Quorums Quicken Queries: Efficient Asynchronous Secure Multiparty Computation

Varsha Dani [†] Valerie King [*] Mahnush Movahedi [†] Jared Saia [†]

Abstract

We describe an asynchronous algorithm to solve secure multiparty computation (MPC) over n players, when strictly less than a 1/8 fraction of the players are controlled by a static adversary. For any function f that can be computed by a circuit with m gates, our algorithm requires each player to send a number of bits and perform an amount of computation that is $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$. This significantly improves over traditional algorithms, which require each player to both send a number of messages and perform computation that is $\Omega(nm)$.

Contact: Varsha Dani, <varsha@cs.unm.edu> Tel: (505) 573-5390

This paper is a regular submission.

Eligible for Best Student Paper Award (Mahnush Movahedi is a full-time student.)

[*]University of Victoria

[†]Department of Computer Science, University of New Mexico, Albuquerque, NM 87131-1386; email: {varsha, mahnush, saia}@cs.unm.edu. This research was partially supported by NSF CAREER Award 0644058 and NSF CCR-0313160.

1 Introduction

Secure multiparty computation (MPC) abstracts numerous important problems in distributed security, and so, not surprisingly, there have been thousands of papers over the last several decades addressing this problem. In the MPC problem, n players each have a private input, and their goal is to compute the value of an n-ary function, f, over the inputs. They must do this without revealing any information about the inputs, other than what is revealed by the output of the function and their own input. The problem is complicated by the fact that a hidden subset of the players are controlled by an adversary which actively tries to subvert this goal.

Unfortunately, there is a striking barrier that prevents wide-spread use of MPC algorithms: resource inefficiency. In particular, if there are n players involved in the computation and the function f can be computed by a circuit with m gates, then most algorithms require each player to send a number of messages and perform a number of computations that is $\Omega(mn)$ [14, 15, 6, 1, 16, 12, 17, 18, 3].

Recent years have seen exciting improvements in the cost of secure MPC when m is much larger than n. [9, 11, 10]. For example, the computation and communication cost for the algorithm of [10] is $\tilde{O}(m)$ plus a polynomial in n. However, the additive polynomial in n is large (e.g. $\Omega(n^6)$) and so these new algorithms are only efficient for relatively small n. Thus, there is still a need for secure MPC algorithms that are efficient in both n and m.

1.1 Model and Formal Problem Statement

Model: We assume a private and authenticated communication channel exists between every pair of players. Unlike much past work on secure MPC, we do *not* assume the existence of a public broadcast channel.

We further assume a completely asynchronous communication model. In particular, the adversary can arbitrarily delay messages sent over all communication channels. Latency in this model is defined to be the maximum length of any chain of messages (see [7, 2]). In particular, latency ignores all computation by individual processors.

The function to be computed is presented as a circuit C with m gates, numbered $1, 2, \ldots, m$, where the gate numbered 1 is the output gate. Also, it is convenient for presentation to assume that all gates have in-degree 2 and out-degree at most 2, however we can tolerate arbitrary constant degrees. We also assume that all computations in the circuit occur over a finite field $\mathbb{F}$.

An unknown subset of the players are controlled by an adversary that is actively trying to prevent the computation of the function. We say that the players controlled by the adversary are *bad* (*i.e.* Byzantine) and that the remaining players are *good*. We assume a *static* adversary, meaning that the adversary must select the set of bad players at the start of the algorithm. In our model, the adversary is computationally unbounded, and so we make no cryptographic hardness assumptions.

Problem Statement: In secure MPC, there are n players, and each player i has a private input, x_i. There is an n-ary function f that is known to all players. The goal is to ensure that: 1) all players learn the value $f(x_1, x_2, \ldots, x_n)$; and 2) the inputs remain as private as possible: each player i learns nothing about the private inputs other than what is revealed by $f(x_1, x_2, \ldots, x_n)$ and x_i. In general, we seek to obtain a result that is equivalent to the scenario where all inputs are sent to a trusted external party, who computes the output, and then sends it back to each player.

In the asynchronous setting, the problem is challenging even with a trusted external party. In particular, the trusted party can not distinguish between the case when it has not received an input

because either 1) the player sending the input message is bad; or 2) the message sent by the player has been delayed indefinitely by the adversary.

Here is how a trusted party can help. Let t be the number of bad players. Since up to t of the inputs may never be sent, the trusted party must wait until at least $n - t$ inputs are received and then compute the function f over those inputs, with default values for the inputs not received. Finally, the output and the set of players from which inputs were received can be sent back to the players.

The goal of an asynchronous secure multiparty computation protocol is to simulate the above scenario, even without the trusted party. Our additional goal is to do this using a scalable amount of communication.

Let S denote the set of players from whom input is received by the (simulated) trusted party. $|S| \geq n - t$, as noted above.[1] For arbitrary such S a description of S requires $\Omega(n)$ bits, and cannot be sent back to the players using only a scalable amount of communication. Thus, we relax the standard requirement that S be sent back to the players. Instead we require that at the end of the protocol each good player learns the output of f; whether or not their own input was included in S; and the *size* of S.

1.2 Our Results

We describe an algorithm (Algorithm 1) to perform efficient, asynchronous secure MPC. The main result of this paper is as follows.

Theorem 1.1. Assume there are n players, strictly less than a 1/8 fraction of which are bad, and an n-ary function f that can be computed by a circuit with m gates. If the good players follow Algorithm 1, then w.h.p., the algorithm terminates and they can solve secure MPC, while ensuring:

1. Each player sends at most $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$ messages;
2. Each player performs $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$ computations; and
3. Total latency is $O(d + \text{polylog}(n))$ where d is the depth of the circuit.

Paper Organization: The rest of this paper is organized as follows. Section 2 gives a technical overview. In Section 3, we describe our algorithms. We conclude and give problems for future work in Section 4. All proofs are in Sections A and B.

2 Technical Overview

2.1 Building Blocks

We now describe the fundamental protocols that we use as building blocks in this paper: AVSS-SHARE, AVSS-RECONSTRUCT, and HW-MPC.

Our algorithm repeatedly uses an asynchronous verifiable secret sharing scheme from [4]. This scheme consists of two protocols. First, a sharing protocol, which we denote AVSS-SHARE. During this protocol, a dealer shares the secret among the players and each player verifies for itself that there is a unique secret defined by the shares. Second, a reconstruction protocol, which we denote AVSS-RECONSTRUCT. During this protocol, the players reconstruct the secret from their shares.

[1] 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.

Both of these protocols work correctly even if up to 1/4 of the players are bad. In particular, no information about the secret can be obtained if the bad players share information; and AVSS-RECONSTRUCT reconstructs the correct secret despite any efforts of the bad players to prevent this. The latency of both protocols is $O(1)$ and the communication cost of both protocols is polynomial in x, where x is the number of players participating in the protocol. In this paper, we will use the protocols only among small sets of players ("quorums") of logarithmic size, so x will be $O(\log n)$ and the communication cost per invocation will be polylogarithmic in n.

We also use a (heavy-weight) asynchronous algorithm for MPC that we call HW-MPC. This algorithm, due to Ben-Or et al. [4], is an errorless protocol for MPC that tolerates up to 1/4 bad players. The latency is at most linear in m and the number of bits sent is a polynomial in n and m.

2.2 Our Contributions

Our result in Theorem 1.1 requires three new techniques: Asynchronous Quorum Formation, Quorum-Based Gate Evaluation, and a Commitment Tree. Asynchromous Quorum formation and Commitment Tree techniques address the challenge of asynchronicity.

Asynchronous Quorum Formation: A quorum is a set of $C \log n$ players for some constant C. A quorum is called *good* if at least a 7/8 fraction of the players in it are good. In the *quorum formation problem*, we want a set of n players, t of which may be bad, to agree on a set of n good quorums. Moreover, we want the quorums to be load-balanced: each player is mapped to $O(\log n)$ quorums.

Recently, King et al. [19] described an efficient algorithm to solve the quorum formation problem, w.h.p., in the synchronous model with full-information. Our first result builds on the result of [19] to solve the quorum formation problem in the asynchronous model, with private channels. Our new algorithm, CREATE-QUORUMS, is described by the following theorem, whose proof is given in Section B.

Theorem 2.1. CREATE-QUORUMS solves the quorum formation problem, w.h.p., in the asynchronous communication model with private channels, when $t < (1/4 - \epsilon)n$ for any[2] fixed, positive ϵ. Moreover, it ensures:

- Each player sends at most $\tilde{O}(\sqrt{n})$ bits;
- Each player performs $\tilde{O}(\sqrt{n})$ computations.
- Total latency is $O(\text{polylog}(n))$.

Quorum-Based Gate Evaluation: Below, we assume all random variables are chosen uniformly at random from $\mathbb{F}$. Each player initializes the circuit by secret-sharing a random number R to the quorum associated with its input X, using AVSS-SHARE, and sending $R_X + X$ to the members of that quorum, as well. This quorum then provides an input to one or more gates in the cirucuit.

