Quorums Quicken Queries: Efficient Asynchronous Secure Multiparty Computation⋆

Varsha Dani[1], Valerie King[2], Mahnush Movahedi[1], and Jared Saia[1],⋆⋆

[1] University of New Mexico
[2] University of Victoria

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

Additionaly, we define the *threshold counting problem* and present a distributed algorithm to solve it in the asynchronous communication model. Our algorithm is load balanced, with computation, communication and latency complexity of $O(\log n)$, and may be of independent interest to other applications with a load balancing goal in mind.

1 Introduction

Recent years have seen a renaissance in secure multiparty computation (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 players 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?

In this paper, we take a first step towards solving MPC for large distributed systems. We describe algorithms that require each player to send a number of messages and perform an amount of computation that is $\tilde{O}(\frac{m}{n} + \sqrt{n})$, where n is the number of players and m is the number of gates in the circuit to be computed. This significantly improves over current algorithms, which require each player to both send a number of messages and perform computation that is $\Omega(nm)$. We now describe our model and problem.

⋆ Full version of this paper is available at `http://cs.unm.edu/~movahedi/mpc.pdf`.
⋆⋆ Partially supported by NSF CAREER Award 0644058 and NSF CCR-0313160.

Model: There are n players, with a private and authenticated channel between every pair of players. Communication is via asynchronous message passing, so that sent messages may be arbitrarily and adversarially delayed. Latency in this model is defined as the maximum length of any chain of messages (see [11,3]).

We assume a Byzantine adversary controls an unknown subset of up to t of the players. These players are *bad* (*i.e.* Byzantine) and the remaining players are *good*. The good players run our algorithm, but the bad players may deviate in an arbitrary manner. Our adversary is *static*, *i.e.* it must select the set of bad players at the start of our algorithm. The adversary is computationally unbounded. Thus, we make no cryptographic hardness assumptions.

MPC Problem: Each player, p_i, has a private input x_i. All players know a n-ary function f. We want to ensure that: 1) all players learn the value of f on the inputs; and 2) the inputs remain as private as possible: each player p_i learns nothing about the private inputs other than what is revealed by the output of f and the player's private value x_i.

In the asynchronous setting, the problem is challenging even with a trusted third party. In particular, the trusted party can not determine the difference between a message never being sent and a message being arbitrarily delayed, and so the t bad players can always refrain from sending any messages to the trusted party. Thus, the trusted party must wait to receive $n - t$ inputs. Then it must compute the function f using default values for the missing inputs, and send the output back to the players as well as the number of received inputs.[1] The goal of an asynchronous MPC protocol is to simulate the above scenario, without the trusted third party.

The function to be computed is presented as a circuit C with m gates. For convenience of presentation, we assume each gate has fan-in two and fan-out at most two. 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 also assume that all computations in the circuit occur over a finite field $\mathbb{F}$; The size of $\mathbb{F}$ depends on the specific function to be computed but must always be $\Omega(\log n)$. All the inputs, outputs and messages sent during the protocol are elements of $\mathbb{F}$, and consequently, messages will be of size $\log |\mathbb{F}|$.

Our MPC result requires solutions to the following two problems, which may be of independent interest.

Threshold Counting: There are n good players each with a bit initially set to 0. At least τ of the players will eventually set their bits to 1. The goal is for all the players to learn when the number of bits with values 1 is at least τ.

Quorum Formation: There are n players, up to t of whom may be bad. A *quorum* is a set of $c \log n$ players for some constant c. A quorum is called *good* if the fraction of bad players in it is at most $t/n + \delta$ for a fixed positive δ. We want all n players to agree on a set of n good quorums, and we want the quorums to be load-balanced: each player is mapped to $O(\log n)$ quorums.

---

[1] We send back only the number of inputs used, not the set of players whose inputs are used. This is required to ensure scalability; see Section 6.

1.1   Our Results

The main result of this paper is summarized by the following theorem.

Theorem 1. *Assume there are n players, less than a $\frac{1}{8} - \epsilon$ fraction of which are bad for some fixed $\epsilon > 0$, and an n-ary function, f that can be computed by a circuit of depth d with m gates. If all good players follow Algorithm 1, then with high probability (w.h.p.), they will solve MPC, while ensuring:*

1. *Each player sends at most $\tilde{O}(\frac{m}{n} + \sqrt{n})$ field elements,*
2. *Each player performs $\tilde{O}(\frac{m}{n} + \sqrt{n})$ computations, and*
3. *Expected total latency is $O(d \operatorname{polylog}(n))$.*

Our additional results are given by the following two theorems.

