, the parties in Q_i all agree that $i \in S$. If $i \notin S$, then ACMPC in Line 5 has determined that less than $\frac{5q}{8}$ flag bits are set to 1. Since Q_i contains less than $q/8$ corrupted parties, more than $q/2$ parties set their flags to 0 and the parties in Q_i all agree that $i \notin S$. As a result, at least $n-t$ input quorums agree that their corresponding inputs are in S, and hence $|S| \geq n-t$.

We now show that with $T < q/8$ corrupted parties in each quorum, the parties in each input quorum will hold sufficient shares to keep the secret input uniquely reconstructible. Consider a single input quorum Q with q parties and T corrupted parties. The quorum runs the Wait-for-Input protocol as stated in Section 4.2. Every party P_i in Q holds a bit b_i that is initially set to 0. We have two possibilities: (1) the quorum corresponds to a consistent input, or (2) the quorum corresponds to an input that is going to be set to the default. We are only interested in the first case. In this case, the parties in Q will eventually set their flag bits (i.e., b_i's) to 1 as in Line 3 of Wait-for-Input. Since messages may be delayed indefinitely and T parties are corrupted, we can only guarantee that at most $q-2T$ parties set their flag bit to 1. Thus in the worst case, $2T$ parties participate in ACMPC with bit 0, where half of them have failed to set their bits to 1 due to asynchrony, and the other half are lying. In order to keep the secret reconstructible from its shares, we need at least $3q/4$ shares. Therefore, $q-2T > 3q/4$ and hence $T < q/8$.

Since our simulator is straight-line and black-box, and the protocol is perfectly secure, it follows from Theorem 10 that the Input Commitment stage is UC-secure. □ □

5.4.2 Security of Circuit Evaluation

Similar to the synchronous case we start with security proof of Async-Gen-Rand. The ideal functionality $F_{\text{Gen-Rand}}$ is similar to the synchronous version as described in Protocol 8.

Lemma 8. *If algorithm Build-Quorums finished successfully, the protocol Async-Gen-Rand is securely UC-realizes functionality $F_{\text{Gen-Rand}}$ in the asynchronous model, with $1/8-\epsilon$ malicious corruptions.*

Proof. We prove the t-security of Gen-Rand in the F_{ACMPC}-hybrid model, which is similar to Protocol Async-Gen-Rand except that every call to ACMPC is replaced with a call to the ideal functionality F_{ACMPC}. The corresponding simulator $\mathcal{S}_{\text{Async-Gen-Rand}}$ is given in Protocol 19.

Protocol 19 $\mathcal{S}_{\text{Async-Gen-Rand}}$

Inputs. For a gate $u \in C$, the inputs $\{\rho_j\}_{P_j \in I_u}$ of the corrupted parties $P_1, \ldots, P_q$ in the quorum associated with u.

Simulation: For every $P_i \in (Q_u)$ run F_{ACMPC} to compute function $r_u = \sum_{k=1}^{q} \rho_j$ up to the with the following inputs,

1. If $P_i \in (Q_u - I_u)$ (i.e., for every honest party P_i), call F_{ACMPC} with dummy input, set $\rho_j = 0$.
2. For every $P_j \in I_u$, call F_{ACMPC} with input ρ_j.
3. Add $r_u^{(j)}$ to the view of P_j.

The views of the corrupted parties in the hybrid execution and the simulation are indistinguishable because both protocols simply call the F_{FCMPC} functionality. Since F_{ACMPC} correctly computes the sum of the inputs as the output, the two views are identically distributed. Since our simulator is straight-line and black-box, and the protocol is perfectly secure, Gen-Rand is UC-secure. □ □

We now proceed to the security proof of Async-Circuit-Eval in the asynchronous model. The ideal functionality $F_{\text{Async-Circuit-Eval}}$ is same as synchronous case that is given in Protocol 10.

Lemma 9. *If algorithms Build-Quorums and Thresh-Count finished successfully, the protocol Async-Circuit-Eval securely UC-realizes functionality $F_{\text{Async-Circuit-Eval}}$ in the asynchronous model, with $1/8-\epsilon$ malicious corruptions.*

Proof. We prove the t-security of Async-Circuit-Eval in the ($F_{\text{Async-Gen-Rand}}$,$F_{\text{ACMPC}}$)-hybrid model, which is similar to Protocol 2 except that every call to ACMPC and Async-Gen-Rand is replaced with a call to F_{ACMPC} and $F_{\text{Async-Gen-Rand}}$ respectively. This proof is completely similar to the synchronous case and we can use a simulator similar to $\mathcal{S}_{\text{Circuit-Eval}}$ that is given in Protocol 11. The only difference is that the simulator calls the asynchronous version of the functionalities.

Since the proof is similar, we do not repeat the whole proof here and only describe that the output of the honest parties are identically distributed in the real execution and the simulation. We prove this by induction on the height of u. The induction hypothesis is that $\widehat{y}_u = y_u + r_u$ where y_u is the correct output of gate u. The base case is true based on F_{AInput}.

Suppose the induction hypothesis is valid for v and w that are children of u. In the computation of u, the hybrid execution runs F_{ACMPC}. We now argue based on the definition of the function computed by F_{ACMPC} that the output of F_{ACMPC} is $\widehat{y}_u = r_u + y_u$. By the induction hypothesis, the functionality F_{ACMPC} can reconstruct correct r_v and r_w and consequently it can correctly find y_v and y_w even if a $1/8$ fraction of the inputs are missing. It is because the $7/8$ fraction of the parties in Q_v and Q_w hold correct values of $\widehat{y}_v$ and $\widehat{y}_w$. Thus, even if another $1/8$ fraction of the input shares set to default value when running F_{AMPC} due to the asynchronous nature of the communication, there is still enough shares correct (6/8 fraction). In other words, the shared output will represent a valid Reed-Solomon codeword with at least a 2/3 fraction valid shares for decryption. Hence, ACMPC has enough redundancy needed for correctly reconstructing the output. error correct the shares of corrupted parties Thus, F_{ACMPC} can correctly compute $f_u(y_v, y_w) + r_u$.

The t-security of Async-Circuit-Eval is obtained by combining Lemma 8 (securely computing $F_{\mathsf{Async\text{-}Gen\text{-}Rand}}$), Theorem 14 (securely computing F_{ACMPC}) and the fact that we showed the t-security of Circuit-Eval in the ($F_{\mathsf{Async\text{-}Gen\text{-}Rand}}$,$F_{\mathsf{ACMPC}}$)-hybrid model, and using the modular sequential composition theorem. □ □

5.4.3 Security of Output Propagation

The ideal functionality for the Output Propagation stages of our asynchronous MPC protocol is similar to its synchronous version, which is given in Protocol 12. Since its security proof and simulator are also similar to the synchronous case, we omit the proof and only state the lemma.

Lemma 10. *The Output Propagation stage of our asynchronous MPC protocol securely UC-realize functionality F_{Output} in the asynchronous model, with $1/8 - \epsilon$ malicious corruptions.*

5.4.4 Security of Protocol 1

We are now ready to put everything together and show that our asynchronous MPC protocol can securely compute any n-ary function f. The idea functionality of Protocol 1, F_{AMPC} is described in Protocol 20.

Protocol 20 F_{AMPC}

Goal. The functionality is parametrized by an arbitrary n-input function $f : \mathbb{F}^n \to \mathbb{F}^n$. The functionality interacts with n parties and the adversary.

Functionality:

1. There exists a set S that contains $n - t$ parties.
2. For every input x_i of f where $P_i \in S$, the functionality receives x_i from party P_i, where $i \in [n]$.
3. For every input x_i of f where $P_i \notin S$, the functionality sets x_i a default value.
4. The functionality computes $y = f(x_1, x_2, ..., x_n)$ and sends y to all parties and adversary.

Lemma 11. *For every function $f : \mathbb{F}^n \to \mathbb{F}^n$ and $t < n/8 - \epsilon$, our asynchronous MPC protocol is t-secure for F_{AMPC} parameterized by f over private synchronous channels and in the presence of a static malicious adversary.*

Proof. We prove the t-security of our asynchronous MPC protocol in the ($F_{\mathsf{Async\text{-}Input}}$, $F_{\mathsf{Async\text{-}Circuit\text{-}Eval}}$, $F_{\mathsf{Async\text{-}Output}}$)-hybrid model, which is similar to Protocol 1 except that every subprotocols stages is replaced with a call to their functionalities respectively. The corresponding simulator is similar to $\mathcal{S}_{\mathsf{MPC}}$ that is given in Protocol 15 The only difference is that every call to a functionalities is replaced with its asynchronous version.

The views of the corrupted parties in the hybrid execution and the simulation are indistinguishable for the input stage and Circuit-Eval stage since the simulator simply run the functionality of $F_{\mathsf{Async\text{-}Input}}$, $F_{\mathsf{Async\text{-}Circuit\text{-}Eval}}$. Note that in the asynchronous version, the functionality $F_{\textsf{Async-Input}}$ guarantees to generate the set S for $n-t$ valid inputs before $F_{\textsf{Async-Circuit-Eval}}$ starts. We now show that the output stage correctly compute the output so that the output of hybrid execution is similar to the simulation. Let z be the output gate of C. In hybrid execution, by Lemma 9, all parties in the output quorum Q_z eventually agree on $y_z + r_z$ and hold shares of r_z. In the Output Reconstruction stage, these parties reconstruct r_z. Since at least a $7/8$ fraction of them are honest, they correctly reconstruct r_z. Since all honest parties in Q_z know $y_z + r_z$ and subtract from it the reconstructed r_z, they all eventually learn y_z. Thus, all parties in Q_z eventually learn y_z, which is equal to the simulation's output. Finally, the simulator run $F_{\textsf{Async-Output}}$ so that all the parties can learn y_z. Thus, the only value that is added to the view of the adversary for both output stages are y_z and r_z. This completes the proof of t-security of our main protocol in the ($F_{\textsf{Async-Input}}$,$F_{\textsf{Async-Circuit-Eval}}$, $F_{\textsf{Async-Output}}$)-hybrid model.

In lemmas 7, 9, and 10 we proved that each stage is securely realize the functionality of that stage. Thus, our main protocol can securely realize $F_{\textsf{MPC}}$. □ □

5.5 Cost Analysis

In this section, we analyze the resource costs of our protocols.