The technique for evaluating a gate of the circuit, is illustrated in Figure 1. The three ovals in the figures represent three good quorums, one for each input to the gate and one for the gate. At the start of the gate evaluation, we assume X and Y are the inputs to the gate. Every player in the input quorum for X, (Y, resp.) has the value $X + R_X$ ($Y + R_Y$, resp.), and a share of the secret R_X (resp., R_Y). Step 1 describes these initial conditions of the input quorums.

[2] In general we prove the correctness of Theorem 2.1 for $t < (1/4 - \epsilon)n$ and by assuming $\epsilon > 1/8$, we ensure the fraction of good players is strictly greater than 7/8 which is necessary for our definition of good quorums.

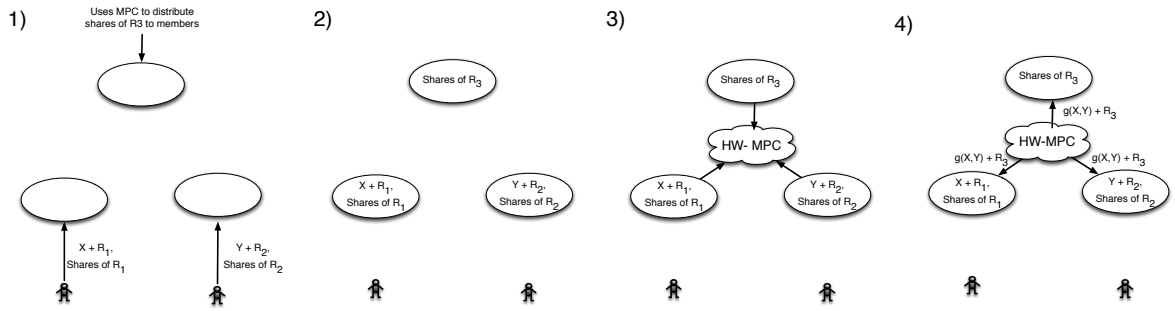

Figure 1: Example Computation of a Gate

In Step 2, the gate quorum runs an algorithm to distributed shares of a third random value R_3. Step 2 illustrates what is known by the players of each quorum after Step 1.

In Steps 3 and 4, the players of all three quorums run HW-MPC, using all of their inputs, in order to compute the value $g(X, Y) + R_3$. This is the output of the gate g on the inputs, masked by the random value R_3. The gate quorum now knows 1) the output of the gate plus the value of R_3; and 2) shares of R_3. Thus the gate quorum may serve as an input quorum to a higher level gate in the circuit.

Commitment Tree: We use a Commitment Tree to ensure that inputs of $n - t$ of the players are committed to before the circuit evaluation occurs. A Commitment Tree is a complete binary tree with n leaf nodes, where each node of the tree is a quorum. Each of the n players is assigned to one leaf node, and a key job of the tree is to count the number of players that have properly committed an input value to the appropriate quorum. The tree enables efficient input commitment with asynchronous communication, and may be of use for other problems beyond MPC.

Consider a quorum Q in the tree. A naive approach to counting input commitments is for each child of Q to send Q a message with its number of received inputs, every time there is a change in this number. The problem with this approach is that it is not load-balanced: each quorum at depth i has $\frac{n}{2^i}$ descendants in the tree, and therefore, in the worst case, sends $\frac{n}{2^i}$ messages to its parent. Thus a child of the root, sends $n/2$ messages to the root. To solve the load-balancing problem, we require a clever "halving trick" in which each quorum aggregates a certain number of messages before sending them to its parent. We describe details of this method in Section 3.

3 Our Algorithm

Our algorithm makes use of a *circuit graph*, G, that is based on the circuit that computes f. The circuit graph is a directed acyclic graph over $m + n$ nodes. There are n of these nodes, one per player, that we call the *input nodes*. There are m remaining nodes, one per gate, that we call the *gate nodes*. For every pair of gate nodes x and y, there is an edge from x to y iff the output of the gate represented by node x is an input to the gate represented by node y. Also for any input node z and gate node y, there is an edge from z to y if the player represented by gate node z has an input that feeds into the gate represented by node y.

For any two nodes x and y in G, if x has an edge to y, we say that x is a *child* of y and that y is a *parent* of x. Also, for a given node v, we will say the *height* of v is the number of edges on the longest path from v to any input node in G.

We number the nodes of G canonically with $1, 2, \ldots m + n$ from the input nodes up, in such a

way that the input node numbered i corresponds to player i. We refer to the node corresponding to the output gate as the *output node* and denote it **outnode**.

Algorithm 1 consists of four main parts. The first part is to run CREATE-QUORUMS in order to agree on n good quorums. The second part of the algorithm is INPUT-COMMITMENT. In this part, players form the Commitment Tree. Then, each player i *commits* its input values to quorum i at the leaf nodes of the Commitment Tree and finally the players in that quorum decide whether these values are part of the computation or not. The details of how this part of the algorithm works is described in Section 3.1. The third part of the algorithm is evaluation of the circuit (Step 3 of Algorithm 1, described in detail in Section 3.2). Finally, the output from the circuit evaluation is sent back to all the players by performing Algorithm 10.

Algorithm 1 Main Algorithm

1. All players run CREATE-QUORUMS
2. All players run INPUT-COMMITMENT
3. All players participate in the computation of the circuit:
   (a) For each input node v, if p is in the quorum for v, then after finishing INPUT-COMMITMENT with $mVal(v)$ and a share of $mask(v)$ as its output, p runs GATE-COMPUTATION on each parent node of v with inputs $mVal(v)$ and a share of $mask(v)$.
   (b) For each gate node v, if p is in the quorum for v:
      i. p runs GENERATE-MASK on v and gets a share of $mask(v)$ as output
      ii. p runs GATE-COMPUTATION on v with its share of $mask(v)$ as the input and gets $mVal(v)$ as output
      iii. p runs GATE-COMPUTATION on each parent node of v with input $mVal(v)$ and p's share of $mask(v)$
   (c) After finishing computation of the gate represented by the output node, the players at the output node reconstruct the output (Algorithm 9).
4. All players perform OUTPUT-PROPAGATION (Algorithm 10)

3.1 INPUT-COMMITMENT

The algorithm INPUT-COMMITMENT (Algorithm 2) makes use of the Commitment Tree. The Commitment Tree is a complete binary tree with n leaf nodes, one for each player. We assume a canonical numbering of the quorums 1 through n. We also assume a canonical numbering of nodes in the Commitment Tree, starting with the leaf nodes and numbering left to right and bottom up. We assign the quorum numbered i to any node in the Commitment Tree with number j s.t. (j mod n) $= i$. This assignment ensures that the quorum mapped to leaf node i in the Commitment Tree is the same as the quorum mapped to input node i in G.

When we say quorum A sends a message M to quorum B, we mean that every (good) player in quorum A sends M to every player in quorum B. A player in quorum B is said to have received M from A if it receives M from at least 7/8 of the players in A.

The Commitment Tree is used for input commitment as follows. Let v_i denote the leaf node in the Commitment Tree associated with player i and let Q_i denote the quorum assigned to ths node. Player i samples a value uniformly at random from F and sets $mask(v_i)$ to this value. Let

Q_i be the quorum associated with leaf node v_i in the Commitment Tree. Player i sets $mVal(v_i)$ to $x_i + mask(v_i)$ and send $mVal(v_i)$ to all players in Q_i. Next, player i uses the algorithm AVSS-SHARE to commit to the secret value $mask(v_i)$ to all players in Q_i. Once player i has verifiably completed this process for the values x_i and $mask(v_i)$, we say that player i has *committed* its masked value to Q_i, and each player j in Q_i then sets a bit $b_{i,j}$ to 1. If player i's shares *fail* the verification process, then the $b_{i,j}$'s are set to 0, which is also the value they default to when not explicitly set to 1.

The Commitment Tree determines when at least $n-t$ inputs have been committed. The process starts with the quorums at the leaf nodes of the Commitment Tree. When player i commits its value to quorum Q_i using the process stated above, then the players in Q_i notify the players in the parent quorum of Q_i that player i has successfully completed the commit process.

The players in the quorum associated with each internal node v of the Commitment Tree maintain four quantities λ_v, ρ_v, σ_v and τ_v.

- λ_v and ρ_v represent the input count received by v from its left and right children. Initially, $\lambda_v = \rho_v = 0$ for all v.
- σ_v represents the input count thus far sent to v's parent. This is also zero initially.
- τ_v represents the input count v waits to receive from its children, before communicating with its parent, or in the case of the root, before sending the computation triggering message down the tree. Initially τ_v is $\left\lceil \frac{n-t}{2^d} \right\rceil$; where d is the depth of the node.