Theorem 2. *Assume n good players follow Algorithm τ-COUNTER. Then w.h.p., the algorithm solves the threshold counting problem, while ensuring:*

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

Theorem 3. *Assume n players, up to $t < (\frac{1}{4} - \epsilon)n$ of whom are bad, for fixed $\epsilon > 0$. If all good players follow the CREATE-QUORUM protocol, the following are ensured w.h.p.:*

1. *The players agree on n good quorums,*
2. *Each player sends at most $\tilde{O}(\sqrt{n})$ bits,*
3. *Each player performs $\tilde{O}(\sqrt{n})$ computations,*
4. *Total latency is $O(\operatorname{polylog}(n))$.*

In the rest of the paper we discuss the algorithms and ideas involved in obtaining these results. Detailed proofs are deferred to the full version [16].

2   Related Work

The study of secure computation started in 1982 with the seminal work of Yao [27]. Later Goldrich, Micali, and Wigderson [20] proposed the first generic scheme for solving a cryptographic notion of MPC. This work was followed by some unconditionally-secure schemes in late 1980s [6,10,26,5,21,22,4]. Unfortunately, these methods all have poor communication scalability that prevents their wide-spread use. In particular, if there are n players involved in the computation and the function f is represented by a circuit with m gates, then these algorithms require each player to send a number of messages and perform a number of computations that is $\Omega(mn)$ (see [18,19,17]).

Recent years have seen exciting improvements in the cost of MPC when m is much larger than n [12,14,13]. For example, the computation and communication cost for the algorithm described by Damgård *et al.* in [13] 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 MPC algorithms that are efficient in both n and m.

We first introduced the notion of using quorums for local communication to decrease the message cost in a brief announcement [15]. In that paper, we described a synchronous protocol with bit complexity of $\tilde{O}(\frac{m}{n} + \sqrt{n})$ per player that can tolerate a computationally unbounded adversary who controls up to $(\frac{1}{4} - \epsilon)$ fraction of the players for any fixed positive ϵ. This paper improves our previous result by handling asynchronous communication. One important challenge in the asynchronous communication model is to ensure that at least $n - t$ inputs are committed to, before the circuit evaluation. To address this issue we introduce and solve the *threshold counting problem.*

Boyle, Goldwasser, and Tessaro [8] describe a synchronous cryptographic protocol to solve MPC problem that is also based on quorums. Their algorithm uses a fully homomorphic encryption (FHE) scheme and thus, tolerates a computationally-bounded adversary that can take control of up to $(\frac{1}{3} - \epsilon)$ fraction of players for any fixed positive ϵ. Their protocol requires each player to send polylog(n) messages of size $\tilde{O}(n)$ bits and requires polylog(n) rounds. Interestingly the cost of the protocol is independent of the circuit size.

Counting Networks: Threshold counting can be solved in a load-balanced way using *counting networks*, which were first introduced by Aspnes, Herlihy, and Shavit [2]. 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, Herlihy, and Shavit [2] establish an $O(\log^2 n)$ upper bound on the depth complexity of counting networks. Since the latency of counting is dependent to the depth of the network, minimizing the network's depth is a goal for papers in this area. A simple explicit construction of an $O(\log n c^{\log^* n})$-depth counting network, and a randomized construction of an $O(\log n)$-depth counting networkwhich works with high probability is described in [24,25]. These constructions use the AKS sorting network [1] as a building block. While the AKS sorting network and the resulting counting networks have $O(\log n)$ depth, large hidden constants render them impractical. We note that the threshold counting problem is simpler than general counting.

3   Preliminaries

We say an event occurs *with high probability (w.h.p)*, if it occurs with probability at least $1 - 1/n^c$, for some $c > 0$ and sufficiently large n. We assume all computations occur over a finite field $\mathbb{F}$. Every time we use a mask during the protocol, we assume the mask is a value chosen uniformly at random from $\mathbb{F}$.

We now describe protocols that we use as building blocks in this paper.