Lemma 12. *During the Input Commitment stage in our both MPC protocols, each quorum sends at most* $\textsf{polylog}(n)$ *messages and computes at most* $\textsf{polylog}(n)$ *operations.*

Proof. In our both protocols, since each quorum has $O(\log n)$ parties, $\textsf{polylog}(n)$ messages are sent by each quorum during VSS and Reconst to check whether the input is correctly secret-shared. In our asynchronous protocol, each quorum is mapped to at most one of the input gates and hence one of the nodes in the count tree. Thus, from Theorem 3 it follows that the total number of messages sent by each quorum is $O(\log n)$. Therefore in our both protocols, each quorums sends at most $\textsf{polylog}(n)$ messages. Since the computation costs of VSS and Reconst are polynomial in their input size, each quorums computes at most $\textsf{polylog}(n)$ operations. □ □

Lemma 13. *If all honest parties follow our protocols, then with high probability, each party sends at most* $\tilde{O}(m/n+\sqrt{n})$ *messages and computes at most* $\tilde{O}(m/n+\sqrt{n})$ *operations.*

Proof. By Theorem 4, we need to send $\tilde{O}(\sqrt{n})$ messages per party to build the quorums. Subsequently, each party must send messages for each quorum in which it is a member. Recall that each party is in $\Theta(\log n)$ quorums.

By Lemma 12, each quorum sends at most $\textsf{polylog}(n)$ messages during Input stage. Recall that each quorum is mapped to $\Theta(\frac{m+n}{n})$ nodes of C. A quorum runs Gen-Rand and the gate evaluation step of Circuit-Eval once per node it is mapped to in C. Since each gate has in-degree two and out-degree at most two, a quorum runs CMPC at most three times for every node it is mapped to in C. Also, at most $\textsf{polylog}(n)$ messages are sent per party per instance of CMPC, Gen-Rand, and gate evaluation. Finally, each quorum sends/computes $O(\log n)$ messages/operations in the dissemination of the output. Thus, each quorum sends/computes $\textsf{polylog}(n)$ messages/operations per node it represents. Using a similar argument for our asynchronous protocol, we conclude that in our both MPC protocols each party sends/computes $\tilde{O}(m/n+\sqrt{n})$ messages/operations. □ □

Lemma 14. *The total latency of our synchronous MPC protocol is* $O(d\,\textsf{polylog}(n))$*, where d is depth of the circuit the protocol computes.*

Proof. Based on Theorem 4, the latency for creating quorums is $\textsf{polylog}(n)$. Since the protocol evaluates the circuit level by level and the circuit has d levels, the total latency is d times the latency of CMPC over $\log n$ parties, which is $O(d\,\textsf{polylog}(n))$. □ □

Lemma 15. *The expected latency of our asynchronous MPC protocol is* $O(d\,\textsf{polylog}(n))$*, where d is depth of the circuit the protocol computes.*

Proof. We bounded the expected latency of our asynchronous MPC protocol in Section 4.2.2. □ □

6 Asynchronous Threshold Counting

In this section, we prove Theorem 3 by constructing a protocol called Thresh-Count for solving the threshold counting problem as defined in 2. In our asynchronous MPC protocol presented in Section 4, we run Thresh-Count among a set of quorums, where each quorum represents an honest party.

Recall that in the threshold counting problem there are n honest parties in an asynchronous communication network with private channels. Each party has an input flag, which is initially 0. At least τ of the parties' bits will eventually be set to 1 based on an external event. When this happens, we say the threshold is reached. The goal is for each of the parties to terminate at some time after the threshold is reached.

Although in our application τ is linear in n, we address the more general case, where $\tau = O(n)$. Our algorithm depends on prior knowledge of τ. As specified in Theorem 3, each party running the algorithm sends and receives $O(\log n)$ messages of constant size and performs $O(\log n)$ computations; moreover the total latency is $O(\log n)$.

For ease of presentation, we first describe an algorithm which works when $\tau = \Theta(n)$, in particular, when τ is at least $n/2$. We then indicate why this fails when τ is smaller, and show how to modify it so that it works for all τ. The formal algorithm is shown as Protocol 21.

Consider a complete binary tree, where each party sends his input to a unique leaf node when it is set to 1. Then, for every node v, each child of v sends v a message giving the number of inputs it has received so far and it sends a new message every time this number changes. The problem with this approach is that it is not load-balanced: each node at depth i has $n/2^i$ descendants in the tree, and therefore, in the worst case, sends and receives $n/2^i$ messages. Thus, a child of the root sends $n/2$ messages to the root and receives the same number of messages from its children.

One may argue why a simple modification to the above algorithm cannot solve the load-balancing issue. For example, assume a single binary tree with τ leaves, where each node passes up to his parent only after receiving messages from both children. Unfortunately, deterministic algorithms such as this cannot solve the threshold counting problem in a load-balanced manner: The adversary can carefully but simply choose the nodes he wants to send his inputs in a way to make sure the threshold is never met and the algorithm always fails. For the example mentioned above, it is enough for the adversary to choose nodes with honest siblings. The adversary never sends his own inputs, but since the nodes of the tree are waiting for both inputs, the algorithm always fails to identify the threshold. In our algorithm, we use randomness in our favor to make sure the adversary cannot corrupt the counting process and the protocol succeeds with high probability. However, there is a small probability of error in which the threshold will not be identified. We added this discussion to the paper.

To solve the load-balancing problem, we use a randomized approach which ensures with high probability that each leaf of the data structure receives at least $7 \log n$ messages and does not communicate with its parent until it has done so. Subsequent messages it receives are not forwarded to its parent but rather to other randomly chosen leaves to ensure a close to uniform distribution of the messages.

Our algorithm consists of up and down stages. For the up stage the parties are arranged in a predetermined tree data structure, which we call the *count tree*. The count tree consists of a root node with $O(\log n)$ children, each of which is itself the root of a complete binary tree; these subtrees have varying depths as depicted in Figure 4. In the up stage, parties in the trees count the number of 1-inputs, i.e., the number of parties' inputs that are set to **1**. The root then eventually decides when the threshold is reached. In the down stage, the root notifies all the parties of this event via a complete binary tree of depth $\log n$. Note that the tree used in the down stage has the same root as the count tree.

Let $D = \lceil \log \frac{\tau}{14 \log n} \rceil$. Note that $D = O(\log n)$. The root of the count tree has degree D. Each of the D children of the root is itself the root of a complete binary subtree, which we will call a *collection subtree*. For $1 \leq j \leq D$, the jth collection subtree has depth $D + 1 - j$. Party 1 is assigned to the root and parties 2 to $D + 1$, are assigned to its children, i.e., the roots of the collection subtrees, with party $j + 1$ being assigned to the jth child. The remaining nodes of the collection trees are assigned parties in order, starting with $D + 2$, left to right and top to bottom. One can easily see that the entire data structure has fewer than n nodes, (in fact it has fewer than $\frac{\tau}{3 \log n}$ nodes) so some parties will not be assigned to any node. The leaves of each collection subtree are *collection nodes*, while the internal nodes of each collection tree are *adding nodes*.

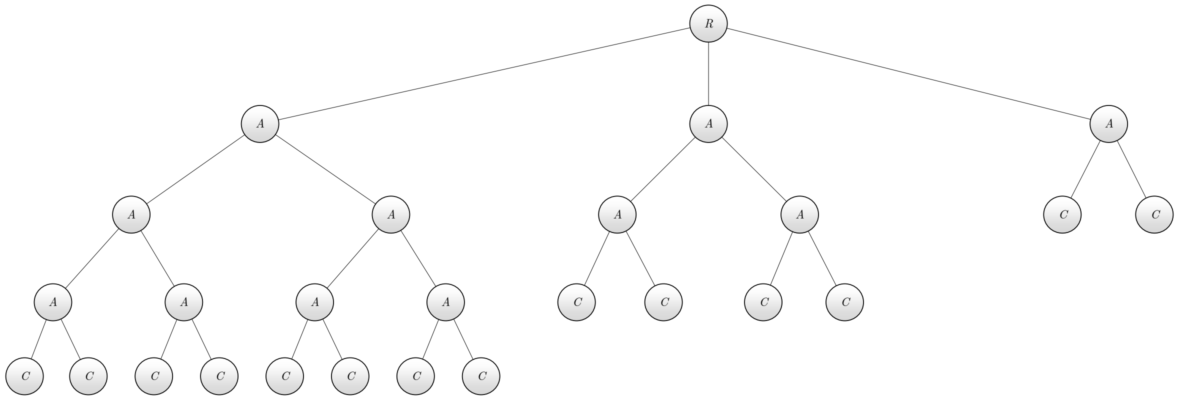

Figure 4: The count tree for $n = 2048$ and $\tau = 1232$. $D = \lceil \log \frac{1232}{14 \times 11} \rceil = 3$. The node marked R is the root, nodes marked A are adding nodes, and nodes marked C are collection nodes.

6.1 Up Stage

When a party's input is set to **1**, it sends a ⟨Flag⟩ message, which we will sometimes simply refer to as a flag, to a uniformly random collection node from the first collection subtree. Intuitively, we want the flags to be distributed as evenly as possible among the collection nodes. The parameters of the algorithm are set up so that with high probability each collection node receives at least $7 \log n$ ⟨Flag⟩ messages.

Each collection node in the j-th collection tree waits until it has received $7 \log n$ flags. It then sends its parent a ⟨Count⟩ message. For each additional flag received, up to $14 \log n$, it chooses a uniformly random collection node in the $(j+1)$-st collection subtree and forwards a flag to it. If $j = D$, then it forwards these $14 \log n$ flags directly to the root. Subsequent flags are ignored. Again, we use the randomness to ensure a close to even distribution of flags with high probability.

Each adding node waits until it has received a ⟨Count⟩ message from each of its children. Then, it sends a ⟨Count⟩ message to its parent. We note that, with high probability, each adding node sends exactly one message during the algorithm. The parameters of the algorithm are arranged so that all the ⟨Count⟩ messages that are sent in the the jth collection subtree together account for $\tau/2^j$ of the 1-inputs. Thus, all the ⟨Count⟩ messages in all the collection subtrees together account for $\tau\left(1 - \frac{1}{2^D}\right)$ of the 1-inputs. At least $\frac{\tau}{2^D}$ 1-inputs remain unaccounted for. These 1-inputs and up to $O(\log n)$ more are collected as flags at the root.

6.2 Down Stage

When party 1, at the root, has accounted for at least τ 1-inputs, it starts the down stage by sending the ⟨Done⟩ message to parties 2 and 3. For $j > 1$, when party j receives the ⟨Done⟩ message, it forwards this message to parties $2j$ and $2j + 1$. Thus, eventually, the ⟨Done⟩ message reaches all the parties, who then learn that the threshold has been met.

Note that all three types of messages sent in this protocol, ⟨Flag⟩, ⟨Count⟩ and ⟨Done⟩, are notifications only; they do not contain any numerical value. Since 2 bits are sufficient to distinguish three different kinds of messages, all the messages sent in this protocol are 2-bit strings. Note that we distinguish between flags and ⟨Count⟩ messages since the root receives both types. However, it is the only node for which this is a problem. We could add another node, as the $(D+1)$st child of the root, (equivalently as a collection subtree of depth 0), which waits for $14 \log n$ messages, and sends a ⟨Count⟩ message to the root. In so doing, we could eliminate the need to distinguish ⟨Flag⟩ and ⟨Count⟩ message explicitly, since they would be automatically identified by the role of the receiving node. Thus, we could actually reduce all message lengths to a single bit.