Each player p in the quorum at node v runs INPUT-COMMITMENT to continually update these values based on messages received from quorums associated with nodes that are either a parent or child of v. The main idea is that player p will communicate its input count to its parent only when the increase in input count, since the last communication to the parent, exceeds half the number of inputs the parent is waiting for. When at least $n-t$ inputs have been detected at the root, the quorum at the root sends a $\langle$DONE$\rangle$ message to its two children. This is forwarded by each receiving node to its children and eventually gets back to the leaf nodes.

When a player j in Q_i receives the $\langle$DONE$\rangle$ message, j participates in a HW-MPC with other members of Q_i, using $b_{i,j}$ as its input. This HW-MPC determines if at least 5/8 of the bits are 1. If they are, then the quorum decides that $i \in S$ and uses the received value of $mVal(v_i)$ and shares of $mask(v_i)$ as their input into GATE-COMPUTATION. Otherwise they set $mVal(v_i)$ to the default input and the shares of $mask(v)$ to 0. We call this the 5/8-MAJORITY step.

3.2 Evaluating the Circuit

We assign nodes of G to quorums as follows. The output node of G is assigned to quorum 1; then every other node in G numbered i is assigned to quorum number j where $j = (i \mod n)$.

Assume player j is in quorum Q_v at the leaf node v of Commitment Tree, which is the same quorum assigned to input node v in G. The computation phase of the protocol for player j starts after the completion of the 5/8-MAJORITY step for node v in INPUT-COMMITMENT algorithm.

The first step is to generate shares of uniformly random field elements for all gate nodes. If player j is in a quorum at the gate node v, he generates shares of $mask(v)$, a uniformly random field element, by participating in the GENERATE-MASK algorithm. These shares are needed as inputs to the subsequent perform of HW-MPC algorithm.

Next, players begin computation of the gate associated with each gate node (Algorithm 8). Let Q_v be the quorum assigned to node v. For each input node v, the players in Q_v know the

masked input $mVal(v)$ and each has a share of the random element $mask(v)$, although the actual input and mask are unknown to any single player. From the bottom (input gates) of the circuit to the top, we ensure the following gate invariant: For each gate node v with children $\ell(v)$ and $r(v)$, let $x = value(\ell(v))$ and $y = value(r(v))$ be the inputs to the gate associated with v and let $value(v)$ be the output of v as it would be computed by a trusted party. Then before participating in HW-MPC each player in Q_v has a share of the random element $mask(v)$ (via Algorithm 7), and each player in $Q_{\ell(v)}$ (resp. $Q_{r(v)}$) has the masked value $mVal(\ell(v))$ and a share of $mask(\ell(v))$ (resp.$mVal(r(v))$ and a share of $mask(r(v))$). These are their inputs into the HW-MPC at v. After participating in HW-MPC, each player in Q_v learns the masked value $mVal(v)$, which is equal to $value(v) + mask(v)$. Of course they still have the shares of $mask(v)$, so they are now prepared for the computation at the gates which are parents of v. Note that both $value(v)$ and $mask(v)$ themselves are unknown to any individual.

The output of the quorum associated with the output node in G is the output of the entire algorithm. Thus the quorum $Q_{\mathbf{outnode}}$ will unmask the output (Algorithm 9). The last step of the algorithm is to send this output to all players. We do this using a complete binary tree rooted at the output quorum. (see Algorithm 10).

Algorithm 2 INPUT-COMMITMENT

Run the following algorithms in parallel:

1. Algorithm IC-PLAYER.
2. Algorithm IC-INPUT.
3. Algorithm IC-INTERNAL.
4. Algorithm IC-ROOT

Algorithm 3 IC-PLAYER

Player i with input x_i

1. $Q_i \leftarrow$ the quorum at the leaf node v_i associated with input of player i
2. Sample a uniformly random value from $\mathbb{F}$ and set $mask(v_i)$ to this value.
3. $mVal(v_i) \leftarrow x_i + mask(v_i)$
4. Send $mVal(v_i)$ to all the players in the quorum for the ith input.
5. Use the algorithm AVSS-SHARE to commit the secret value $mask(v_i)$ to the players in Q_i.

3.3 Some Remarks

Note that it may be the case that a player p participates $k > 1$ times in the quorums performing a single instance of HW-MPC. In such a case, we allow p to play the role of k different players in the

Algorithm 4 IC-INPUT

Player j in Quorum Q_i associated with node v_i

1. After receiving $mVal(v_i)$ and a share of $mask(v_i)$, participate in the AVSS-SHARE verification protocol and agreement protocol to determine whether consistent shares for $mask(v_i)$ and the same $mVal(v_i)$ are sent to everyone.

2. If the AVSS-SHARE verification protocol and the agreement protocol end and it is agreed that $mVal(v_i)$ was the same for all and shares of $mask(v_i)$ are valid and consistent, set $b_{i,j} \leftarrow 1$ and send your parent quorum the message $\langle 1 \rangle$.

3. Upon receiving $\langle \text{DONE} \rangle$ from your parent quorum, participate in 5/8-MAJORITY using $b_{i,j}$ as your input. If it returns FALSE, reset $mVal(v_i)$ to the default and your share of $mask(v)$ to 0. In this case $i \notin S$.

Algorithm 5 IC-INTERNAL

Player in Quorum Q_v at Internal Node v

1. $\lambda_v \leftarrow 0; \quad \rho_v \leftarrow 0; \quad \sigma_v \leftarrow 0; \quad \tau_v \leftarrow \left\lceil \frac{n-t}{2^{\text{depth}(v)}} \right\rceil; \quad \text{done} \leftarrow \text{false}$

2. while not done
   (a) receive message M from a quorum Q
   (b) If $M = \langle \text{DONE} \rangle$ from and Q is your parent quorum
   - forward M to both your children.
   - done $\leftarrow$ true

   (c) Else if $M = \langle \text{DONE} \rangle$ and Q is your left (resp. right) child quorum,
   - $\lambda_v \leftarrow \lambda_v + x$ (resp. $\rho_v \leftarrow \rho_v + x$)
   - Go to step (e).

   (d) Else if $M = \langle \lambda, \rho, \sigma, \tau \rangle$ and Q is your parent quorum
   - If you are the left child and $\sigma_v > \lambda$ then $\lambda \leftarrow \sigma_v$.
     If you are the right child and $\sigma_v > \rho$ then $\rho \leftarrow \sigma_v$.
   - $\tau_v \leftarrow \lceil \frac{\tau - (\lambda + \rho - \sigma)}{2} \rceil$
   - Go to step (e).

   (e) If $\lambda_v + \rho_v - \sigma_v \geq \tau_v$,
   - $\sigma_v \leftarrow \sigma_v + \tau_v$.
   - Send message $\langle \sigma_v \rangle$ to your parent quorum.

   Otherwise send message $\langle \lambda_v, \rho_v, \sigma_v, \tau_v \rangle$ to both your children.

Algorithm 6 IC-ROOT

Player in the quorum Q_{outnode} at the output node outnode

1. $\lambda_v \leftarrow 0$; $\rho_v \leftarrow 0$; $\tau_v \leftarrow n - t$; done $\leftarrow$ false.
2. while not done
   - Upon receiving message $\langle x \rangle$ from your left (resp. right) child,
     - $\lambda_v \leftarrow \lambda_v + x$ (resp. $\rho_v \leftarrow \rho_v + x$)
     - If $\lambda_v + \rho_v \geq \tau_v$,
       - send the $\langle$DONE$\rangle$ message to both your children.
       - done $\leftarrow$ true
     - Otherwise
       - $\tau_v \leftarrow \tau_v - (\lambda_v + \rho_v)$.
       - Send message $\langle \lambda_v, \rho_v, \lambda_v + \rho_v, \tau_v \rangle$ to both your children.

Algorithm 7 GENERATE-MASK

This protocol is run by each player p in a quorum associated with each gate node $v \in G$ to generate $mask(v)$.

1. Choose uniformly at random an element $r_{p,v} \in \mathbb{F}$ (this must be done independently each time this algorithm is run and independently of all other randomness used to generate shares of inputs etc.)
2. Use AVSS-SHARE and Create verifiable secret shares of $r_{p,v}$ for each player in the quorum associated with v and deal these shares to all the players in the quorum associated with v including itself.
3. Participate in the AVSS-SHARE verification protocol for each received share. If the verification fails, set the particular share value to zero.
4. Add together all the shares (including the one it dealt to itself). This sum will be player p's share of the value $mask(v)$.

Algorithm 8 GATE-COMPUTATION

For each gate node v with children $\ell(v)$ and $r(v)$, the participants are the players in Q_v, $Q_{\ell(v)}$ and $Q_{r(v)}$.