Secret Sharing In secret sharing, a player, called the dealer, wants to distribute a secret amongst a group of participants, each of whom is allocated a share of the secret. The secret can be reconstructed only when a sufficient number of shares are combined together and each of the shares reveals nothing to the player possessing it. If a method is used to ensure the dealer sends shares of a real secret and not just some random numbers, then the new scheme is called Verifiable Secret Sharing (VSS). As our model is asynchronous, we use the asynchronous VSS (or AVSS) scheme described by Benor, Canneti and Goldreich in [5]. We denote the sharing phase by AVSS-SHARE and the reconstruction phase by AVSS-REC. The protocol of [5] works correctly even if up to $\frac{1}{4}$ of the players are bad. The latency of the protocols is $O(1)$ and the communication cost is poly(q), where q 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 q will be $O(\log n)$ and the communication cost per invocation will be polylog(n).

Heavy-Weight MPC: We use a heavy-weight asynchronous algorithm for MPC donated by HW-MPC. This algorithm, due to Ben-Or et al. [5], is an errorless MPC protocol that tolerates up to $\frac{1}{4}$ bad players. Let q be the number of players who run a HW-MPC to compute a circuit with $O(q)$ gates. The expected latency of HW-MPC is $O(q)$ and the number of messages sent poly(q). In this paper, we will use HW-MPC only for logarithmic number of players and gates, *i.e.*, $q = O(\log n)$ and the communication cost per invocation is polylog(n).[2]

Asynchronous Byzantine Agreement: In the Byzantine agreement problem, each player is initially given an input bit. All good players want to agree on a bit which coincides with at least one of their input bits. Every time a broadcast is required in our protocol, we use an asynchronous Byzantine agreement algorithm from [9], which we call ASYNCH-BA.

4   Technical Overview

We briefly sketch the ideas behind our three results.

Quorum-Based Gate Evaluation: The main idea for reducing the amount of communication required in evaluating the circuit is quorum-based gate evaluation. Unfortunately, if each player participates in the computation of the whole circuit, it must communicate with all other players. Instead, in quorum-based gate evaluation, each gate of the circuit is computed by a *gate gadget*. A gate gadget consists of three quorums: two *input quorums* and one *output quorum*. Input quorums are associated with the gate's children which serve inputs to the

---

[2] To make sure our algorithm has the expected total latency equal to $O(d \operatorname{polylog}(n))$, every time we need to run the HW-MPC algorithm, we run $O(\log n)$ same copy of it each for $O(\operatorname{polylog}(n))$ steps.
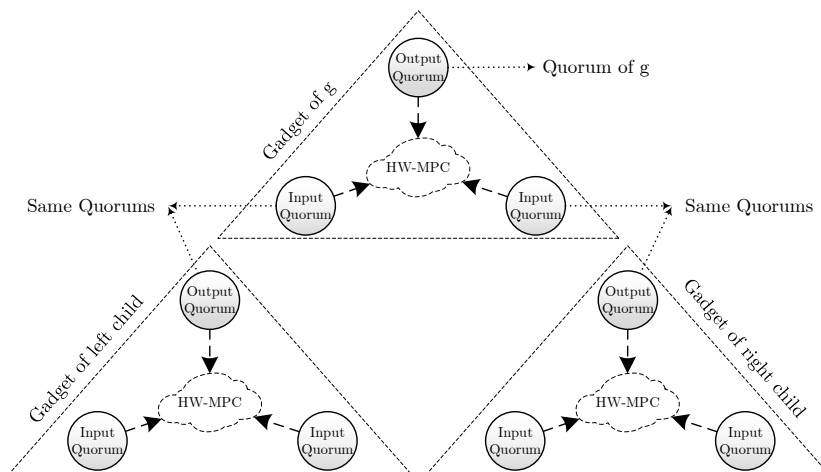
Fig. 1. The gate gadgets for gate g and its left and right children

gate. Output quorum is associated with the gate itself and is responsible to create a shared random mask and maintain 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 g, the output quorum of g's gadget is the input quorum of the gate gadget for all of g's parents (if any).

The players in each gate gadget run Hw-MPC among themselves to perform the gate operation. To make sure the computation is correct and secure, 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 number is not known to any individual player. 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 the masked version of the inputs and shares of the mask.

This gate gadget computation is continued in the same way for all gates of the circuit until the final output of whole circuit is evaluated. This technique for evaluating a gate of the circuit using quorums, is illustrated in Figure 5.2 and the details are described in Section 5.1.