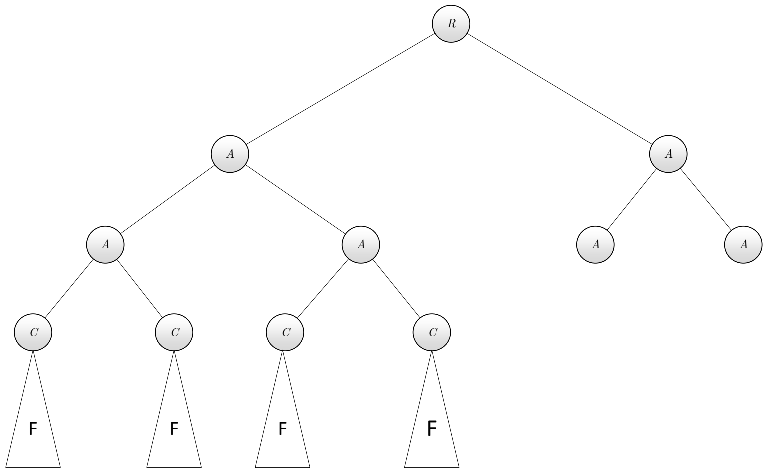

Figure 5: The count tree for $n = 2048$ and $\tau = 616$. $D = \lceil \log \frac{616}{14 \times 11} \rceil = 2$. The node marked R is the root, nodes marked A are adding nodes and nodes marked C are collection nodes. The filters, marked F, are complete binary trees of depth 7, with 128 leaves each, for a total of 512 filter leaves.

6.3 Handling Sublinear Thresholds

Now, we consider the case where $\tau = o(n)$. It is easy to see that the worst load in terms of the number of received messages is when all n inputs are 1. In this case, a collection node in the first collection subtree receives, on average, $14(n/\tau)\log n$ flags. When $\tau = \Theta(n)$, this is still $O(\log n)$, but when $\tau = o(n)$ this is $\omega(\log n)$. Before we describe how to fix this, we note that the problem exists *only* in the leaves of the *first* collection subtree. Subsequent collection nodes receive only $O(\log n)$ flags, because each node only forwards up to $14\log n$ flags.

For the sake of having a definite cutoff and tractable constants, we will apply the following fix whenever $\tau < n/2$. Below each collection node in the first collection tree, we put in a *filter*, which is a complete binary tree of depth $\log n - 2 - D$ with $\frac{7n\log n}{2\tau}$ leaves. This is equivalent to extending the first collection tree to depth $\log n - 2$ so that it has $n/4$ leaves. The collection nodes will remain at depth D though. See Figure 5.

When a party's input is set to 1, it selects a random collection node in the first collection tree, but rather than sending a flag directly to it, it sends the flag to a random leaf of the collection node's filter. The nodes in the filter simply forward any flags they receive, up to $21\log n$, to their parent in the filter. Subsequent flags are ignored. Apparently, this means that the collection node at the root of the filter cannot receive more than $42\log n$ flags, which solves the load problem. Moreover, we have not only transferred the problem to the leaves of the filter. Since there are so many more of them, each one actually receives fewer flags on average and the parameters are adjusted to make their maximum load $O(\log n)$ with high probability. As we will also see in the analysis, these filters do not filter out too many flags; when there are only τ 1-inputs among the parties, with high probability all the flags get through.

6.4 Proof of Theorem 3

In this section, we prove the correctness and resource costs of Protocol 21. The process of each party independently selecting a random collection node to notify after his input has been set to **1** can be modeled as a balls and bins problem and hence be approximated by the Poisson distribution.

6.4.1 Preliminaries

We first recall the following Chernoff bound for a Poisson random variable from Mitzenmacher and Upfal [MU05].

Theorem 15 (Theorem 5.4 of [MU05])**.** *Let $Y \sim Poisson(\mu)$. Then,*

Protocol 21 Thresh-Count

Goal. n is the number of parties, τ is the threshold, b is a flag bit initially set to zero, which may be set to one by an external event throughout the protocol and $D = \lceil \log(\frac{\tau}{14 \log n}) \rceil$. The algorithm notifies all the parties upon receiving τ flag bits set to one.

1. **Setup.** No messages sent in this stage:
   (a) Build the count tree and set party 1 as the root:
   For $1 \leq j \leq D$, party $j+1$ is a child of the root (and the root of the jth collection subtree with depth $D+1-j$). Starting with party $D+2$, the remainder of the nodes are assigned to parties, left to right and top to bottom. If $\tau < n/2$ the remaining parties are assigned to filters, left to right and top to bottom.
   (b) Let $sum = 0$ for the root.
2. **Up Stage.**
   (a) Upon receiving event ⟨Ready⟩, choose a uniformly random collection node v from collection subtree 1,
   - If $\tau > n/2$, then send a ⟨Flag⟩ to v.
   - Otherwise, choose a uniformly random leaf in v's filter and send a ⟨Flag⟩ to it.

   (b) Upon receiving a ⟨Flag⟩, if previously forwarded fewer than $21 \log n$ flags, forward the flag to parent. Otherwise, ignore it.
   (c) Perform the following steps to collect nodes in the collection subtree j:
   - Upon receiving $7 \log n$ ⟨Flag⟩s, send parent a ⟨Count⟩ message.
   - Upon subsequently receiving a ⟨Flag⟩, if $j < D$, send it to a uniformly random collection node in collection subtree $j+1$. If $j = D$, then send it directly to the root. Do this for up to $14 \log n$ flags. Then, ignore all subsequent ⟨Flag⟩ messages.

   (d) Upon receiving ⟨Count⟩ from both children, send ⟨Count⟩ to the parent.
   (e) If $sum < \tau$, then
   - Upon receiving a ⟨Count⟩ from party $j+1$, set $sum \leftarrow sum + \tau/2^j$.
   - Upon receiving a ⟨Flag⟩, $sum \leftarrow sum + 1$.
3. **Down Stage.** If $sum \geq \tau$, then
   (a) Party 1 (the root): Send ⟨Done⟩ to parties 2 and 3, and then terminate.
   (b) Party j for $j > 1$: Upon receiving ⟨Done⟩ from party $\lfloor j/2 \rfloor$, forward it to parties $2j$ and $2j+1$ (if they exist), and then terminate.

1. *for* $x < \mu$, $\text{Prob}(Y \le x) \le e^{-\mu}(e\mu/x)^x$, *and*

2. *for* $x > \mu$, $\text{Prob}(Y \ge x) \le e^{-\mu}(e\mu/x)^x$.

Lemma 16. *Assume αk balls are thrown independently and uniformly at random into k bins. Let E_1 denote the event that the minimum load is less than $\alpha/2$, and let E_2 denote the event that the maximum load exceeds $3\alpha/2$. Then,*

$$\text{Prob}(E_1) \le ek\sqrt{\alpha k}\left(\frac{2}{e}\right)^{\alpha/2} \tag{1}$$

and

$$\text{Prob}(E_2) \le ek\sqrt{\alpha k}\left(\frac{8e}{27}\right)^{\alpha/2}. \tag{2}$$

Proof. For $1 \le i \le k$, let X_i denote the number of balls in the ith bin, and let $Y_i \sim \text{Poisson}(\alpha)$ be an independent Poisson random variable with mean α. It is well known that the distribution of each X_i is close to that of Y_i, and moreover that the *joint* distribution of the X_i's is well approximated by the joint (i.e., product) distribution of the Y_i's (see Chapter 5 in Mitzenmacher and Upfal [MU05]). Indeed, Corollary 5.11 from [MU05] states that for any event E that is monotone in the number of balls, if E occurs with probability at most p in the Poisson approximation, then E occurs with probability at most $2p$ in the exact case. Since maximum and minimum load are both clearly monotone increasing in the number of balls, applying this corollary we have:

$$\begin{aligned}\text{Prob}(E_1) &= \text{Prob}\,(\exists i \text{ s.t. } X_i \le \alpha/2)\\ &\le 2\,\text{Prob}\,(\exists i \text{ s.t. } Y_i \le \alpha/2)\\ &\le 2\sum_{i=1}^{k}\text{Prob}\left(Y_i \le \frac{\alpha}{2}\right)\\ &\le 2k\left(\frac{2}{e}\right)^{\alpha/2},\end{aligned}$$

where the last inequality follows from Theorem 15 with $\mu = \alpha$ and $x = \alpha/2$. Similarly,

$$\begin{aligned}\text{Prob}(E_2) &= \text{Prob}\,(\exists i \text{ s.t. } X_i > 3\alpha/2)\\ &\le 2\,\text{Prob}\,(\exists i \text{ s.t. } Y_i > 3\alpha/2)\\ &\le 2\sum_{i=1}^{k}\text{Prob}\left(Y_i \ge \frac{3\alpha}{2}\right)\\ &\le 2k\left(\frac{8e}{27}\right)^{\alpha/2},\end{aligned}$$

where the last inequality follows from Theorem 15 with $\mu = \alpha$ and $x = 3\alpha/2$. □ □

6.4.2 Protocol Analysis

Let σ be the number of 1-inputs. We know that $\tau \le \sigma \le n$. Let $s = \sigma/\tau$. For simplicity of the analysis, we will assume that the first τ flags to be sent are *marked* while the remaining $\sigma - \tau$ are *unmarked*. As we track the progress of the flags through our data structure, we pay particular attention to the marked flags. Due to asynchrony, the marked flags need not be the first τ to arrive at their destinations.

Lemma 17. *Suppose $\tau \ge n/2$. In the Thresh-Count algorithm, with probability at least $1 - \frac{1}{7n\log n}$, the first collection subtree satisfies all of the following:*

1. *Each collection node receives between $7s\log n$ and $21s\log n$ flags.*

2. *The* ⟨Count⟩ *messages generated in this tree, when they reach the root, account for $\tau/2$ 1-inputs.*

3. *At least $\tau/2$ and at most τ flags are forwarded to the second collection tree.*

Proof. The process of sending σ ⟨Flag⟩ messages to the collection nodes in the first collection tree can be modeled as a balls and bins problem as in Lemma 16 with $\alpha = 14s\log n$ and $k = \tau/14\log n$. E_1 and E_2 are, respectively, the events that some collection node fails to receive $7s\log n$ flags and that some collection node receives more than $21s\log n$ flags. By applying the lemma, we get

$$\begin{aligned}\text{Prob}(E_1) &\le \frac{2\tau}{14\log n}\left(\frac{2}{\mathrm{e}}\right)^{7s\log n}\\ &\le \frac{2n}{14\log n}2^{-0.4426\times 7s\log n}\\ &\le \frac{1}{7n^{2s}\log n}\end{aligned}$$

and

$$\begin{aligned}\text{Prob}(E_2) &\le \frac{2\tau}{14\log n}\left(\frac{8\mathrm{e}}{27}\right)^{7s\log n}\\ &\le \frac{2n}{14\log n}2^{-0.3121\times 7s\log n}\\ &\le \frac{1}{7n^{1.1s}\log n}\end{aligned}$$

Thus, the probability that (a) fails is at most $\frac{1+n^{0.9s}}{7n^{2s}\log n}$.