1. If you are a player in $Q_{\ell(v)}$, (resp. $Q_{r(v)}$) use ($mVal(\ell(v))$, share of $mask(\ell(v))$) (resp. ($mVal(r(v))$, share of $mask(r(v))$)) as your input to HW-MPC. If you are a player in Q_v, use share of $mask(v)$ as your input to HW-MPC.
2. Participate in HW-MPC.
3. $mVal(v) \leftarrow$ value returned by HW-MPC.

Algorithm 9 OUTPUT-RECONSTRUCTION

This protocol is run by all players in $Q_{\mathbf{outnode}}$.

1. Reconstruct $mask(\mathbf{outnode})$ from its shares using AVSS-RECONSTRUCT.
2. Set the output $\mathbf{o} \leftarrow mVal(\mathbf{outnode}) - mask(\mathbf{outnode})$.
3. Send $\mathbf{o}$ to all players in the quorums numbered 2 and 3

Algorithm 10 OUTPUT-PROPAGATION

Performed by the players in each quorum Q other than the quorum numbered 1.

1. $i \leftarrow$ quorum number of Q
2. Receive $\mathbf{o}$ from the quorum numbered $\lfloor i/2 \rfloor$.
3. Send $\mathbf{o}$ to all the players in quorums numbered $2i$ and $2i+1$.

secure MPC, one for each quorum to which p belongs. This ensures that the fraction of bad players in any instance of HW-MPC is always less than 1/4. HW-MPC maintains privacy guarantees even in the face of gossiping coalitions of constant size. Thus, player p will learn no information beyond the output and its own inputs after running this protocol.

Also note that although we have not explicitly included this in INPUT-COMMITMENT, it is very easy for the players to compute the size of the computation set S. Once each input quorum Q_i has performed the 5/8-MAJORITY step and agreed on the bit $b_i = \mathbf{1}_{i \in S}$ they can simply use the Commitment Tree as an addition circuit to add these bits together and then disperse the result. Note that this is a multiparty computation, all of whose inputs are held by good players, since each input bit b_i is jointly held by the entire quorum Q_i and all the quorums are good. Thus the computation can afford to wait for all n inputs and will compute the correct sum.

4 Conclusion

We have described a Monte Carlo algorithm to perform asynchronous Secure Multiparty Computation in an efficient manner. Our algorithms are efficient in the sense that they require each player to send $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$ messages and perform $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$ computations. They tolerate a static adversary that controls up to a $1/8 - \epsilon$ fraction of the players, for ϵ any positive constant.

Many open problems remain including the following. 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 a MPC problem such as the beet auction problem described in [6]. Finally, can we prove upper and lower bounds for resource costs to solve MPC in the case where the adversary is *dynamic*, able to take over players at any point during the algorithm?

References

[1] B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: efficient verification via secure computation. *Automata, Languages and Programming*, pages 152–163, 2010.

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

[3] Z. Beerliova and M. Hirt. Efficient multi-party computation with dispute control. In *Theory of Cryptography Conference*, 2006.

[4] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-Fifth ACM Symposium on the Theory of Computing (STOC)*, 1993.

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

[12] W. Du and M.J. Atallah. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on New security paradigms*, pages 13–22. ACM, 2001.

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

[14] K.B. Frikken. Secure multiparty computation. In *Algorithms and theory of computation handbook*, pages 14–14. Chapman & Hall/CRC, 2010.

[15] O. Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 1998.

[16] W. Henecka, A.R. Sadeghi, T. Schneider, I. Wehrenberg, et al. Tasty: Tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462. ACM, 2010.

[17] M. Hirt and U. Maurer. Robustness for free in unconditional multi-party computation. In *Advances in CryptologyCRYPTO 2001*, pages 101–118. Springer, 2001.

[18] M. Hirt and J. Nielsen. Upper bounds on the communication complexity of optimally resilient cryptographic multiparty computation. *Advances in Cryptology-ASIACRYPT 2005*, pages 79–99, 2005.

[19] V. King, S. Lonergan, J. Saia, and A. Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2011.

[20] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the Symposium on Discrete Algorithms(SODA)*, 2006.

[21] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *Foundations of Computer Science(FOCS)*, 2006.

Appendix

A Analysis

In this section, we give the proof of Theorem 1.1.

We begin by noting that the error probability in Theorem 1.1 comes entirely from the possibility that CREATE-QUORUMS may fail to result in good quorums (see Theorem 2.1). All other components of our algorithm are exact algorithms with no error probability.

For the remainder of this section we will assume that we are in the good event, *i.e.* that the players have successfully formed n good quorums.

Lemma A.1. If quorum A sends quorum B a message M, eventually it is received by all good players in B.

Proof. Recall that quorum A sends a message M means that all good players in the quorum send the same message M. A player in B considers himself to have received the message M from A if he receives it from at least $7/8$ of the players in A. Since n good quorums have been successfully formed, more than $7/8$ of the players in each quorum are good. In particular, this is true for A. Thus at least $7/8$ of the members of A send M to each member of B. Since the adversary must eventually deliver all the messages that have been sent, albeit with arbitrary delays, it follows that eventually each good player in B must receive M from at least $7/8$ of the members of A. □

We remark that although the above lemma shows that a quorum A can send a message M to another quorum B, there is some subtlety involved in using this fact in the algorithm. The players in a quorum are going to communicate with their parent when they have at least half as many inputs as the parents threshold. However, since among other reasons multiple messages may arrive simultaneously, when the threshold is set, not all players 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 A.1 tells us that if all the players 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 the players in the quorum send the same message to the parent quorum, we have required that even if a player's received inputs exceed his threshold, he should only inform the parent of having met the threshold, not of having exceeded it. The remaining inputs are held, to be sent later.

We now show the correctness of INPUT-COMMITMENT.

We'll say that a quorum has come to agreement on X if all good players in the quorum agree on X.

Lemma A.2. INPUT-COMMITMENT eventually results in all good players receiving the ⟨DONE⟩ message. Moreover, there exists a set S

1. For all $1 \leq i \leq n$, quorum Q_i eventually agrees on whether or not player i is in S;
2. There are at least $n - t$ quorums Q_i for $1 \leq i \leq n$ such that Q_i agrees that player i's input is in S;
3. For $1 \leq i \leq n$, quorum Q_i agrees that player i is in S if and only if the players in Q_i collective have enough shares to reconstruct player i's input. If Q_i agrees that $i \in S$ then player i's input will be used to compute the output of the algorithm; if not, the predetermined default input value will be used.

Proof. The ⟨DONE⟩ message is sent by the root quorum when it has detected at least $n-t$ inputs in the tree. By Lemma A.1, once the ⟨DONE⟩ message is sent, it is eventually received and forwarded by all good players in the roots children. Since we are conditioning on all quorums being good, it follows by induction that the message is eventually received by all the input quorums and therefore all the good players.

Thus to prove that all players eventually receive the ⟨DONE⟩ message, it suffices to prove that INPUT-COMMITMENT eventually results in the root quorum detecting $n - t$ inputs. We will proceed by an inductive argument. First note that since there are $n - t$ good players, it follows that at least $n - t$ valid inputs are eventually committed to at the input level, *i.e.*, there are at least $n - t$ inputs in the subtree below the root. We will show that at any stage of the algorithm, a node with a threshold of k which actually has k previously unaccounted inputs in the subtree below it will eventually detect k inputs. As a base case, this is certainly true for the internal nodes just above input level, since the input nodes send along the message $\langle 1 \rangle$ as soon as they have an input. So whether a height 1 node has its threshold at 2 or has reduced it to 1, if it has that many inputs below it, it will eventually detect them. Consider a node v at height h, which has at some point in the algorithm, just set its threshold at k. We will make two induction hypotheses. The first is that if a node at height at most $h - 1$ has at least as many inputs in its subtree as its threshold, it will eventually detect those inputs. The second is that if a node with height h and a threshold strictly less than k, has at least as many inputs in its subtree then it will eventually detect them. Now suppose v, which has height h and threshold k, has at least k previously unaccounted inputs below it. Then its children, which are at height $h - 1$, set their thresholds at $\lceil k/2 \rceil$. By the pigeonhole principle, at least one of the children has at least $\lceil k/2 \rceil$ inputs in its subtree. By the first hypothesis, that child will eventually detect those inputs. At that point it will notify its parent Q of $\lceil k/2 \rceil$ inputs. v now needs only $\lfloor k/2 \rfloor$ more inputs, and since its subtree originally contained at least k previously unaccounted inputs, after accounting for the $\lceil k/2 \rceil$, it still contains at least $\lfloor k/2 \rfloor$ inputs. By the second hypothesis, v eventually detects those inputs, and therefore has, at that point, detected k inputs. By induction, applying this to the root node, we see that the root node eventually detects $n - t$ inputs, as claimed.

Now, for each i, consider the players in Q_i. Let $q = |Q_i|$. Each player in Q_i has set a bit to either 1 or 0 depending on whether or not he has received a valid input shares from player i. Upon receiving the ⟨DONE⟩ message, the players in Q_i perform 5/8-MAJORITY to decide whether at least $\frac{5q}{8}$ of them have set the bit to 1. If they have, they decide that $i \in S$.