Threshold Counting: Our interest in threshold counting for this paper is to ensure that at least $n-t$ inputs are committed to, before the circuit evaluation occurs. To solve the threshold counting problem, we design a new distributed data structure and algorithm called τ-Counter. The τ-Counter enables threshold counting with asynchronous communication, and may be of use for other problems beyond MPC.

To give intuition, we first consider a naive approach for counting in asynchronous model. Assume a complete binary tree where each player sends its

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 showing the number of inputs it has received so far and it sends the updated message every time this number changes. The problem with this approach is that it is not load-balanced: each node at depth i has $\frac{n}{2^i}$ descendants in the tree, and therefore, in the worst case, sends and receives $\frac{n}{2^i}$ messages. Thus, a child of the root sends $n/2$ messages to the root and receives the same number of messages from its children. To solve the load-balancing problem, we use a randomized approach which ensures w.h.p. 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. Subsequnt 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. The details of our algorithm are described in Section 5.1 and more formally in Algorithm 2. Theorem 2 describes the resource complexity of the algorithm.

Asynchronous Quorum Formation: Recently, King et al. [23] described an efficient algorithm to solve the quorum formation problem, w.h.p., in the synchronous model with full information. Our new algorithm, CREATE-QUORUM, builds on the result of [23] to solve the quorum formation problem in the asynchronous model, with private channels. The properties of CREATE-QUORUM, are described by Theorem 3. The algorithm and the proof are deferred to the full version [16] due to space restrictions.

5   Our Algorithm

Our algorithm makes use of a *circuit graph*, G, which is based on the circuit C that computes f. We assume the gates of the circuit C are numbered $1, 2, \ldots, m$, where the gate numbered 1 is the output gate. The circuit graph is a directed acyclic graph over $m + n$ nodes. There are n of these nodes, one per player, that we call *input nodes*. There are m remaining nodes, one per gate, that we call *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. Similar to our definition in C, 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. For each node in G, we define the following variables. Q_v is the quorum associated with node v. y_v is the output of the gate corresponding to v. Finally, r_v is a random mask and $\hat{y}_v$ is the masked output associated with node v, *i.e.* $\hat{y}_v = y_v + r_v$.

We number the nodes of G canonically in such a way that the input node numbered i corresponds to player p_i. We refer to the node corresponding to the output gate as the *output node*.

Algorithm 1 consists of four parts. The first part is to run CREATE-QUORUM in order to agree on n good quorums. The second part of the algorithm is

INPUT-COMMITMENT in which, quorums form the count tree. Then, each player p_i *commits* its input values to quorum i at the leaf nodes of the count tree and finally the players in that quorum decide whether these values are part of the computation or not. The details of this part of the algorithm is described in Section 5.1. The third part of the algorithm is evaluation of the circuit, described in detail in Section 5.2). Finally, the output from the circuit evaluation is sent back to all the players by quorums arranged in a complete binary tree.

Algorithm 1. Main Algorithm

1. All players run CREATE-QUORUM,
2. All players run INPUT-COMMITMENT,
3. All players run CIRCUIT-EVAL,
4. Propagate the output by quorums arranged in a complete binary tree.

5.1   Input-Commitment

In this section we describe a Monte Carlo algorithm, called τ-COUNTER that performs threshold counting for a threshold $\tau \geq n/2$.

The algorithm consists of up and down stages. For the up stage the players are arranged in a (pre-determined) tree data structure consisting 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. The players in the trees count the number of 1-inputs, *i.e.* the number of players' inputs that are set to **1**. As a result, the root can decide when the threshold is reached. In the down stage, the root notifies all the players of this event via a complete binary tree of depth $\log n$. Note that the trees used in the up and down stages have the same root. In what follows, unless otherwise specified, "tree" will refer to the tree data structure used for the up stage.

Let $D = \lceil \log \frac{\tau}{14 \log n} \rceil$. Note that $D = O(\log n)$. The root of our 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 $D + 1 - j$. Player 1 is assigned to the root and players 2 to $D + 1$, are assigned to its children, *i.e.* the roots of the collection subtrees, with player $j + 1$ being assigned to the jth child. The remaining nodes of the collection trees are assigned players 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 players 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*.

When a player's input is set to **1**, it sends a $\langle \mathsf{Flag} \rangle$ 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 close to evenly among the collection nodes. The parameters of the algorithm are set up so that w.h.p. each collection node receives at least $7 \log n$ $\langle \mathsf{Flag} \rangle$ messages.