To see (b), we note that there are $\tau/(14\log n)$ collection nodes in the first collection subtree, each of whom generates a ⟨Count⟩ message when it has received $7\log n$ flags. The flags correspond to distinct 1-inputs, and hence together they account for $\tau/2$ 1-inputs. Thus, (b) fails only if some node fails to receive at least $7\log n$ flags, which is already accounted for in the failure of (a).

To prove (c), we need to track the progress of the marked flags. Let E_1' and E_2' denote respectively, the events that some node fails to receive at least $7\log n$ marked flags and that some node receives more than $21\log n$ marked flags. Then, since there are τ marked flags, applying Lemma 16 with $\alpha = 14\log n$ and $k = \tau/14\log n$ we see that

$$\begin{aligned}\text{Prob}(E_1') &\le \frac{2\tau}{14\log n}\left(\frac{2}{\mathrm{e}}\right)^{7\log n}\\ &\le \frac{2n}{14\log n}2^{-0.4426\times 7\log n}\\ &\le \frac{1}{7n^{2}\log n}\end{aligned}$$

and

$$\begin{aligned}\text{Prob}(E_2') &\le \frac{2\tau}{14\log n}\left(\frac{8\mathrm{e}}{27}\right)^{7\log n}\\ &\le \frac{2n}{14\log n}2^{-0.3121\times 7\log n}\\ &\le \frac{1}{7n^{1.1}\log n}.\end{aligned}$$

Within each collection node, by transferring the marks from some marked flags to some unmarked flags, we may assume that the marked flags are the first to arrive. We can do this transfer because it does not change the distribution of marked and unmarked flags between the nodes, nor does it change the total number of marked flags across all collection nodes. The advantage of this change is that in following the algorithm, each node will first use all its marked flags before using unmarked flags.

In particular, as long as E_1' and E_2' do not occur, each node will use $7\log n$ flags to generate a $\langle\mathsf{Count}\rangle$ message, after which it will be left with between 0 and $14\log n$ marked flags. Since it forwards up to $14\log n$ flags to the next collection subtree, it follows that it will forward *all* of its marked flags and possibly some unmarked flags to the next subtree. Since there are τ marked flags across all the collection nodes, and the $\langle\mathsf{Count}\rangle$ messages account for $\tau/2$ of them, it follows that the remaining $\tau/2$ marked flags are forwarded. Hence, at least $\tau/2$ flags are forwarded. Moreover, since there are $\tau/(14\log n)$ nodes and each forwards at up to $14\log n$ flags, at most τ flags are forwarded, which establishes (c).

Now, let $E = E_1 \cup E_2 \cup E_1' \cup E_2'$ be the union of all the bad events we've encountered. For large enough n,

$$\begin{aligned}\text{Prob}(E) &\le \frac{1}{7n^{2s}\log n} + \frac{1}{7n^{1.1s}\log n} + \frac{1}{7n^{2}\log n} \\ &\quad + \frac{1}{7n^{1.1}\log n} \\ &\le \frac{1}{7n\log n}\end{aligned}$$

Thus, with probability at least $1 - \frac{1}{7n\log n}$, (a), (b), and (c) are all true, as desired. □ □

We will also need to prove a similar lemma when $\tau < n/2$. Note that when $\tau \ge n/2$, we have $\sigma \le 2\tau$, or $s = \sigma/\tau \le 2$. When $\tau < n/2$, σ may be much bigger than τ. Let $M = \min\{\sigma/\tau, 2\}$.

Lemma 18. *Suppose $\tau < n/2$. In the* Thresh-Count *algorithm, with probability at least $1 - \frac{1}{7n\log n}$, the first collection subtree satisfies all of the following:*

1. *Each collection node receives between $7\log n$ and $21M\log n$ flags.*

2. *Each filter node receives at most $21M\log n$ flags.*

3. *The $\langle\mathsf{Count}\rangle$ messages generated in this tree, when they reach the root, account for $\tau/2$ 1-inputs.*

4. *At least $\tau/2$ and at most τ flags are forwarded to the second collection tree.*

Proof. When $\tau < n/2$, the flags are not sent directly to the collection nodes, but rather to leaf nodes of the filters below the collection nodes. We will say that a filter receives a flag if the flag is received by any of its leaf nodes.

We first note that each party's process of selecting a random collection node, and then choosing a random leaf in its filter, is equivalent to choosing a uniformly random leaf node from among all the leaf nodes for all the filters. We've already remarked that adding the filters is equivalent to extending the first collection subtree to depth $\log n - 2$ while keeping the collection layer the same. Thus, there are $n/4$ filter leaf nodes to choose from. Using the Poisson approximation and an argument similar to the one in Lemma 16, it is easy to see that when $\sigma \le n$ parties each independently send a flag to a uniformly random filter leaf node out of $n/4$ choices, the probability of the event E_0, that there is a leaf node that receives more than $21\log n$ flags is less than $n^{-\log\log n}$.

Once the flags have been sent to the leaf nodes of the filters, they are forwarded up the filter from nodes to their parents, all the way to the collection node, with the only caveat that nodes do not forward more than $21\log n$ flags. Since each node has two children, it follows that each node in the filter receives at most $42\log n$ flags, and the same applies to the collection nodes. At the same time, viewing the process as first selecting a collection node, and then a filter leaf node below it, we see as in Lemma 17 that the probability of the event E_2, that there is a filter that receives more than $21s\log n$ flags is at most $\frac{1}{7n^{1.1s}\log n}$. Since no node in the filter can get more flags than the filter as a whole, it follows that the filter nodes and the collection nodes all receive no more than $21M\log n = \min\{21s\log n, 42\log n\}$ flags. This shows (b) and the upper bound in (a).

To show that the collection nodes each receives at least $7\log n$ flags with high probability and that together the collection nodes receive at least τ flags, we will once again track the marked flags. As we have remarked previously, although the marked flags are the first τ to be sent, by asynchrony, they need not be the first τ to arrive at the filters. Thus, it need not be the case that all these marked flags are forwarded through to the collection nodes. Nevertheless, we will argue that for every marked flag that fails to be forwarded, at least one unmarked flag was forwarded instead. To see this, note that as in Lemma 17, all the filters receive between $7\log n$ and $21\log n$ *marked* flags, except with probability $\frac{1+n^{0.9}}{7n^2\log n}$. Thus, each node in a filter can have at most $21\log n$ marked flags arrive at it.

Now, suppose a filter node fails to forward one or more marked flags. It can only do this if it has previously forwarded $21\log n$ flags, and since it can receive at most $21\log n$ marked flags, it follows that it has already forwarded at least as many *unmarked* flags as it is choosing to ignore marked ones. Once again, by transferring marks from the marked flags that are dropped to the unmarked flags that have been sent in their place, we can ensure that except with probability $\frac{1+n^{0.9}}{7n^2\log n}$, between $7\log n$ and $21\log n$ marked flags get through each filter to the corresponding collection node, and at least τ marked flags get through all the filters together, to the collection layer of the first collection subtree. This shows the lower bound in (a) and sets us up to show (c) and (d).

For (c), we will once again pretend, by transferring marks that at each node the marked flags are the first to arrive and be used. As before, we do this without altering the distribution of marked and unmarked flags between collection nodes. Note that each newly marked flag at the collection node corresponds to a distinct 1-input, so the $7\log n$ of them used by each of $\tau/(14\log n)$ collection nodes to generate a ⟨Count⟩ message accounts for $\tau/2$ 1-inputs at the root. This leaves between 0 and $14\log n$ marked flags at each collection node, which adds up to $\tau/2$ of them across all the collection nodes. Since each collection node forwards up to $14\log n$ flags, all the marked flags are forwarded, so that at least $\tau/2$ flags are forwarded to the next collection subtree. Since each of $\tau/(14\log n)$ collection nodes forwards up to $14\log n$ flags, at most τ flags are forwarded to the next collection tree, proving (d).

Finally, adding up the probabilities of all the bad events we've encountered, we see that for large enough n, $\frac{1+n^{0.9}}{7n^2\log n}+\frac{1}{7n^{1.1}\log n}+n^{-\log\log n}<\frac{1}{7n\log n}$. It follows that with probability at least $1-\frac{1}{7n\log n}$, (a), (b), (c), and (d) are all true, as desired. □ □

We are now ready to study what happens further up in the data structure. We will say that the algorithm succeeds up to level j if for all $i \le j$ the following are true:

1. All the collection nodes in the ith collection subtree receive between $7\log n$ and $42\log n$ flags.
2. The ⟨Count⟩ messages generated in the ith subtree account for $\tau/2^i$ 1-inputs at the root.
3. Between $\tau/2^i$ and $\tau/2^{i-1}$ flags are forwarded from the ith collection subtree to the $(i+1)$st collection subtree

Lemma 19. *Let $j \le D$. In the Thresh-Count algorithm, with probability at least $1-\frac{j}{7n\log n}$, the algorithm succeeds up to level j.*

Proof. We proceed by induction on j. We have already established the base case $j = 1$ in Lemmas 17 and 18. Now suppose $j \ge 2$, and for an induction hypothesis we assume that the algorithm succeeds to level $j-1$ with probability at least $1-\frac{j-1}{7n\log n}$. Let us condition on this event. This means that between $\tau/2^{j-1}$ and $\tau/2^{j-2}$ flags are forwarded to the jth collection subtree, which has $\frac{\tau}{2^{j-1}14\log n}$ collection nodes.

Thus, we can apply Lemma 16 with α between $14\log n$ and $28\log n$. The proof that conditioned on the algorithm having succeeded up to level $j-1$, it succeeds to level j, except with probability $\frac{1}{7n\log n}$, is identical to the proof of Lemma 17. By Bayes' law and the induction hypothesis, the unconditional probability that the algorithm succeeds to level j

$$\left(1-\frac{j-1}{7n\log n}\right)\left(1-\frac{1}{7n\log n}\right) \ge 1-\frac{j}{7n\log n},$$

as desired. □ □

Corollary 1. *With probability at least $1-\frac{1}{7n}$, the root node successfully accounts for at least τ 1-inputs.*

Proof. The last collection subtree is the one corresponding to $j = D$, and by Lemma 19, with probability at least $1-\frac{D}{7\log n}$ the root has accounted for $\sum_{j=1}^{D}\tau/2^j = \tau(1-2^{-D})$ 1-inputs, and moreover, between $\tau/2^D$ and $\tau/2^{D-1}$ flags have been forwarded directly to the root, by the collection nodes in the last collection subtree. Since no randomness is involved, the root eventually receives all of these flags. Thus, conditioned on the algorithm succeeding up to level D, the root eventually accounts for at least τ 1-inputs. Since $D < \log\tau < \log n$, the success probability is at least $1-\frac{1}{7n}$. □ □

We now prove the Theorem 3. Lemmas 17 to 19 and Corollary 1 show that with probability at least $1 - \frac{1}{7n}$, the root accounts for at least τ 1-inputs while ensuring the following:

1. Filter nodes receive no more than $42 \log n$ messages and send no more than $21 \log n$ messages.

2. Collection nodes receive no more than $42 \log n$ messages and send no more than $14 \log n + 1$ messages. (The extra 1 is for the $\langle \mathsf{Count} \rangle$ message.)