Note that if Q_i is one of the quorums that has contributed to the root's threshold of $n - t$ then at least $\frac{7q}{8}$ of the players in Q_i received input shares from player i before they received the ⟨DONE⟩ message. Of these, more than $\frac{3q}{4}$ players are good and have set their bit to 1, since Q_i contains less than $q/8$ bad players. Since the asynchronous HW-MPC computes even if as many as $q/8$ inputs are missing, the players in Q_i will correctly decide that at least $\frac{5q}{8}$ bits among them are set to 1. Thus Q_i will agree that $i \in S$. It follows that at least $n - t$ input quorums will agree that the corresponding input is in S, and therefore $|S| \geq n - t$.

Now suppose Q_i is not among the quorums accounted for in the root's threshold. The players in Q_i will agree that $i \in S$ if and only if HW-MPC detects at least $\frac{5q}{8}$ bits set to 1. Since Q_i contains less than $q/8$ bad players, this means that more than $q/2$ players are good *and* have set their bit to 1, *i.e.* they have received valid input shares from player i. But this is sufficiently many *good* shares to run AVSS-RECONSTRUCT. Thus we have shown that the players in Q_i will agree that $i \in S$ if and only if i really has committed to an input. □

We now show the correctness of GATE-COMPUTATION.

For a leaf node $v \in G$ that is numbered i for some $1 \leq i \leq n$, if player i is in S, we define $value(v)$ to be x_i. If player i is not in S, we define $value(v)$ to be the predefined value. For a gate node $v \in G$, we define $value(v)$ to be the output of the gate associated with v in the circuit, when using the given input values for all players in S and the predefined value for all players not in S.

Also for each node v recall that we have a field element $mask(v) \in \mathbb{F}$. If v is an input node corresponding to player $i \in S$, $mask(v)$ is the field element selected by player i in INPUT-COMMITMENT. If v is an input node corresponding to player $i \notin S$, $mask(v)$ is 0. If v is a gate node $mask(v)$ is the random field element jointly generated using GENERATE-MASK by the quorum at v.

Let V be the set of all nodes in G which are either gate nodes or input nodes corresponding to good players in S.

Lemma A.3. $\{mask(v)\}_{v \in V}$ are fully independent and uniformly random in $\mathbb{F}$. Moreover, these are also independent of the set of all other masks.

Proof. The masks corresponding to input nodes for good players in S are uniformly random by choice (see IC-PLAYER). To see that the masks for the gates are uniformly random, recall that if v is a gate node $mask(j) = \sum_{j \in Q_v} r_{v,j}$ where $r_{j,i}$ is the value selected by player j in GENERATE-MASK. The players commit to the $r_{v,j}$ values by running AVSS-SHARE If player j is good, $r_{v,j}$ is uniformly random. If player j is bad then $r_{v,j}$ could be anything. However, once the players have committed to the values the bad players can no longer influence the sum of the $r_{v,j}$, nor can they bias the distributions of the $r_{v,j}$ in any way, because of the correctness of AVSS-SHARE. Since the sum of elements of $\mathbb{F}$ is uniformly random if at least one of them is uniformly random, it follows that $mask(v)$ is uniformly random. The independence follows from the fact that all good players have sampled their values independently, and although the bad players have access to some of the shares $r_{v,j}$ they do not have enough to reconstruct any of the masks, and this means the shares contain no information about the masks. □

Lemma A.4. For each node v in G, eventually all good players in Q_v agree on a field element $mVal(v) \in \mathbb{F}$ such that $mVal(v) - mask(v) = value(v)$. Moreover if v is either a gate or an input node corresponding to $i \in S$, each player in the quorum Q_v has a AVSS share of $mask(v)$. If v is an input node for $i \notin S$ then $mask(v)$ and its shares are 0.

Proof. First consider an input node v corresponding to player i. By Lemma A.2, the players in Q_i have eventually either received consistent AVSS shares of $mask(v)$ and agreed on $mVal(v)$ as well as on i being in S or else they have agreed that $i \notin S$ and have set $value(v)$ and $mVal(v)$ to the predefined value and $mask(v)$ and all its shares to 0.

Now let v be a gate node. By the correctness of GENERATE-MASK the players in Q_v have AVSS shares of $mask(v)$. We prove the claim about $mVal(v)$ by induction on the height of v. As a base case, we already know from above that for input nodes v the players in Q_v agree on $mVal(v)$. Suppose for all nodes v' whose height is less than the height of v, that all the good players in $Q_{v'}$ agree on a field element $mVal(v')$ and that this differs from $value(v)$ by the field element $mask(v)$ they implicitly agree on through the shares they hold. Then the inductive hypothesis holds for the children of v, which we will call u and w. In the computation at node v, the three quorums, Q_v, Q_u and Q_w run HW-MPC with inputs $mVal(u), mVal(w)$ and the shares of $mask(v), mask(u)$ and $mask(w)$. By correctness of HW-MPC, eventually all good players terminate the computation and agree on a common value.

To see that this common value is actually $mask(v)+value(v)$ we note that the function computed by HW-MPC consists of the following steps: 1) reconstructing $mask(v)$, $mask(u)$ and $mask(w)$ from their shares, 2) inferrring $value(u)$ and $value(w)$ from the corresponding masks and masked values; 3) computing $value(v)$ from $value(u)$ and $value(w)$; and 4) adding $mask(v)$ back in to get $mVal(v)$. All of this will, of course, be opaque to the players involved. Attempts to corrupt this computation by lying about $mVal(u)$ or $mVal(w)$ are easily thwarted, because of the high redundancy in these as inputs. For each of these masked values many more players provide them correctly as try to lie, since each of Q_u and Q_w have at most 1/8th bad players. Moreover, the AVSS-RECONSTRUCT used to infer the masks from their shares can tolerate up to a fourth of the shares being corrupted. Thus, since all quorums are good, even if the bad players lie about their shares of the masks, they cannot change the value of the computation. It follows that $mVal(v) = value(v) + mask(v)$. By induction, all the nodes in G compute the correct masked values. □

Corollary A.5. Eventually all players at the output node will learn $value($**outnode**$)$.

Proof. By Lemma A.4, eventually all the players at the output node agree on $mVal($**outnode**$)$ and have shares of $mask($**outnode**$)$. During OUTPUT-RECONSTRUCTION, these players run the AVSS-RECONSTRUCT. Since at least 7/8ths of them are good, they correctly reconstruct $mask($**outnode**$)$. Since all the good players at the output node know $mVal($**outnode**$)$, subtracting from it the reconstructed $mask($**outnode**$)$, they all eventually learn $value($**outnode**$)$. □

Lemma A.6. Eventually $value($**outnode**$)$ is learned by all good players.

Proof. This follows by induction. Since quorum 1 is at the output node, Corollary A.5 provides a base case. Now suppose the correct output has been learned by all the players in quorums numbered j for all $j < i$. Consider the players in quorum i. During the run of OUTPUT-PROPAGATION, they will receive putative values for the output from the players at quorum $\lfloor i/2 \rfloor$. Since quorum $\lfloor i/2 \rfloor$ is good, and by induction hypothesis all good players in it have learned the correct output, it follows that all good players in quorum $\lfloor i/2 \rfloor$ send the same message, which is the correct output. By Lemma A.1 eventually all good players in quorum i learn the correct output. By induction, all the players learn the correct value. □

A.1 Input Privacy

We devote the rest of the section to showing that privacy of inputs is preserved. All results in this section are conditioned on the event that all the quorums are good.

Recall that V is the set of nodes in G that are either gate nodes or input nodes corresponding to good players in S.

Lemma A.7. Let v be any node in V other than the output node. Using only messages received by him as part of the algorithm, no player can learn any information about $value(v)$, except what is implicit in his own input and the final output $value($**outnode**$)$ of the circuit.

Proof. We prove this for a gate node v. By Lemmas A.3 and A.4, the value recovered by HW-MPC during the computation of g is $mVal(v) = value(v)+mask(v)$, where $mask(v)$ is a uniformly random element of $\mathbb{F}$, independent of all other randomness in the algorithm. In particular this means that $mVal(v)$ holds no information about $value(v)$. If player i is not in any of the quorums at v or its neighbors, then all the messages he receives during the algorithm are independent of $mask(v)$, and hence $mVal(v)$, and hence he cannot learn anything about $value(v)$. On the other hand, if player i is involved in the computation of v or one of its neighbors, then he may hold a share $mask(v)$

as well as shares of the individual random field elements whose sum is $mask(v)$. In this case we appeal to the privacy of HW-MPC and the embedded VSS algorithm to see that although he may learn $mVal(v)$, he cannot learn any additional information about the shares of $mask(v)$ and hence about $mask(v)$ itself. Thus, he cannot learn any information about $value(v)$ except what is implicit in his input and the circuit output $value(\mathbf{outnode})$.