Each collection node in the jth collection tree waits until it has received $7 \log n$ flags. It then sends its parent a $\langle\mathsf{Count}\rangle$ 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 w.h.p.

Each adding node waits until it has received a $\langle\mathsf{Count}\rangle$ message from each of its children. Then it sends a $\langle\mathsf{Count}\rangle$ message to its parent. We note that each adding node sends exactly one message during the algorithm. The parameters of the algorithm are arranged so that all the $\langle\mathsf{Count}\rangle$ messages that are sent in the the jth collection subtree together account for $\tau/2^j$ of the 1-inputs. Thus all the $\langle\mathsf{Count}\rangle$ 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 and upto $O(\log n)$ more are collected as flags at the root.

Algorithm 2. τ-COUNTER

n is the number of players, τ is the threshold. $D = \lceil\log(\frac{\tau}{14 \log n})\rceil$

1. Setup (no messages sent here):
   (a) Build the data structure:
      - Player 1 is the root.
      - For $1 \le j \le D$, player $j+1$ is a child of the root and the root of the jth collection subtree, which has depth $D + 1 - j$.
      - The remainder of the nodes in the collection subtrees are assigned to players left to right and top to bottom, starting with player $D + 2$.
   (b) Let $sum = 0$ for the root.
2. Up stage
   (a) Individual Players: upon input change to **1** choose a uniformly random collection node v from collection subtree 1 and send a $\langle\mathsf{Flag}\rangle$ to v.
   (b) Collection nodes in collection subtree j:
      - Upon receiving $7 \log n$ $\langle\mathsf{Flag}\rangle$s from collection subtree $j - 1$, if $j > 1$, or from individual players if $j = 1$, send parent a $\langle\mathsf{Count}\rangle$ message.
      - Upon subsequently receiving a $\langle\mathsf{Flag}\rangle$, send it to a uniformly random collection node in collection subtree $j + 1$, if $j < D$. If $j = D$ then send these directly to the root. Do this for up to $14 \log n$ flags. Then ignore all subsequent $\langle\mathsf{Flag}\rangle$ messagess.
   (c) Adding nodes: Upon receiving $\langle\mathsf{Count}\rangle$ messages from both children, send parent a $\langle\mathsf{Count}\rangle$ message.
   (d) Root: While $sum < \tau$
      - Upon receiving a $\langle\mathsf{Count}\rangle$ message from the root of collection subtree j, increment sum by $\tau/2^j$
      - Upon receiving a $\langle\mathsf{Flag}\rangle$ message, add one to sum.
3. Down stage (now $sum \ge \tau$)
   (a) Player 1 (the root): Send $\langle\mathsf{Done}\rangle$ to Players 2 and 3, then terminate.
   (b) Player j for $j > 1$: Upon receiving $\langle\mathsf{Done}\rangle$ from Player $\lfloor j/2 \rfloor$, forward it to Players $2j$ and $2j + 1$ (if they exist) and then terminate.

When player 1, at the root, has accounted for at least τ 1-inputs, it starts the down stage by sending the ⟨Done⟩ message to players 2 and 3. For $j > 1$, when player j receives the ⟨Done⟩ message, it forwards to players $2j$ and $2j+1$. Thus, eventually the ⟨Done⟩ message reaches all the players, who then know that the threshold has been met. The formal algorithm is shown in Algorithm 2.

The algorithm INPUT-COMMITMENT is based on τ-COUNTER assuming that the nodes in the data structure are assigned quorums. We assume that the quorums have a canonical numbering $Q_1, Q_2, \ldots Q_n$ and the role of "player" i in τ-COUNTER is played by quorum Q_i. 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.

Algorithm 3. INPUT-COMMITMENT

Run the following algorithms in parallel:

1. Algorithm IC-PLAYER.
2. Algorithm IC-INPUT.
3. Algorithm τ-COUNTER with $\tau = n - t$ and with quorums as participants.

Algorithm 4. IC-PLAYER

Run by player p_i with input x_i

1. $Q_v \leftarrow$ the quorum at the leaf node v associated with input of player p_i
2. Sample a uniformly random value from $\mathbb{F}$ and set r_v to this value.
3. $\hat{y}_v \leftarrow x_i + r_v$
4. Send $\hat{y}_v$ to all the players in Q_v
5. Run AVSS-SHARE to commit the secret value r_v to the players in Q_v.