3. The root receives no more than $\tau / 2^{D-1} = 28 \log n$ $\langle \mathsf{Flag} \rangle$ messages.

Additionally, the adding nodes each receive two $\langle \mathsf{Count} \rangle$ messages and send one $\langle \mathsf{Count} \rangle$ message, and the root receives $D \leq \log n$ $\langle \mathsf{Count} \rangle$ messages, one from each of the collection subtrees. Individual parties send at most one message each when their input is set to **1**. We have already remarked that the messages used in this algorithm can be encoded using two bits. Thus, in the Up stage of the algorithm, each party sends and receives $O(\log n)$ messages of constant size. In the Down stage, $\langle \mathsf{Done} \rangle$ messages are sent via a canonical complete binary tree, so each party except the root receives exactly one $\langle \mathsf{Done} \rangle$ message, and each party that is not a leaf in the tree sends (at most) two $\langle \mathsf{Done} \rangle$ messages. Since all messages that are sent are eventually received, eventually all the parties receive the $\langle \mathsf{Done} \rangle$ message and terminate. Since the depths of the data structure used in the Up stage and the binary tree used in the Down stage are both $\log n$, the longest chain of messages is of length $2 \log n$, and hence the total latency is $O(\log n)$. Finally, since the computations done by each node during the algorithm amount to counting the number of messages it receives and generating up to $14 \log n$ random numbers, each node performs $O(\log n)$ computations. □

6.5 Using Quorums as Nodes in the Count Tree

So far in this section, we have assumed that all of the nodes in the count tree follow the protocol honestly. However, this is not the case in our MPC model, where some of the parties can play maliciously. To fix this, we assign a quorum to each node in the tree and let the quorums perform the roles of the parties. In our MPC protocol described in Section 4, we introduce Protocol 4 that allows us to run the threshold counting algorithm in a malicious setting.

Lemma 6 shows that a quorum Q can securely send a message M to another quorum Q'. However, there is some subtlety involved in using this fact. Every party in a quorum communicates with its parent when it has received at least half as many inputs as the parents' threshold. However, due to asynchrony, multiple messages may arrive simultaneously; when the threshold is set, not all parties in the quorum may be in the same state. Some may already have more inputs than the threshold, while others may still be waiting because messages from their children have been delayed. Lemma 6 tells us that if all parties in the quorum send the *same* message to the parent quorum, then the parent quorum can resolve that message. Thus, in order to ensure that all parties in the quorum send the same message to the parent quorum, we have required that even if a party's received inputs exceed his threshold, it should only inform the parent of having met the threshold, not of having exceeded it. The remaining inputs are held to be sent later.

7 Asynchronous Quorum Formation

In this section, we describe the quorum building algorithm of King et al. [KSSV06b, KLST11], and then adapt it to the asynchronous communication model to prove Theorem 4.

Dealing with asynchronous communication, particularly in conjunction with malicious faults, is notoriously tricky. To exercise particular care in this challenging domain, we include complete algorithms and proofs. Also, we believe our result on asynchronous quorum formation may be of independent interest, and so for the sake of completeness, we include the full result in this paper.

One may alternatively use the asynchronous Byzantine agreement protocol of Braud-Santoni et al. [BGH13] to build a set of n quorums. Similar to [KLST11], [BGH13] provides an almost everywhere to everywhere agreement protocol, which enables all parties to agree on a global string that is random enough to be used as an input to a sampler to generate a collection of n quorums. This protocol requires each party on average to send $\mathsf{polylog}(n)$ field elements, and perform $\mathsf{polylog}(n)$ computations. However, it is not load-balanced as some parties may send a linear number of field elements. Using this result, our MPC protocols will cost only polylogarithmic bits and computations.

We start the description of our protocol by defining the *semi-random-string agreement problem*, where the goal is to agree on a single string of length $O(\log n)$ with a constant fraction of random bits, where for any positive constant ϵ, a $1/2+\epsilon$ fraction of the parties are honest. King et al. [KLST11] present an asynchronous algorithm as an additional result that we call SRS-to-Quorum. The SRS-to-Quorum algorithm can go from a solution for *semi-random-string agreement* problem to a solution for the *quorum building* problem. Thus, their techniques can be extended to the asynchronous model assuming a scalable asynchronous solution for the semi-random-string agreement problem. We describe the Build-Quorums algorithm based on SRS-to-Quorum and an algorithm, which we call SRS-Agreement, that solves the semi-random-string agreement problem in the asynchronous model with pairwise channels.

Protocol 22 Build-Quorums

Goal. Generate n quorums.

1. All parties run SRS-Agreement.
2. All parties run SRS-to-Quorum.

King et al. [KSSV06b] present a synchronous algorithm, where a set of parties, up to $(1/3-\epsilon)n$ of which are controlled by an adversary, can reach almost-everywhere[1] agreement with probability $1-o(1)$. Their primary technique is to divide the parties into groups of polylogarithmic size; each party is assigned to multiple groups. In parallel, each group uses the bin election algorithm of [Fei99] to *elect* a small number of parties from within their group to move on. This step is recursively repeated on the set of elected parties until the number of remaining parties in this set becomes polylogarithmic. At this point, the remaining parties can solve the semi-random-string agreement problem. Provided the fraction of corrupted parties in the set of remaining parties is less than 1/3 with high probability, these parties succeed in agreeing on a semi-random string. Then, these parties send the result to the rest of the parties.

Bringing the parties to an agreement on a semi-random string is trickier in the asynchronous model. The major difficulty is that the bin election algorithm cannot be used in the asynchronous model since the adversary can prevent a fraction of the honest parties from being part of the election. We present a similar algorithm to [KSSV06b] that solves this issue in the asynchronous model with private channels. The main result of this section is as follows.

Theorem 16. *Suppose there are n parties, for any fix positive ϵ constant fraction $1/4-\epsilon$ of which are corrupted. There is a polylogarithmic (in n) bounded degree network and a protocol such that:*

1. *With high probability, a $1-O(1/\log n)$ fraction of the honest parties agree on the same value (bit or string).*
2. *Every honest party sends and processes only a polylogarithmic (in n) number of bits.*
3. *The number of rounds required is polylogarithmic in n.*

The important novelty of our method when compared to the result of King et al. [KSSV06b] is that instead of the bin election algorithm, we run the classic asynchronous MPC protocol to determine which parties can proceed to the next level. The simple version of our election method is presented as Simple-Elect-Subcommittee in Protocol 23 that has the properties described in Lemma 20. The complete protocol and its proof of correctness are given in Section 7.5

Lemma 20. *Let W be a committee of $\Theta(\log^8 n)$ parties, where the fraction, f_W, of honest parties is greater than 3/4. There exists any constant c such that with probability at least $1-n^{-3c/4}$, the Elect-Subcommittee protocol elects a subset W_B of W such that $|W_B| = c\log^3 n$ and the fraction of honest parties in W_B is greater than $(1-1/\log n)f_W$. The Elect-Subcommittee protocol uses a polylogarithmic number of bits and polylogarithmic number of rounds in a fully connected network.*

Proof. The proof follows from a straightforward application of the union and Chernoff bounds. Let X be the number of honest parties in W_B. By the correctness of the ACMPC algorithm, each party in W_B is randomly chosen from

[1] This is done by relaxing the requirement that all honest parties come to an agreement at the end of the protocol to instead require that a $1-o(1)$ fraction of honest parties reach agreement. This relaxation is called *almost-everywhere agreement* and was first introduced by Dwork et al. [DPPU86].

Protocol 23 Simple-Elect-Subcommittee

Goal. $\Theta(\log^8 n)$ parties agree on a subcommittee of size $\Theta(\log^3 n)$. The protocol is performed by parties $P_1, \ldots, P_k \in W$ with $k = \Theta(\log^8 n)$.

1. Party P_i generates a vector of $c \log^3 n$ random numbers chosen uniformly and independently at random from 1 to k, where each random number maps to one party. The value of c is calculated in Lemma 20.
2. Run ACMPC to compute the component-wise sum modulo k of all the vectors.
3. Let W_B be the set of winning parties, which are those associated with the components of the sum vector.
4. Return W_B as the elected subcommittee.

W. Let Y_i be an indicator random variable, that equals 1 if the i-th member of W_B is honest. Then, $E[Y_i] = f_W$ and $E[X] = f_W c \log^3 n$. Using the Chernoff bound, we have

$$\begin{aligned} &Pr[X < (1 - 1/\log n) f_W c \log^3 n] \\ &\quad = Pr[X < (1 - 1/\log n) E[X]] \\ &\quad \le e^{-\frac{E[X]/\log^2 n}{2}} \\ &\quad < n^{-3c/4}. \end{aligned}$$

□ □

We establish a polylogarithmic bound on the number of bits used in Elect-Subcommittee protocol since the bit cost of Elect-Subcommittee is polynomial in the number of parties participating in the algorithm.

7.1 The Election Graph

Our algorithms use of an election graph to determine which parties will participate in which elections. This graph was described in [KSSV06a, KSSV06b] and is repeated here.

Before describing the election graph, we first present a result similar to that used in [CL95]. Let X be a set of parties. For a collection $\mathcal{F}$ of subsets of X, a parameter δ, and a subset X' of X, let $\mathcal{F}(X', \delta)$ be the sub-collection of all $F' \in \mathcal{F}$ for which

$$\frac{|F' \bigcap X'|}{|F'|} > \frac{|X'|}{|X|} + \delta.$$

In other words, $\mathcal{F}(X', \delta)$ is the set of all subsets of $\mathcal{F}$ whose overlap with X' is larger than the "expected" size by more than a δ fraction. Let $\Gamma(r)$ denote the neighbors of node r in a graph.