The proof for an input node of a good player in S is similar except that we will have to appeal to the privacy of AVSS-SHARE rather than the privacy of HW-MPC. □

Lemma A.8. The input privacy of player $i \notin S$ is preserved.

Proof. The security of AVSS-SHARE ensures that player i's input is not discovered by the players in the Q_i. Since $i \notin S$, i's input is not used in any further part of the algorithm, no other messages sent in the algorithm can contain any information about it. □

We now explore a stronger notion of privacy. Ben-Or *et al.* [5] distinguish between the two kinds of deviant behaviour among players. The bad players are players controlled by an adversary who may indulge in arbitrary kinds of erratic behaviour to try to break the protocol in any way they can. However Ben-Or *et al.* also consider players who are good, in the sense that they follow the protocol, but may also send and receive messages external to the protocol, to attempt to learn whatever additional information they can. Such players are called *gossiping* players. A protocol is called t-private if no coalition of size t (including coalitions of gossiping players) can learn anything more than what is implied by their private inputs and the circuit output. The synchronous MPC protocol of [5] is $(n/3 - \delta)$-private for any $\delta > 0$.

We note that our algorithm *is* susceptible to adaptively chosen coalitions of gossiping players. Indeed, if all the players in a quorum at a node v gossip with each other, they can reconstruct $mask(v)$ and hence $value(v)$. In particular, the players in the quorum at an input node can jointly reconstruct the corresponding input.

However, we can establish the following result, which shows that for large coalitions chosen non-adaptively (in particular, the adversarial players) our algorithm will preserve privacy.

Lemma A.9. Let A be any set of players such that for every quorum Q, $A \cap Q$ contains fewer than half of the players in Q. Let v be any node in V. Then the coalition A cannot learn any information about $value(v)$ that cannot be computed from their (collective) private inputs and the circuit output $value(\mathbf{outnode})$.

Proof. Once again we prove this only for gate node v in V. The proof for an input node is similar. We know that HW-MPC when run at v computes $value(v) + mask(v)$, where $mask(v)$ is uniform in $\mathbb{F}$ and independent of all other randomness in the algorithm. As noted in the proof of Lemma A.7, the players in A who are not in the quorums at v or any of its neighbors are irrelevant to the coalition: all of the information that they hold is completely independent of $mask(v)$ and $mVal(v)$, so they cannot assist in uncovering any information about $value(v)$, except what is implicit in their private inputs.

Now consider the players in the quorums at v or any of its neighbors. These players participate in one or more of the instances of HW-MPC which involve v: the computation of v itself or the computations in which the output of v is an input. Here we appeal to the privacy of HW-MPC to see that the players cannot learn any additional information that is not implied by their inputs. The players in A are unable to directly determine $mask(v)$, since the only relevant inputs are the shares of $mask(v)$, and they do not have enough of those.

Finally, let us consider the players from A at v itself. These players also do not have enough shares of $mask(v)$ to reconstruct it on their own. However, they recieve shares of each of the other shares of $mask(v)$ multiple times: once during the input commitment phase of each HW-MPC in which v is involved. Each time, they do not get enough shares of shares $mask(v)$ to reconstruct any shares of $mask(v)$. However, can they combine the shares of shares from different runs of AVSS-SHARE for the same secret to gain some information? Since fresh, 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 AVSS-SHARE does not give the players in A enough shares to reconstruct anything, it follows that they do not learn any information about $mask(v)$. Since $mask(v)$ is uniformly random, so is $mask(v)$ and it follows that the coalition A cannot get any extra information about $value(v)$. □

Corollary A.10. The bad players cannot learn any information, except what is implied by the output and the inputs to which they committed, about the input of any good player.

Proof. This follows immediately from Lemma A.9, since each quorum consists of no more than 1/8th bad players. □

Let q be the size of the smallest quorum. Recall that $q = \Theta(\log n)$.

Corollary A.11. Our algorithm is $(\frac{q}{2} - 1)$-private.

Proof. Since q is the size of the smallest quorum, any set of size $\frac{q}{2} - 1$ intersects a quorum Q in fewer than half of its members. The result follows from Lemma A.9 □

We end with a simple analysis of the resource cost of our algorithm.

Lemma A.12. During INPUT-COMMITMENT, each node in the Commitment Tree sends at most $O(\log^2 n)$ messages.

Proof. We begin by showing that a node sends at most $O(\log^2 n)$ messages to its children. We first recall that the root starts out with a threshold of $n - t$. This threshold decreases to less than half its previous size each time the root receives a message, and when it reaches zero, the root's role in the algorithm ends. Thus the root sends at most $\log(n - t)$ messages to its children.

Now consider an internal node v. Every time v receives a message from one of its children, its *effective threshold*, $\tau_v - (\lambda_v + \rho_v - \sigma_v)$ decreases to less than half its previous value, and this can happen at most $\log \tau_v < \log(n - t)$ times. On the other hand, every time v receives a message from its parent, the value of τ_v is reset, and in the intervening period, if v has sent a message to its parent, then $\sigma_v = \lambda_v + \rho_v$, so that the effective threshold actually increases. However, the new value of τ_v is at most half the previous one, and the initial value was at most $n - t$, hence this can happen at most $\log(n - t)$ times. Thus v sends at most $O(\log^2 n)$ messages to its children.

Now what about the messages a node v sends to its parent? We note that initially, v may send one message to its parent, when it has received enough inputs from its own children to meet its initial threshold. However, subsequently, every message sent to its parent is preceded by a received message from its parent. So the upward messages sent by v exceed the downward messages sent by v's parent by at most 1, and so this is also $O(\log^2 n)$.

Since each quorum is mapped to at most one internal node of the count tree, it follows that the total number of messages sent by each quorum is $O(\log^2 n)$. □

Lemma A.13. If all good players follow Algorithm 1, w.h.p., each players sends at most $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$ messages.

Proof. By Theorem 2.1 we need to send $\tilde{O}(\sqrt{n})$ messages per player to build the quorums. Subsequently, each player must send messages for each quorum in which he is a member. Recall that each players is in $\Theta(\log n)$ quorums.

By Lemma A.12 each quorum sends $O(\log^2(n))$ messages during INPUT-COMMITMENT. An additional polylog(n) messages are sent by each quorum to determine whether the input it represents is in S.

Recall that each quorum is mapped to $\Theta(\frac{m+n}{n})$ nodes of G. A quorum runs GENERATE-MASK and GATE-COMPUTATION once per node it is mapped to in G. Since each gate has in-degree 2 and out-degree at most 2, a quorum runs HW-MPC at most 3 times for every node it is mapped to in G. polylog(n) messages are sent per player per instance of HW-MPC, GENERATE-MASK and GATE-COMPUTATION. Finally each quorum sends $O(\log n)$ messages in the dissemination of the output.

Thus each quorum sends polylog(n) messages per node which it represents. It follows that each player sends $\tilde{O}(\frac{n+m}{n} + \sqrt{n})$ messages. □

B Asynchronous Scalable Solution to Quorum Formation Problem

This section is a modified reproduction of the paper [21] by King et al. We have changed the algorithms and proofs as necessary to adapt to our asynchronous environment. However, we decided to include all the algorithms and proofs to make it easier for the reader.

We start this section by defining the *semi-random-string agreement problem* in which we want to reach a situation where, for any positive constant ϵ, $1/2 + \epsilon$ fraction of players are good and agree on a single string of length $O(\log n)$ with a constant fraction for random bits. King et al. [19] 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 the solution for *quorum formation* problem. Thus, their techniques can be extended to the asynchronous model assuming a scalable asynchronous solution for semi-random-string agreement problem. We describe CREATE-QUORUMS algorithm based on SRS-TO-QUORUM and an algorithm, that solves semi-random-string agreement problem in the asynchronous model with pairwise channels that we call SRS-AGREEMENT.

Algorithm 11 CREATE-QUORUMS

Performs by all players:

1. All players run SRS-AGREEMENT.
2. All players run SRS-TO-QUORUM

King et al. [21] present a synchronous algorithm that a set of players, up to 1/3 of which are controlled by an adversary, can reach almost-everywhere[3] agreement with probability $1-o(1)$. Their main technique is to divide the players into groups of polylogarithmic size; each player is assigned

[3]King et al. [21] relax the requirement that all uncorrupted players reach agreement at the end of the protocol, instead requiring that a $1 - o(1)$ fraction of uncorrupted players reach agreement. They refer to this relaxation as almost-everywhere agreement.

to multiple groups. In parallel, each group then use bin election algorithm [13] to *elect* a small number of players from within their group to move on. This step is recursively repeated on the set of elected players until size of the remaining players in this set becomes polylogarithmic. At this point, the remaining players can solve the semi-random-string agreement problem (Similarly, they can run bysantine agreement protocol to agree on a bit). Provided the fraction of bad players in the set of remaining players is less than 1/3 w.h.p., these players succeed in agreeing on a semi-random string. Then, these players communicate the result value to the rest of the players.