Algorithm 5. IC-INPUT

Run by player p_j in Quorum Q_v associated with node v responsible for the input of p_i

1. After receiving $\hat{y}_v$ and a share of r_v, participate in the AVSS-SHARE verification protocol and agreement protocol to determine whether consistent shares for r_v and the same $\hat{y}_v$ are sent to everyone.
2. If the AVSS-SHARE verification protocol and the agreement protocol end and it is agreed that $\hat{y}_v$ was the same for all and shares of r_v are valid and consistent, set $b_{i,j} \leftarrow 1$ and you are ready to start the τ-COUNTER algorithm with ⟨Flag⟩ message.
3. Upon receiving ⟨Done⟩ from your parent quorum, participate in 5/8-MAJORITY using $b_{i,j}$ as your input. If it returns FALSE, reset $\hat{y}_v$ to the default value and your share of r_v to 0.

The τ-COUNTER is used for input commitment in the following way. Let v denote the input node associated with player p_i who holds input x_i. Quorum Q_i is assigned to this node. Player p_i samples r_v uniformly at random from $\mathbb{F}$,

sets $\hat{y}_v$ to $x_i + r_v$ and sends $\hat{y}_v$ to all players in Q_i. Next, player p_i uses AVSS-SHARE to commit to the secret value r_v to all players in Q_i. Once player p_i has verifiably completed this process for the values x_i and r_{v_i}, we say that player p_i has *committed* its masked value to Q_i, and each player p_j in Q_i then sets a bit $b_{i,j}$ to 1. If player p_i's shares *fail* the verification process, then the $b_{i,j}$'s are set to 0, which is also the default value. Note that the quorum Q_i's input for τ-COUNTER will be **1** if 5/8 of the $b_{i,j}$ are 1. The quorums acting as nodes in the τ-COUNTER data structure, run Algorithm 2 with threshold of $n - t$ to determine when at least $n - t$ inputs have been committed. Note that when a quorum has to select a random quorum to communicate with, they must agree on the quorum via a multiparty computation.

Based on the down stage in τ-COUNTER, when at least $n - t$ inputs have been detected at the root node, it sends a $\langle$Done$\rangle$ message to all the quorums via a complete binary tree. When a player p_j who is a member of quorum Q_i receives the $\langle$Done$\rangle$ message, p_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 set to 1. If they are, then the quorum determines that the i-th input (x_i) is part of the computation and uses the received value of $\hat{y}_v$ and shares of r_v as their input into GATE-EVAL. Otherwise, they set $\hat{y}_v$ to the default input and the shares of r_v to 0. We call this the 5/8-MAJORITY step.

5.2 Evaluating the Circuit

We assign nodes of G to quorums in the following way. The output node of G is assigned to quorum 1; then every node in G numbered i (other than the output node) is assigned to quorum number j, where $j = (i \mod n)$. Assume player p is in quorum Q_v at the leaf node v of count tree, which is the same quorum assigned to input node v in G. The circuit evaluation phase of the protocol for player p starts after the completion of the 5/8-MAJORITY step for node v in INPUT-COMMITMENT. After this step, for each input node v, players in Q_v know the masked input $\hat{y}_v$ and each has a share of the random element r_v, although the actual input and mask are unknown to any single player. The first step is to generate shares of uniformly random field elements for all gate nodes. If player p is in a quorum at gate node v, he generates shares of r_v, a uniformly random field element, by participating in the MASK-GENERATION algorithm. These shares are needed as inputs to the subsequent run of HW-MPC.