Lemma 21. *Let l, r, n be positive integers such that l and r are all no more than n and $r/l \ge l n^{1-z} n$ for some real number z. Then, there is a bipartite graph $G(L, R)$ such that $|L| = l$ and $|R| = r$ and*

1. *Each node in R has degree $\log^z n$.*
2. *Each node in L has degree $O((r/l) \log^z n)$.*
3. *Let $\mathcal{F}$ be the collection of sets $\Gamma(r)$ for each $r \in R$. Then, for any subset L' of L, $|\mathcal{F}(L', 1/\log n)| < \max(l, r)/\log^{z-2} n$.*

The proof of Lemma 21 follows from a counting argument using the probabilistic method and is omitted. The following corollaries follow immediately by repeated application of the above lemma.

Corollary 2. *Let ℓ^* be the smallest integer such that $n/\log^{\ell^*} n \le \log^{10} n$. There is a family of bipartite graphs $G(L_i, R_i), i = 0, 1, \ldots, \ell^*$, and constants c_1 and c_2 such that $|L_i| = n/\log^i n$, $|R_i| = n/\log^{i+1} n$, and*

1. *Each node in R_i has degree $\log^{c_1} n$.*

2. *Each node in L_i has degree $O(\log^{c_2} n)$.*

3. *Let $\mathcal{F}$ be the collection of sets $\Gamma(r)$ for each $r \in R$. Then, for any subset L_i' of L_i, $|\mathcal{F}(L_i', 1/\log n)| < |R_i|/\log^6 n$.*

4. *Let $\mathcal{F}'$ be the collection of sets $\Gamma(l)$ for each $l \in L$. Then, for any subset R_i' of R_i, $|\mathcal{F}'(R_i', 1/\log n)| < |L_i|/\log^6 n$.*

Corollary 3. *Let ℓ^* be the smallest integer such that $n/\log^{\ell^*} n \le \log^{10} n$. There is a family of bipartite graphs $G(L_i, R_i), i = 0, 1, \ldots, \ell^*$, such that $|L_i| = n/\log^i n$, $|R_i| = n/\log^{i+1} n$, and*

1. *Each node in R_i has degree $\log^5 n$.*

2. *Each node in L_i has degree $O(\log^4 n)$.*

3. *Let $\mathcal{F}$ be the collection of sets $\Gamma(r)$ for each $r \in R$. Then, for any subset L_i' of L_i, $|\mathcal{F}(L_i', 1/\log n)| < |L_i|/\log^3 n$.*

Lemma 21 and its corollaries show there exists a family of bipartite graphs with strong expansion properties, which allow the formation of subsets of parties, where all but a small fraction contain a majority that is honest.

We are now ready to describe the election graph. Throughout, we refer to nodes of the election graph as e-nodes to distinguish them from nodes of the static network. Let ℓ^* be the minimum integer ℓ such that $n/\log^{\ell} n \le \log^{10} n$; note that $\ell^* = O(\log n/\log\log n)$. The topmost layer ℓ^* has a single e-node, which is adjacent to every e-node in layer $\ell^* - 1$. For the remaining layers $\ell = 0, 1, ..., \ell^* - 1$, there are $n/\log^{\ell+1} n$ e-nodes. There is an edge between the ith e-node, A, in layer ℓ and the jth e-node, B, in layer $\ell + 1$ if and only if there is an edge between the ith node in $L_{\ell+1}$ and the jth node in $R_{\ell+1}$ from Corollary 3. In such a case, we say that B is the parent of A, and A is the child of B. Note that e-nodes have many parents.

Each e-node will contain a set of parties known as a *committee*. All e-nodes, except for the one on the top layer and those in layer 0, will contain $c\log^3 n$ parties. Initially, we assign the n parties to e-nodes on layer 0 using the bipartite graph $G(L_0, R_0)$ described in Corollary 3. The i^{th} party is a member of the committee contained in the j^{th} e-node of layer 0 if and only if there is an edge in G between the i^{th} node of L_0 and the j^{th} node of R_0. Note every e-node on layer 0 has $\log^5 n$ parties in it.

The e-nodes on higher layers have committees assigned to them during the course of the protocol. Let A be an e-node on layer $\ell > 0$, let $B_1, \ldots, B_s$ be the children of A on layer $\ell - 1$, and suppose that we have already assigned committees to e-nodes on layers lower than ℓ. If $\ell < \ell^*$, we assign a committee to A by running Elect-Subcommittee on the parties assigned to $B_1, \ldots, B_s$, and assigning the winning subcommittee to A (note that we can run each of these elections in parallel). If A is at layer ℓ^*, the parties in A, $B_1, \ldots, B_s$, run byzantine agreement for Byzantine agreement.

7.2 Static Network with Polylog-Bounded Degree

We now repeat the description of the bounded degree static network [KSSV06b] and show how it can be used to hold elections specified by the election graph. For each e-node A, we form a collection of parties, which we call its s-node: $s(A)$. Intuitively, the s-node $s(A)$ serves as a central communication point for an election occurring at e-node A. Our goal is to bound the fraction of s-nodes controlled by the adversary by a decreasing function in n, namely $1/\log^{10} n$, for each layer. As the number of s-nodes grows much smaller with each layer, we need to make each s-node more robust. To do this, the number of parties contained in the s-node increases with the layer. Specifically, the s-nodes for layer i are sets of $\log^{i+12} n$ parties. We determine these s-nodes using the bipartite graph from Lemma 21, where L is a collection of n nodes, one for each party, R is the set of s-nodes for layer i and the degree of each node in R is set to $\log^{i+12} n$. The neighbors of each node in R constitute a set of parties in an s-node on layer i.

We use the term *link* to denote a direct connection in the static network. The communications for an election A will all be routed through $s(A)$: a message from a party x to $s(A)$ on layer i will pass from the party to a layer 0 s-node, whose parties will forward the message to a layer 1 s-node and so on, the goal being to reliably transmit the message via increasingly larger s-nodes up to $s(A)$. Similarly, communications with an individual party x from $s(A)$ will be transmitted down to a layer 0 s-node whose parties will send the message to x. We describe the connections in the static network.

Connections in the static network. Consider the following:

- Let A be an e-node on layer 0 in the election graph. Every party in A has a link to every party in $s(A)$.
- Let A and B be e-nodes in the election graph at levels i and $i-1$ respectively such that A is a parent of B. Thus, $s(A)$ has $\log^{i+12} n$ parties in it and $s(B)$ has $\log^{i+11} n$ parties in it. Let G be a bipartite graph as in Lemma 21 where L is the set of parties in $s(A)$, R is the set of parties in $s(B)$ and the degree of R is set to $\log^{c_1} n$ and the degree of L is set to $O(\log^{c_2} n)$. If there is an edge between two nodes in L and R respectively, then the corresponding party in $s(A)$ has a link to the corresponding party in $s(B)$. We will sometimes say that $s(A)$ is adjacent to $s(B)$ in the static network.

The following lemma follows quickly from the application of Lemma 21 and its corollaries. Item (1) follows from Lemma 3.1; items (2) and (4) from Corollary 3.2; and item (3) from Corollary 3.1. Although item (2) only makes a guarantee about layer 0 e-nodes, we will see eventually that with high probability, the fraction of corrupted e-nodes on every layer is small.

Lemma 22. *With high probability, the election graph and the static network have the following properties:*

1. *(Bad s-nodes) Any s-node whose fraction of corrupt parties exceeds $b + 1/\log n$ will be called bad. Else, we will call the s-node good. No more than a $1/\log^{10} n$ fraction of s-nodes on any given layer are bad.*
2. *(Bad e-nodes) Any e-node whose fraction of corrupt parties exceeds $b + 1/\log n$ will be called bad. Else we call the e-node good. No more than a $1/\log^2 n$ fraction of e-nodes on layer 0 are bad.*
3. *(Bad s-node to s-node connection) For any pair of e-nodes A and B joined in the election graph, the parties in s-nodes $s(A)$ and $s(B)$ are linked such that the following holds. For any subset W_A of parties in $s(A)$, at most a $1/\log^6 n$ fraction of parties in $s(B)$ have more than a $|W_A|/|s(A)| + 1/\log n$ fraction of their links to $s(A)$ with parties in W_A.*
4. *(Bad e-node to e-node connection) Let $|I|$ represent the total number of e-nodes on layer i in the election graph. For any set W of e-nodes on any layer i, at most a $1/\log^2 n$ fraction of e-nodes on layer $i+1$ have more than $|W|/|I| + 1/\log n$ fraction of their neighbors in W.*

The degree of the static network is polylogarithmic.

7.3 Communication Protocols

A *permissible path* is a path of the form $P = x, s(A_0), s(A_1), ..., .s(A_i)$ where x is a party in A_0, i is the current layer of elections being held, each A_j is an e-node on layer j, and there is an edge in the election graph between A_j and A_{j+1} for $j = 0, ..., i$. Each party y in an s-node $s(A)$ on each layer j keeps a *List* of permissible paths that determine which parties' messages it will forward. The List (for $y \in s(A)$) represents y's view of which parties are elected (to the subcommittee) at A that are still participating in elections on higher layers. If y's List indicates that x is such a party, then the List will also have the entire path for x, which stretches from x to the elections on layer i in which x is currently participating in. We have the following definitions.

- We say a *s-node knows a message* [resp., *knows a permissible path*, or resp., *knows a* List *of permissible paths*] if $1 - b - 2/\log n$ parties in the s-node are honest and receive the same message [resp., have the same path on their Lists, or resp., all have the same List.]
- A permissible path P is good if every s-node on the path knows P. Else the path is bad. We will show our construction of the static network ensures at most a $1/\log n$ fraction of the permissible paths are bad.

We now describe three primitive communication subroutines: Sendhop, Send, and MessagePass. The subroutine Sendhop describes how s-nodes (with direct links) communicate with each other, Send describes how a party communicates with an s-node, and MessagePass describes how two parties communicate with each other.

Sendhop(s, r, m, P)**.** A message m can be passed from s (the sender) to r (the receiver) from a level i to a level $i-1$ or from a level i to a level $i+1$, where s and r are s-nodes on these layers or one of s, r is a 0-layer s-node and the other is a party. If a party x sends a message to a layer 0 s-node $s(A)$ it sends the message to every party in $s(A)$ (note by construction it will have a direct link with every party in $s(A)$). Similarly if a message is sent from a layer 0 s-node $s(A)$ to a party x, every party in $s(A)$ sends the message to x.

When an s-node $s(A)$ sends a message to s-node $s(B)$, every party in $s(A)$ sends the message to those parties of $s(B)$ to which it has a direct link. When each party in $s(B)$ receives such a set of messages, it waits until it receives the same messages from the majority to determine the message. If there is no majority value, the party ignores the message. Along with sending the message the parties also send information that specifies along which path P the message is being sent. Each time a message is received by a party of an s-node $s(B)$ on layer $j \le i$, it checks that:

1. The message came from the s-node previous to it in the path P; if not the message is dropped.
2. The path P (or its reverse) is on its List of permissible paths. If not, the message is dropped.
3. Only messages that conform to the protocol in size and number are forwarded up and down the permissible paths. If more or longer messages are received from a party, messages from that party are dropped.