Bringing players to agreement on a semi-random string is trickier in the asynchronous model. The major difficulty is that the bin election algorithm can not be used in asynchronous model since the adversary can prevent a fraction of the good players from being heard and then prevent them to be part of the election. We present a similar algorithm to [21] that solves this issue in asynchronous model with private channels. The main result of this section is as follows.

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

- With high probability, a $1 - O(1/\ln n)$ fraction of the uncorrupted players agree on the same value (bit or string).
- Every uncorrupted player sends and processes only a polylogarithmic (in n) number of bits.
- The number of rounds required is polylogarithmic in n.

The important novelty of our method compare to King et al. [21] is that instead of bin election algorithm, we use HW-MPC to decide on the players who move on to the next level. The simple version of our election method is presented in Algorithm 12, that has properties as described in Lemma B.2. The complete algorithms and its proof of correctness are in the next sub sections of this appendix.

Algorithm 12 Simple Version of ELECT-SUBCOMMITTEE

Performs by players $p_1, ..., p_k \in W$ with $k = \Omega(\ln^8 n)$.

1. Player p_i generate a vector of $c \ln^3 n$ random numbers chosen uniformly and independently at random from 1 to k where each random number maps to one player.
2. Run HW-MPC to compute the component-wise sum modulo k of all the vectors. Arbitrarily, add enough additional numbers from 1 to k to the sum vector to ensure it has $c \ln^3 n$ unique numbers.
3. Let W_B be the set of winning players which are those associated with the components of the sum vector.
4. Return W_B as the elected subcommittee.

Lemma B.2. Let W be a committee of $\Omega(\ln^8 n)$ players, where the fraction, f_W, of uncorrupted players is greater than 3/4. Then, there exists some constant c, such that w.h.p., the ELECT-SUBCOMMITTEE protocol elects a subset W_B of W such that $|W_B| = c \ln^3 n$ and the fraction of uncorrupted players in W_B is greater than $(1 - 1/\ln n) f_W$. 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 union and Chernoff bounds. Let X be the number of good players in W_B. By the correctness of the HW-MPC algorithm, each player in W_B is randomly chosen from W. Let Y_i be an indicator random variable, that equals to 1 if the i-th member of W_B is good. Then, $E[Y_i] = f_W$ and $E[X] = f_W c_1 \ln^3 n$. Using Chernoff bounds, we have $Pr[X < (1 - 1/\ln n) f_W c_1 \ln^3 n] = Pr[X < (1 - 1/\ln n)E[X]] \leq e^{-\frac{E[X]/\ln^2 n}{2}} < 1/n^c$. Since $f_W > 1/2$, setting $c_1 = 4c$, establishes the first part of Lemma B.2. □

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 players participating in the algorithm.

B.1 The election graph

Our algorithms make use of an election graph to determine which players will participate in which elections. This graph was described in [20, 21] and is repeated here.

Before describing the election graph, we first present a result similar to that used in [8]. Let X be a set of players. 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 B.3. Let l, r, n be positive integers such that l and r are all no more than n and $r/l \geq l n^{1-z} n$. Then, there is a bipartite graph $G(L, R)$ such that $|L| = l$ and $|R| = r$ and

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

The proof of Lemma B.3 follows from a counting argument using the probabilistic method and is omitted.

The following corollaries follows immediately by repeated application of the above lemma.

Corollary B.4. Let ℓ^* be the smallest integer such that $n/\ln^{\ell^*} n \leq \ln^{10} n$. 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 $|L_i| = n/\ln^i n$, $|R_i| = n/\ln^{i+1} n$, and

1. Each node in R_i has degree $\ln^{c_1} n$.
2. Each node in L_i has degree $O(\ln^{c_2} n)$.
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, $|\mathcal{F}(L'_i, 1/\ln n)| < |R_i|/\ln^6 n$.

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, $|\mathcal{F}'(R'_i, 1/\ln n)| < |L_i|/\ln^6 n$.

Corollary B.5. Let ℓ^* be the smallest integer such that $n/\ln^{\ell^*} n \leq \ln^{10} n$. There is a family of bipartite graphs $G(L_i, R_i), i = 0, 1, \ldots, \ell^*$, such that $|L_i| = n/\ln^i n$, $|R_i| = n/\ln^{i+1} n$, and

1. Each node in R_i has degree $\ln^5 n$.
2. Each node in L_i has degree $O(\ln^4 n)$.
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, $|\mathcal{F}(L'_i, 1/\ln n)| < |L_i|/\ln^3 n$.

Lemma B.3 and its corollaries show there exists a family of bipartite graphs with strong expansion properties which allow the formation of subsets of players where all but a small fraction contain a majority that are uncorrupted.

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 $n/\ln^\ell n \leq \ln^{10} n$; note that $\ell^* = O(\ln n/\ln\ln n)$. 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 $n/\ln^{\ell+1} n$ e-nodes. There is an edge between the ith e-node, A, in layer ℓ and the jth e-node, B, in layer $\ell + 1$ iff there is an edge between the ith node in $L_{\ell+1}$ and the jth node in $R_{\ell+1}$ from Corollary B.5. 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 players known as a *committee*. All e-nodes, except for the one on the top layer and those in layer 0, will contain $c\ln^3 n$ players. Initially, we assign the n players to e-nodes on layer 0 using the bipartite graph $G(L_0, R_0)$ described in Corollary B.5. The i^{th} player is a member of the committee contained in the j^{th} e-node of layer 0 iff 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 $\ln^5 n$ players 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 players 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 players in A, $B_1, \ldots, B_s$, run byzantine agreement for Byzantine Agreement.

B.2 The polylogarithmic bounded degree static network.

We now repeat the description of the bounded degree static network [21] 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 players which we call it *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 $1/\ln^{10} n$, 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 players contained in the s-node increases with the layer. Specifically, the s-nodes for layer i are sets of $\ln^{i+12} n$ players. We determine these s-nodes using the bipartite graph from Lemma B.3, where L is a collection of n nodes, one for each player, R is the set of s-nodes for layer

i and the degree of each node in R is set to $\ln^{i+12} n$. The neighbors of each node in R constitute a set of players 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 player x to $s(A)$ on layer i will pass from the player to a layer 0 s-node, whose players 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 to an individual player x from $s(A)$ will be transmitted down to a layer 0 s-node whose players will transmit the message to x. We describe the connections in the static network.

Connections in the static network

- Let A be an e-node on layer 0 in the election graph. Every player in A has a link to every player 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 $\ln^{i+12} n$ players in it and $s(B)$ has $\ln^{i+11} n$ players in it. Let G be a bipartite graph as in Lemma B.3 where L is the set of players in $s(A)$, R is the set of players in $s(B)$ and the degree of R is set to $\ln^{c_1} n$ and the degree of L is set to $O(\ln^{c_2} n)$. If there is an edge between two nodes in L and R respectively, then the corresponding player in $s(A)$ has a link to the corresponding player in $s(B)$. We will sometimes say that $s(A)$ is adjacent to $s(B)$ in the static network.

The following lemma follows easily from the application of Lemma B.3 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 w.h.p., the fraction of bad e-nodes on every layer is small.

Lemma B.6. W.h.p., the election graph and the static network have the following properties:

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

The degree of the static network is polylogarithmic

B.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 player 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 player y in an s-node $s(A)$ on each layer j keeps a *List* of permissible paths which determine which players' messages it will forward. The List (for $y \in s(A)$) represents y's view of which players 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 player, 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 $1 - b - 2/\ln n$ players in the s-node are uncorrupted 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 $1/\ln n$ 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 player communicates with an s-node, and MESSAGEPASS describes how two players 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 player. If a player x sends a message to a layer 0 s-node $s(A)$ it sends the message to every player in $s(A)$ (note by construction it will have a direct link with every player in $s(A)$). Similarly if a message is sent from a layer 0 s-node $s(A)$ to a player x, every player in $s(A)$ sends the message to x.

When an s-node $s(A)$ sends a message to s-node $s(B)$, every player in $s(A)$ sends the message to those players of $s(B)$ to which it has a direct link. When each player 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 player ignores the message. Along with sending the message the players also send information which specifies along which path P the message is being sent. Each time a message is received by a player of an s-node $s(B)$ on layer $j \leq i$, 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 player, messages from that player are dropped.

SEND(s, r, m, P): {Of the first two parameters, one must be a player ("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 player x in e-node A sends message m to player 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.