Next, players form the gadget for each gate node v with children $\mathsf{left}(v)$ and $\mathsf{right}(v)$ to evaluate the gate associated with v using GATE-EVAL as depicted in Figure 2. The values $y_{\mathsf{left}(v)}$ and $y_{\mathsf{right}(v)}$ are the inputs to the gate associated with v, and y_v is the output of v as it would be computed by a trusted party. First section of figure describes the initial conditions of the gate quorum and two input quorums before participating in HW-MPC. Each player in gate quorum Q_v has a share of the random element r_v (via MASK-GENERATION). Every player in the left input quorum $Q_{\mathsf{left}(v)}$ has the masked value $\hat{y}_{\mathsf{left}(v)} = y_{\mathsf{left}(v)} + r_{\mathsf{left}(v)}$ and a share of $r_{\mathsf{left}(v)}$ (resp. for the right input quorum). In the second section, all the players of the three quorums run HW-MPC, using their inputs, in order to
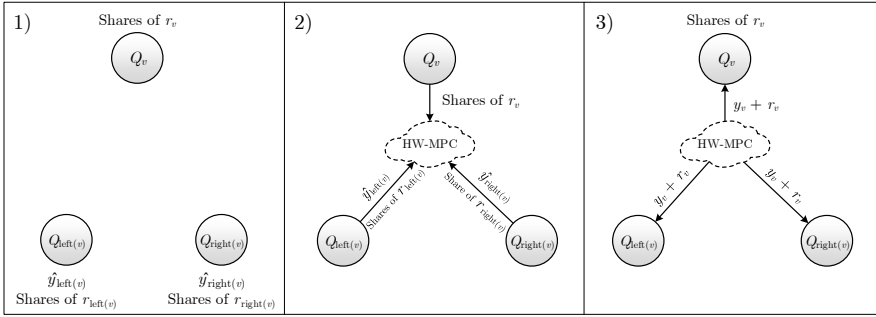
Fig. 2. Example Computation of a Gate associate with node v

Algorithm 6. CIRCUIT-EVAL

Run by each player p in a quorum associated with each node $v \in G$

1. For each input node v, after finishing INPUT-COMMITMENT with $\hat{y}_v$ and a share of r_v as output, p uses these as inputs to GATE-EVAL on each parent node of v.
2. For each gate node v:
   (a) p runs MASK-GENERATION on v and gets a share of r_v as output
   (b) p runs GATE-EVAL on v with its share of r_v as the input and gets $\hat{y}_v$ as output
   (c) p runs GATE-EVAL on each parent node of v with input $\hat{y}_v$ and p's share of r_v
3. After finishing computation of the gate represented by the output node, the players at the output node run OUTPUT-REC to reconstruct the output.

Algorithm 7. MASK-GENERATION

This protocol is run by each player p in a quorum associated with each gate node $v \in G$ to generate r_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. Run AVSS-SHARE to 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 dealt by yourself). This sum will be player p's share of the value r_v.

---

compute $\hat{y}_v$, which is equal to $y_v + r_v$. Third section shows the output of the gate evaluation after participating in Hw-MPC. Each player in Q_v now knows 1) the output of the gate plus the value of r_v; and 2) shares of r_v. Thus, players in Q_v now have the input to perform the computation associated with the parents of v (if any). Note that both y_v and r_v themselves are unknown to any individual.

The gate evaluation is performed for all gate nodes from the bottom of the G to the top. The output of the quorum associated with the output node in G is the output of the entire algorithm. Thus, this quorum will unmask the output via OUTPUT-REC. The last step of the algorithm is to send this output to all players. We do via a complete binary tree of quorums, rooted at the output quorum.

Algorithm 8. GATE-EVAL

This protocol is run for each gate node v with children $\mathsf{left}(v)$ and $\mathsf{right}(v)$, the participants are the players in Q_v, $Q_{\mathsf{left}(v)}$ and $Q_{\mathsf{right}(v)}$.

1. If you are a player in $Q_{\mathsf{left}(v)}$, (resp. $Q_{\mathsf{right}(v)}$) use $(\hat{y}_{\mathsf{left}(v)}$, share of $r_{\mathsf{left}(v)})$ (resp. $(\hat{y}_{\mathsf{right}(v)}$, share of $r_{\mathsf{right}(v)})$) as your input to HW-MPC. If you are a player in Q_v, use share of r_v as your input to HW-MPC.
2. Participate in HW-MPC.
3. $\hat{y}_v \leftarrow$ value returned by HW-MPC.

Algorithm 9. OUTPUT-REC

This protocol is run by all players in $Q_{\text{output-node}}$.

1. Reconstruct $r_{\text{output-node}}$ from its shares using AVSS-REC.
2. Set the circuit output message: $\langle O \rangle \leftarrow \hat{y}_{\text{output-node}} - r_{\text{output-node}}$.
3. Send $\langle O \rangle$ to all players in the quorums numbered 2 and 3.