Send(s, r, m, P)**.** Of the first two parameters, one must be a party ("x") and one must be an s-node ("$s(A)$"). The path P contains the first parameter s as its start and the second parameter r as its endpoint. Send(s, r, m, P) sends the message m from s to r along the path P via repeated application of Sendhop.

MessagePass$(x \in A, y \in B, m, P_x, P_y)$**.** Both A and B are adjacent e-nodes. Hence, $s(A)$ and $s(B)$ are adjacent in the static network. A message from party x in e-node A sends message m to party y in e-node B by first calling Send$(x, s(A), m, P_x)$. Then, $s(A)$ sends m to $s(B)$ by calling Sendhop$(s(A), s(B), m, P)$, where P is the path consisting of two s-nodes $s(A), s(B)$. Finally, the message is transmitted from $s(B)$ to y by calling Send$(s(B), y, m, P_y^r)$, where P_y^r is the reversal of path P_y.

7.4 SRS-Agreement Protocol

Before describing the SRS-Agreement protocol, we first adapt the Elect-Subcommittee protocol for the static network. Let A be an e-node with children $B_1, \dots, B_s$, and let X be the set of all parties from $B_1, \dots, B_s$. For each $i \in [s]$ and $x \in B_i$, let P_x denote a good path of s-nodes from x to $s(B_i)$ concatenated with $s(A)$. At the start of the election for A, we assume that each node in P_x knows P_x and $s(A)$ knows $\{P_x \mid x \in X\}$.

We now describe the implementation of the Elect-Subcommittee algorithm. Every party $x \in X$ generate a vector of random numbers chosen uniformly and independently at random where each random number maps to one party. The parties use the ACMPC protocol to determine the winners (recall that the number of parties in e-nodes is always polylogarithmic, so this can be done sending only polylogarithmic messages). The list of winners is sent up to $s(A)$, where each party in $s(A)$ takes a majority to determine the winners. Then, $s(A)$ sends down the list of winners along all the permissible paths to each party $x \in X$. Parties on the path (i.e., in s-nodes along the path) update their Lists of permissible paths to remove those party-paths who lost as well as those party-pairs who won too many elections (we will quantify this shortly), and make $\log^4 n$ copies of each of the winners' paths and concatenate a different layer $i+1$ s-node parent onto each one. There is a condition in Step 5 that requires parties who have won more than 8 elections to be eliminated. This is a technical condition that ensures the protocol is load-balanced, and parties in an s-node do not communicate more than a polylogarithmic number of bits. We present a detailed description of Elect-Subcommittee in Protocol 24.

We describe the SRS-Agreement in protocol 25. Since every party is a member of $s(A^*)$, steps 5 and 6 will ensure the final result of the protocol is communicated to every party.

Protocol 24 Elect-Subcommittee

Goal. Adapted version of Simple-Elect-Subcommittee for static networks.

```
1. Let $A$ be an e-node with children $B_1, \ldots, B_s$, and let $X$ be the set of all parties from $B_1, \ldots, B_s$. For each $x \in X$: // This stage done in parallel
2.     Party $x$ randomly selects one of $k/(c_1 \log^3 n)$ random numbers chosen uniformly and independently at random from zero to $k$ where each random number maps to one party.
3. Parties in $X$ run ACMPC to compute the component-wise sum modulo $k$ of all the vector. Arbitrarily, add enough additional numbers to the vector to ensure it has $c \log^3 n$ unique numbers.
4. Let $M$ be the set of winning parties, which are those associated with some component of the vector sum.
5.     Each $y \in X$ sends $M$ to $s(A)$ by calling Send$(y, s(A), M, P_y)$.
6.     Parties in $s(A)$ determine $M$ by waiting until they receive the majority of same messages. These become the elected parties.
7. For each party $x \in X$ that is elected, the parties in $s(A)$ use Send$(s(A), x, m, P_x^r)$ to tell $x$, along with each s-node in $P_x$, that $x$ was elected.
8. Each party in each s-node revises its list of permissible paths to:
       Retain only the winners. Eliminate parties who have won more than 8 elections. Make $\log^4 n$ copies of remaining permissive paths, concatenating each with a different s-node neighbor on layer $i + 1$.
9. $s(A)$ sends its list to every adjacent s-node $s(B)$ on layer $i + 1$ using Sendhop$(s(A), s(B), m, P)$, where $P$ is the path consisting only of $s(A)$, $s(B)$.
```

Protocol 25 Scalable-SRS-Agreement

Goal. Parties agree on a semi-random string.

```
1. For $l = 1$ to $l^*$:
2.     For each e-node $A$ in layer $l$, let $B_1, \ldots, B_s$ be the children of $A$ in layer $l - 1$ of the election graph, and
3.         If $l < l^*$, run Elect-Subcommittee on the parties in nodes $B_1, \ldots, B_s$. Assign winning parties to node $A$.
4.         Else parties in nodes $B_1, \ldots, B_s$ solve semi-random-string agreement problem by communicating via the protocol Send through the $s$-node $s(A)$.
5. Let $A^*$ be the e-node on layer $l^*$, every party $x$ assigned to $A^*$ communicates the result of Step 4 to $s(A^*)$ using Send$(x, s(A^*), m, P_x)$.
6. Every party in $s(A^*)$ waits for the majority of same message to determine the result of Step 4.
```

7.5 Proof of Build-Quorums

To establish the correctness of the protocol presented in Section 7.4, we first state some claims regarding the primitive communication protocols. Their proofs follow by straightforward probabilistic arguments and are omitted in the interest of space. Recall the fraction of corrupted parties is b, where $b < 1/4 - \epsilon$ for any fix positive ϵ.

Claim 1. *Let $s(A)$ and $s(B)$ be s-nodes on consecutive layers. Assume the following conditions hold:*

1. *Both $s(A)$ and $s(B)$ are good.*
2. *$s(A)$ is on a permissible path known by $s(B)$.*
3. *There exists a set W of parties from $s(A)$ such that for every message m, all parties in W are honest and agree on a message m. Further W consists of at least a $1 - b - 2/\log n$ fraction of the parties in $s(A)$.*

Then, there is a set W' of parties from $s(B)$ such that for every message m, every party in W' is honest and agrees on the message m after Sendhop$(s(A), s(B), m, P)$ is called. (Here, P is the path $s(A), s(B)$.) Further, W' consists of all but a $1/\log^6 n$ fraction of the honest parties in $s(B)$.

Proof. Every party in W is honest and sends the same message to its connected parties in $s(B)$. The parties in $s(B)$ can afford to wait for the majority of same messages, since $s(A)$ is good and W consists of at least $1 - b - 2/\log n$ fraction of parties, which is more than $1/2$ and for majority we need to receive a fraction of $1/2$ same messages from the parties in $s(A)$. Thus, all honest party but a $1/\log^6 n$ fraction of parties in $s(B)$ will eventually receive the message based on corollary 2. □ □

Claim 2. *Let x be an honest party. Assume P_x is a good path. Then, after Send$(x, s(A), m, P_x)$ is executed, there is a fixed set W of honest parties, which contains all but a $1/\log^6 n$ fraction of the honest parties in $s(A)$ and every party $z \in W$ agrees on m.*

An election at e-node A is *legitimate* if the following two conditions hold simultaneously for more than a $3/4$ fraction of parties x participating in the election at A: (1) party x is honest; (2) The path P_x is good.

Lemma 23. *For a legitimate election at node A, let X be a set of honest parties with good permissible paths. (Note $|X| > 3\log^8 n/4$.) Let W be the set of honest parties in $s(A)$ that know X. Then, after the execution of* Elect-subcommittee*, the parties in W know the winners of the election in A, as do the s-nodes that belong to good paths P_x.*

Proof. From Claim 2, we have that every message m sent by MessagePass$(y \in B_i, z \in B_j, m, P_y, P_z)$ from $y \in X$ to $z \in X$ is received by some fixed set W of honest parties in $s(B_i)$, such that W contains at least $1 - b - 2/\log n$ fraction of the parties in $s(B_i)$. By Claim 1, every message sent by y is received by z. Since X contains more than $3/4$ of the total parties participating in the election, (after running ACMPC) all the parties in X will all agree on the same set of for random parties. Thus, after the parties in X send these values to $s(A)$, $s(A)$ will know the winners. When $s(A)$ sends these winners to X, by repeated application of Claim 1, we have every $x \in X$ and every s-node in P_x will know these winners. □ □

We have shown that in a legitimate election at node A, $s(A)$ knows the list of winners. We next consider when paths are dropped from the permissible path Lists.

7.5.1 Permissible Paths Removal

Let y be a party in some s-node on layer i. A permissible path P_x is removed from a party y's List on layer i if y receives a message from an s-node above it in P_x, indicating either x has won more than 8 elections or x lost in the election held at the last node of P_x. Here, we consider when P_x is removed for the former reason, i.e., we give an upper bound on the fraction of parties that are reported to have won too many elections on layer i.

First, we consider the effect of legitimate elections. The following lemma, a version of which appears in [KSSV06a, KSSV06b], shows that a slight fraction of honest parties wins more than 8 times in legitimate elections on a given layer.

Lemma 24. *With high probability, the parties that win more than 8 elections, counting multiplicities, account for no more than a $16/\log^3 n$ fraction of the honest parties that are winners of legitimate elections.*

Next, we bound the effect of elections that are not legitimate. We first consider the case where $s(A)$ is good, yet the fraction of honest parties participating in A with good paths is less than 3/4. For the remainder of the proof, we shall treat such an e-node A as a bad e-node.

Claim 3. *Suppose less than a $1/7$ fraction of the honest parties of a good $s(A)$ agree on a message m. Then, after Sendhop$(p(A), p(B), m, P)$ is executed, all but a $1/\log^6 n$ fraction of the honest parties in $s(B)$ will ignore m.*

Proof. Even if the corrupted parties agree on m, since $b < 1/4$, the total fraction of parties in $s(A)$ sending the message m is less than 11/28. Thus, at most a $1/\log^6 n$ fraction of the parties in $s(B)$ will receive m from a majority of parties in $s(A)$. □ □

Hence, a good $s(A)$ can only communicate with seven different sets of winners to the s-nodes below it. Since each honest party will send $\log^3 n$ winners, the total number of winners sent is at most $7\log^3 n$. Therefore, a bad e-node can cause at most $7\log^3 n$ parties to have their permissible paths removed.

Next, we consider the effect of a bad s-node. We will assume one bad s-node $s(A)$ on layer i can cause the removal of all the permissible paths for every party participating in the election at A. Since $\log^8 n$ parties participate in an election, and fewer than a $1/\log^{10} n$ fraction of the s-nodes are bad on a layer, the fraction of honest winners affected is less than $1/\log^2 n$. Thus, we can bound the fraction of the honest winners on any layer i that have their permissible paths removed by $1/\log^2 n + 1/\log^3 n + 7\beta_i$; where β_i represents the fraction of bad e-nodes on layer i. Thus, we have the following lemma.