B.4 The Protocol for SRS-AGREEMENT

Before we describe the SRS-AGREEMENT protocol, we first need to adapt the Elect-Subcommittee protocol for the static network. Let A be an e-node with children $B_1, \ldots, B_s$, and let X be the set of all players from $B_1, \ldots, B_s$. 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 player $x \in X$ generate a vector of random numbers chosen uniformly and independently at random where each random number maps to one player. The players use the HW-MPC protocol to determine the winners. (Recall that the number of players 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 player 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 player $x \in X$. players on the path (i.e. in s-nodes along the path) update their Lists of permissible paths to remove those player-paths who lost as well as those player-pairs who won too many elections (we will quantify this shortly), and make $\ln^4 n$ copies of each of the winners' paths and concatenate a different layer $i+1$ s-node parent onto each one. We present a detailed description of Elect-Subcommittee below.

The condition in Step 5 that requires players who have won more than 8 elections to be eliminated is a technical condition that insures the protocol is load-balanced and players in an s-node do not communicate more than a polylogarithmic number of bits. We now describe the SRS-AGREEMENT protocol.

Note every player is a member of $s(A^*)$, thus Steps 5 and 6 will insure the final result of the protocol is communicated to every player.

B.5 Proofs

To establish the correctness of the protocol presented in Section B.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 players is b, where $b < 1/4 - \epsilon$ for any fix positive ϵ.

Claim B.7. Let $s(A)$ and $s(B)$ be s-nodes on consecutive layers. Assume the following three conditions hold.

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

Algorithm 13 ELECT-SUBCOMMITTEE:

1. For each $x \in X$: *// This phase done in parallel*
2. player x randomly selects one of $k/(c_1 \ln^3 n)$ random numbers chosen uniformly and independently at random from zero to k where each random number maps to one player.
3. players in X run HW-MPC to compute the component-wise sum modulo k of all the vector. Arbitrarily, add enough additional numbers to the vector to ensure it has $c \ln^3 n$ unique numbers.
4. Let M be the set of winning players, 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. The players in $s(A)$ determine M by waiting until they receive the majority of same messages. These become the elected players.
7. For each player $x \in X$ that is elected, the players 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 player in each s-node revises its l ist of permissible paths to:

   Retain only the winners. Eliminate players who have won more than 8 elections. Make $\ln^4 n$ 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 SEND-HOP$(s(A), s(B), m, P)$, where P is the path consisting only of $s(A)$, $s(B)$.

Algorithm 14 SCALABLE-SRS-AGREEMENT:

1. For $l = 1$ to l^*:
2. For each e-node A in layer l, (let $B_1, ..., B_s$ be the children of A in layer $l-1$ of the election graph.)
3. If $l < l^*$, run ELECT-SUBCOMMITTEEon the players in nodes $B_1, ..., B_s$. Assign winning players to node A.
4. Else players in nodes $B_1, ..., B_s$ solve semi-random-string agreement problem.
5. Let A^* be the e-node on layer l^*, every player x assigned to A^* communicates the result of step four to $s(A^*)$ using SEND$(x, s(A^*), m, P_x)$.
6. Every player in $s(A^*)$ waits for the majority of same message to determine the result of step four.

Then there is a set W' of players from $s(B)$ such that for every message m, every player in W' is uncorrupted 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 $1/\ln^6 n$ fraction of the uncorrupted players in $s(B)$.

Proof. Every player in W is good and sends the same massage to its connected players in $s(B)$. The players in $s(B)$ can afford to wait for the majority of same messages, since $s(A)$ is good and W consists of at least $1-b-2/\ln n$ fraction of players which is more than $1/2$ and for majority we need to receive a fraction of $1/2$ same messages from the players in $s(A)$. Thus, all good player but a $1/\ln^6 n$ fraction of players in $s(B)$ will eventually receive the message based on corollary B.4. □

Claim B.8. Let x be an uncorrupted player. 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 uncorrupted players which contains all but a $1/\ln^6 n$ fraction of the uncorrupted players in $s(A)$ and every player $z \in W$ 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 players x participating in the election at A: (1) player x is uncorrupted; (2) The path P_x is good.

Lemma B.9. For a legitimate election at node A, let X be a set of uncorrupted players with good permissible paths. (Note $|X| > 3\ln^8 n/4$.) Let W be the set of uncorrupted players in $s(A)$ that know X. Then after the execution of ELECT-SUBCOMMITTEE, the players 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 B.8, 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 uncorrupted players in $s(B_i)$, such that W contains at least $1-b-2/\ln n$ fraction of the players in $s(B_i)$. By Claim B.7, every message sent by y is received by z. Since X contains more than $3/4$ of the total players participating in the election, (after running HW-MPC) all the players in X will all agree on the same set of for random players. Thus after the players 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 B.7, 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.

B.5.1 The removal of permissible paths from Lists

Let y be a player in some s-node on layer i. A permissible path P_x is removed from a player 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 players 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 [20, 21], shows that on a given layer a very small fraction of uncorrupted players win more than 8 times in legitimate elections.

Lemma B.10. W.h.p., the players that win more than 8 elections, counting multiplicities, account for no more than a $16/\ln^3 n$ fraction of the uncorrupted players 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 uncorrupted players 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 B.11. Suppose less than a $1/7$ fraction of the uncorrupted players of a good $s(A)$ agree on a message m. Then after SENDHOP$(p(A), p(B), m, P)$ is executed, all but a $1/\ln^6 n$ fraction of the good players in $s(B)$ will ignore m.

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

Hence a good $s(A)$ can only communicate 7 different sets of winners to the s-nodes below it. Since each uncorrupted player will send $\ln^3 n$ winners, the total number of winners sent is at most $7\ln^3 n$. Hence a bad e-node can cause at most $7\ln^3 n$ players 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 player participating in the election at A. Since $\ln^8 n$ players participate in an election, and fewer than a $1/\ln^{10} n$ fraction of the s-nodes are bad on a layer, the fraction of uncorrupted winners affected is less than $1/\ln^2 n$. Thus we can bound the fraction of the uncorrupted winners on any layer i that have their permissible paths removed by $1/\ln^2 n + 1/\ln^3 n + 7\beta_i$; where β_i represents the fraction of bad e-nodes on layer i. Thus we have the following lemma.

Lemma B.12. Assume the fraction of bad e-nodes on layer i is bounded by $c/\ln^2 n$, for some constant c. Then the fraction of uncorrupted winners that have their permissible paths removed on layer i is bounded by $8c/\ln^2 n$.

B.5.2 Proof of Theorems B.1

We now complete the proof of Theorems B.1. They will follow from the lemma below.

Lemma B.13. On layer i, w.h.p., at least a $1-4/\ln^2 n$ fraction of s-nodes $s(A_j)$ have the following properties:

- $s(A_j)$ is good.
- At least a $1 - b - 4i/\ln n$ fraction of the players in node A_j are uncorrupted 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 $1/\ln^{10} n$ fraction of the s-nodes are bad. If $s(A)$ is good, then every player in A has a good path to $s(A)$. Further by construction all but a $1/\ln^2 n$ fraction of the e-nodes on layer 0 consist of at least a $1 - b - 1/\ln n$ fraction of uncorrupted players. So the lemma is true on layer 0.

Assume the lemma is true for layer i. Then a $1 - 4/\ln^2 n$ fraction of e-nodes are good, more specifically these e-nodes have at least a $1 - b - 4i/\ln n$ fraction of uncorrupted players that have a good path to their corresponding s-node. Since the election is legitimate by Lemmas B.2 and B.9, w.h.p., after ELECT-SUBCOMMITTEE at least a $1 - b - 4i/\ln n - 1/\ln n$ fraction of the players elected are uncorrupted and have a good path to any good parent of their s-node. Thus at least a $1 - b - (4i+1)/\ln n$ fraction of the players elected at layer i are uncorrupted and have good paths to good parent s-nodes on layer $i+1$. By Lemma B.12 this fraction is reduced by at most $32/\ln^2 n$. Thus at least a $1-b-(4i+2)/\ln n$ fraction of the players elected at layer i are uncorrupted 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 $1/\ln^{10} n$, by Corollary B.5 at least a $1 - 1/\ln^2 n - 1/\ln^{10} n$ fraction of the e-nodes (and their corresponding s-nodes) are good on layer $i+1$, and have at least a $1-b-(4i+2)/\ln n - 1/\ln n$ fraction of uncorrupted players that have good paths to their corresponding s-nodes. □

By Lemma B.13, w.h.p. the layer ℓ^* e-node is good. Thus the players in this e-node succeed in solving the semi-random-string agreement problem (step 4 of algorithm SRS-AGREEMENT). Since all the players are in the s-node (though they may appear multiple times) corresponding to A on ℓ^*, by Claim B.8 all but a $O(1/\ln n)$ fraction of the good players learn the final result. To prove the number of bits sent by each player is polylogarithmic we note each player is in a polylogarithmic number of e-nodes and s-nodes on each layer i, and participates in at most a polylogarithmic number of election on layer i. Since the number of layers is $O(\ln n)$ Theorem B.1 follows. Finally, the correctness of Theorem 2.1 follows from Theorem B.1 and the correctness of SRS-TO-QUORUM protocol.