---

6   Some Remarks

As described in the introduction, the goal of MPC is to simulate a trusted third party in computation of the circuit and then send back the computation result to the players. Let S denote the set of players from whom input is received by the (simulated) trusted party. Recall that $|S| \geq n - t$.[3] Thus, for an arbitrary S a description of S requires $\Omega(n)$ bits, and cannot be sent back to the players using only a scalable amount of communication. Therefore, 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.

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 an addition circuit to add these bits together and then disperse the result. This is an MPC, all of whose inputs are held by good players, since each input bit b_i is jointly held

---

[3] 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.

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.

In the protocol proposed in this paper, it may be the case that a player p participates more than one time in the quorums performing a single instance of Hw-MPC. In such a case, we allow p to play the role of more than one different players in the 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.

7   Conclusion

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

Many problems remain open 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 [7]. 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. Ajtai, M., Komlós, J., Szemerédi, E.: An 0(n log n) sorting network. In: Proceedings of STOC 1983, pp. 1–9. ACM, New York (1983)
2. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks and multi-processor coordination. In: Proceedings of STOC 1991, pp. 348–358. ACM (1991)
3. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn., p. 14. John Wiley Interscience (March 2004)
4. Beerliová-Trubíniová, Z., Hirt, M.: Efficient multi-party computation with dispute control. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 305–328. Springer, Heidelberg (2006)
5. Ben-Or, M., Canetti, R., Goldreich, O.: Asynchronous secure computation. In: Proceedings of STOC 1993 (1993)
6. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for noncryptographic fault-tolerant distributed computing. In: Proceedings of STOC 1988, pp. 1–10 (1988)

7. Bogetoft, P., et al.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009)
8. Boyle, E., Goldwasser, S., Tessaro, S.: Communication locality in secure multiparty computation: how to run sublinear algorithms in a distributed setting. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 356–376. Springer, Heidelberg (2013)
9. Canetti, R., Rabin, T.: Fast asynchronous byzantine agreement with optimal resilience. In: Proceeding of STOC 1993, pp. 42–51 (1993)
10. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: Proceedings of STOC 1988, pp. 11–19 (1988)
11. Chor, B., Dwork, C.: Randomization in Byzantine agreement. Advances in Computing Research 5, 443–498 (1989)
12. Damgård, I., Ishai, Y.: Scalable secure multiparty computation. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 501–520. Springer, Heidelberg (2006)
13. Damgård, I., Ishai, Y., Krøigaard, M., Nielsen, J.B., Smith, A.: Scalable multiparty computation with nearly optimal work and resilience. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 241–261. Springer, Heidelberg (2008)
14. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg (2007)
15. Dani, V., King, V., Movahedi, M., Saia, J.: Breaking the o(nm) bit barrier: Secure multiparty computation with a static adversary. In: Proceedings of PODC 2012 (2012)
16. Dani, V., King, V., Movahedi, M., Saia, J.: Quorums quicken queries: Efficient asynchronous secure multiparty computation. Manuscript (2013)
17. Du, W., Atallah, M.: Secure multi-party computation problems and their applications: a review and open problems. In: Proceedings of the 2001 Workshop on New Security Paradigms, pp. 13–22. ACM (2001)
18. Frikken, K.: Secure multiparty computation. In: Algorithms and Theory of Computation Handbook, p. 14. Chapman & Hall/CRC (2010)
19. Goldreich, O.: Secure multi-party computation. Manuscript (1998)
20. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of STOC 1987, pp. 218–229. ACM (1987)
21. Hirt, M., Maurer, U.: Robustness for free in unconditional multi-party computation. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 101–118. Springer, Heidelberg (2001)
22. Hirt, M., Nielsen, J.B.: Upper bounds on the communication complexity of optimally resilient cryptographic multiparty computation. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 79–99. Springer, Heidelberg (2005)
23. King, V., Lonargan, S., Saia, J., Trehan, A.: Load balanced scalable byzantine agreement through quorum building, with full information. In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 203–214. Springer, Heidelberg (2011)
24. Klugerman, M., Plaxton, C.G.: Small-depth counting networks. In: Proceedings of STOC 1992, pp. 417–428 (1992)
25. Klugerman, M.R.: Small-depth counting networks and related topics (1994)
26. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: Proceedings of STOC 1989, pp. 73–85. ACM (1989)
27. Yao, A.: Protocols for secure computations. In: Proceedings of FOCS 1982, pp. 160–164 (1982)