Lemma 25. *Assume the fraction of bad e-nodes on layer i is bounded by $c/\log^2 n$, for some constant c. Then, the fraction of honest winners that have their permissible paths removed on layer i is bounded by $8c/\log^2 n$.*

7.5.2 Proof of Theorems 16

We now complete the proof of Theorem 16, which follows from the following lemma.

Lemma 26. *On layer i, with high probability, at least a $1 - 4/\log^2 n$ fraction of s-nodes $s(A_j)$ have the following properties:*

1. *$s(A_j)$ is good.*
2. *At least a $1 - b - 4i/\log n$ fraction of the parties in node A_j are honest and have good paths to $s(A_j)$ (note this implies $s(A_j)$ knows this path). That is, A_j is a good e-node.*

Proof. We prove the lemma by induction. On all layers and particularly layer 0, only a $1/\log^{10} n$ fraction of the s-nodes are bad. If $s(A)$ is good, then every party in A has a good path to $s(A)$. Further by construction all but a $1/\log^2 n$ fraction of the e-nodes on layer 0 consist of at least a $1 - b - 1/\log n$ fraction of honest parties. So the lemma is true on layer 0.

Assume the lemma is true for layer i. Then, a $1 - 4/\log^2 n$ fraction of e-nodes are good, more specifically these e-nodes have at least a $1 - b - 4i/\log n$ fraction of honest parties that have a good path to their corresponding s-node. Since the election is legitimate by Lemmas 20 and 23, with high probability, after Elect-Subcommittee at least a $1 - b - 4i/\log n - 1/\log n$ fraction of the parties elected are honest and have a good path to any good parent of their s-node. Thus, at least a $1 - b - (4i + 1)/\log n$ fraction of the parties elected at layer i are honest and have good paths to good parent s-nodes on layer $i + 1$. By Lemma 25 this fraction is reduced by at most $32/\log^2 n$. Thus, at least a $1 - b - (4i + 2)/\log n$ fraction of the parties elected at layer i are honest and have good paths to good parent s-nodes on layer $i + 1$. Since the fraction of bad s-nodes on layer $i + 1$ is at most $1/\log^{10} n$, by Corollary 3 at least a $1 - 1/\log^2 n - 1/\log^{10} n$ fraction of the e-nodes (and their corresponding s-nodes) are good on layer $i + 1$, and have at least a $1 - b - (4i + 2)/\log n - 1/\log n$ fraction of honest parties that have good paths to their corresponding s-nodes. □ □

By Lemma 26, with high probability the layer ℓ^* e-node is good. Thus, the parties in this e-node succeed in solving the semi-random-string agreement problem (Step 4 of algorithm SRS-Agreement). Since all the parties are in the s-node (though they may appear multiple times) corresponding to A on ℓ^*, by Claim 2 all but a $O(1/\log n)$ fraction of the honest parties learn the final result. To prove the number of bits sent by each party is polylogarithmic we note each party is in a polylogarithmic number of e-nodes and s-nodes on each layer i and participates in at most a polylogarithmic number of elections on layer i. Since the number of layers is $O(\log n)$ Theorem 16 follows. Finally, the correctness of Theorem 4 follows from Theorem 16 and the correctness of SRS-to-Quorum protocol.

8 Conclusion

We described a Monte Carlo algorithm to perform asynchronous MPC in a scalable manner. Our protocols are scalable in the sense that they require each party to send $\tilde{O}(m/n+\sqrt{n})$ messages and perform $\tilde{O}(m/n+\sqrt{n})$ computations. They tolerate a static adversary that controls up to a $1/8-\epsilon$ fraction of the parties, for ϵ any positive constant. We showed that our protocol is secure in the universal composability framework. We also described efficient algorithms for two important building blocks of our protocol: threshold counting and quorum building. These algorithms can be used separately in other distributed protocols.

The following problems remain open. Can we prove lower bounds for the communication and computation costs for Monte Carlo MPC? Can we implement and adapt our algorithm to make it practical for an MPC problem such as the beet auction problem described in [BCD+09]? Finally, can we prove upper and lower bounds for resource costs to solve MPC in the case where the adversary is *adaptive*, able to take over parties at any point during the algorithm?

9 Acknowledgments

The authors would like to acknowledge supports from NSF under grants CCF-1320994, CCR-0313160, and CAREER Award 644058. We are also grateful for valuable comments from Ran Canetti (Boston University), Shafi Goldwasser (MIT), Aniket Kate (Saarland), Yehuda Lindell (Bar-Ilan), and Seth Pettie (UMich).

References

[Abr74] Milton Abramowitz. *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,*. Dover Publications, Incorporated, 1974.

[AHS91] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks and multi-processor coordination. In *Proceedings of STOC'91*, pages 348–358. ACM, 1991.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $0(n \log n)$ sorting network. In *Proceedings of STOC'83*, pages 1–9, New York, NY, USA, 1983. ACM.

[AL11] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. Cryptology ePrint Archive, Report 2011/136, 2011.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*, page 14. John Wiley Interscience, March 2004.

[BCD+09] P. Bogetoft, D. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. Nielsen, J. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. *Financial Cryptography and Data Security*, pages 325–343, 2009.

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 52–61, New York, NY, USA, 1993. ACM.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. *Advances in Cryptology – CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, chapter Large-Scale Secure Computation: Multi-party Computation for (Parallel) RAM Programs, pages 742–762. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[BE03] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[Bea91] 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. Springer Berlin Heidelberg, 1991.

[BGH13] Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast Byzantine agreement. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 57–64, New York, NY, USA, 2013. ACM.

[BGT13] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation: how to run sublinear algorithms in a distributed setting. In *Proceedings of the 10th theory of cryptography conference on Theory of Cryptography*, TCC'13, pages 356–376, Berlin, Heidelberg, 2013. Springer-Verlag.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

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

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[BTH07] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous MPC. In *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'07, pages 376–392, Berlin, Heidelberg, 2007. Springer-Verlag.

[BW86] E Berlekamp and L Welch. Error correction for algebraic block codes, US Patent 4,633,470, December 1986.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science*, FOCS '01, pages 136–145, Oct 2001.

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

[CCG$^+$14] Nishanth Chandran, Wutichai Chongchitmate, Juan A. Garay, Shafi Goldwasser, Rafail Ostrovsky, and Vassilis Zikas. Optimally resilient and adaptively secure multi-party computation with low communication locality. Cryptology ePrint Archive, Report 2014/615, 2014.

[CD89] B. Chor and C. Dwork. Randomization in Byzantine agreement. *Advances in Computing Research*, 5:443–498, 1989.

[CDG88] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 87–119, London, UK, UK, 1988. Springer-Verlag.

[CFGN96] R. Canetti, U. Friege, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. Technical report, Cambridge, MA, USA, 1996.

[CHP13] Ashish Choudhury, Martin Hirt, and Arpita Patra. Asynchronous multiparty computation with linear communication complexity. In Yehuda Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 388–402. Springer Berlin Heidelberg, 2013.

[CL95] Jason Cooper and Nathan Linial. Fast perfect-information leader-election protocol with linear immunity. *Combinatorica*, 15:319–332, 1995.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[DI06] I. Damgård and Y. Ishai. Scalable secure multiparty computation. *Advances in Cryptology - CRYPTO 2006*, pages 501–520, 2006.

[DIK$^+$08] I. Damgård, Y. Ishai, M. Krøigaard, J. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. *Advances in Cryptology – CRYPTO '08*, pages 241–261, 2008.

[DKMS12] Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Brief announcement: breaking the $o(nm)$ bit barrier, secure multiparty computation with a static adversary. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 227–228, New York, NY, USA, 2012. ACM.

[DKMS14] Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Quorums quicken queries: Efficient asynchronous secure multiparty computation. In *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 242–256. Springer Berlin Heidelberg, 2014.

[DN07] I. Damgård and J.B. Nielsen. Scalable and unconditionally secure multiparty computation. In *Proceedings of the 27th annual international cryptology conference on Advances in cryptology*, pages 572–590. Springer-Verlag, 2007.

[DPPU86] C Dwork, D Peleg, N Pippenger, and E Upfal. Fault tolerance in networks of bounded degree. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 370–379, New York, NY, USA, 1986. ACM.

[Far88] R.W. Farebrother. *Linear Least Squares Computations*. Statistics: A Series of Textbooks and Monographs. Taylor & Francis, 1988.

[Fei99] Uriel Feige. Noncryptographic selection protocols. In *FOCS*, pages 142–153, 1999.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[GHY88] Zvi Galil, Stuart Haber, and Moti Yung. Cryptographic computation: Secure faut-tolerant protocols and the public-key model. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 135–155, London, UK, UK, 1988. Springer-Verlag.

[GKP$^+$13] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run Turing machines on encrypted data. In *Advances in Cryptology – CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 536–553. Springer Berlin Heidelberg, 2013.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[Gol00] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 101–111, New York, NY, USA, 1998. ACM.

[HKI$^+$12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *Information Security and Cryptology – ICISC 2012*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer Berlin Heidelberg, 2012.

[KLR10] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. *SIAM Journal on Computing*, 39(5):2090–2112, March 2010.

[KLST11] Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load balanced scalable Byzantine agreement through quorum building with full information. In *Distributed Computing and Networking*, volume 6522 of *Lecture Notes in Computer Science*, pages 203–214. Springer Berlin Heidelberg, 2011.

[Klu95] Michael Richard Klugerman. *Small-depth Counting Networks and Related Topics*. PhD thesis, Cambridge, MA, USA, 1995. Not available from Univ. Microfilms Int.

[KP92] Michael Klugerman and C. Greg Plaxton. Small-depth counting networks. In *Proceedings of STOC'92*, pages 417–428, 1992.

[KSSV06a] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 990–999, Philadelphia, PA, USA, 2006.

[KSSV06b] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 87–98, Washington, DC, USA, 2006. IEEE Computer Society.

[MSZ15] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Scalable multi-party shuffling. In *International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomized algorithms and probabilistic analysis*. Cambridge University Press, New York, 2005.

[PSR02] B. Prabhu, K. Srinathan, and C. Pandu Rangan. Asynchronous unconditionally secure computation: An efficiency improvement. In *INDOCRYPT 2002, Lecture Notes in Computer Science*, volume 2551, pages 93–107. Springer-Verlag, 2002.

[RS60] Irving Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, pages 300–304, 1960.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SR00] K. Srinathan and C. Pandu Rangan. Efficient asynchronous secure multiparty distributed computation. In *INDOCRYPT 2000, Lecture Notes in Computer Science*, volume 1977, pages 117–129. Springer-Verlag, 2000.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.