Self-Healing Computation*

George Saad and Jared Saia

Department of Computer Science, University of New Mexico,
{saad,saia}@cs.unm.edu

Abstract. In the problem of reliable multiparty computation (RC), there are n parties, each with an individual input, and the parties want to jointly compute a function f over n inputs. The problem is complicated by the fact that an omniscient adversary controls a hidden fraction of the parties.
We describe a self-healing algorithm for this problem. In particular, for a fixed function f, with n parties and m gates, we describe how to perform RC repeatedly as the inputs to f change. Our algorithm maintains the following properties, even when an adversary controls up to $t \leq (\frac{1}{4} - \epsilon)n$ parties, for any constant $\epsilon > 0$. First, our algorithm performs each reliable computation with the following amortized resource costs: $O(m + n \log n)$ messages, $O(m + n \log n)$ computational operations, and $O(\ell)$ latency, where ℓ is the depth of the circuit that computes f. Second, the expected total number of corruptions is $O(t(\log^* m)^2)$, after which the adversarially controlled parties are effectively quarantined so that they cause no more corruptions.

Keywords: Self-Healing Algorithms, Threshold Cryptography, Leader Election

1 Introduction

How can we protect a network against adversarial attack? A traditional approach provides robustness through redundant components. If one component is attacked, the remaining components maintain functionality. Unfortunately, this approach incurs significant resource cost, even when the network is not under attack.

An alternative approach is self-healing, where a network detects the damage made by attacks, inspects the corruption situation and automatically recovers. Self-healing algorithms expend additional resources only when it is necessary to repair from attacks.

In this paper, we describe self-healing algorithms for the problem of *reliable multiparty computation (RC)*. In the RC problem, there are n parties, each with an individual input, and the parties want to jointly compute a function f over n inputs. A hidden $1/4$-fraction of the parties are controlled by an omniscient Byzantine adversary. A party that is controlled by the adversary is said to be *bad*, and the remaining parties are said to be *good*. Our goal is to ensure that all good parties learn the output of f. [1]

RC abstracts many problems that may occur in high-performance computing, sensor networks, and peer-to-peer networks. For example, we can use RC to enable performance profiling and system monitoring, compute order statistics, and enable public voting.

* This research is partially supported by NSF grants: CISE-1117985 and CNS-1017509.

[1] Note that RC differs from secure multiparty computation (MPC) only in that there is no requirement to keep inputs private.

Our main result is an algorithm for RC that 1) is asymptotically optimal in terms of total messages and total computational operations; and 2) limits the expected total number of corruptions. Ideally, each bad party would cause $O(1)$ corruptions; in our algorithm, each bad party causes an expected $O((\log^* m)^2)$ corruptions.

1.1 Our Model

We assume a *static* Byzantine adversary that takes over $t \leq (\frac{1}{4} - \epsilon)n$ parties before the algorithm begins, for any constant $\epsilon > 0$. As mentioned previously, parties that are compromised by the adversary are called *bad*, and the remaining parties are *good*. The bad parties may arbitrarily deviate from the protocol, by sending no messages, excessive numbers of messages, incorrect messages, or any combination of these. The good parties follow the protocol. We assume that the adversary knows our protocol, but is unaware of the random bits of the good nodes. We make use of a public key cryptography scheme, and thus assume that the adversary is computationally bounded.

We assume a partially synchronous communication model. Any message sent from one good node to another good node requires at most h time steps to be sent and received, and the value h is known to all nodes. However, we allow the adversary to be *rushing*: the bad nodes receive all messages from good nodes in a round before sending out their own messages. We further assume that each party has a unique ID. We say that party p has a link to party q if p knows q's ID and can thus directly communicate with node q.

In the reliable multiparty computation problem, we assume that the function f can be implemented with an arithmetic circuit over m gates, where each gate has two inputs and at most two outputs.[2] For simplicity of presentation, we focus on computing a single function multiple times (with changing inputs). However, we can also compute multiple functions with our algorithm.

1.2 Our Result

We describe an algorithm, *COMPUTE*, to efficiently solve reliable multiparty computation. Our main result is summarized in the following theorem.

Theorem 1. *Assume we have n parties providing inputs to a function f that can be computed by an arithmetic circuit with depth ℓ and containing m gates. Then COMPUTE solves RC and has the following properties.*

(1) In an amortized sense[3], any execution of COMPUTE requires:
- *$O(m + n \log n)$ messages sent by all parties;*
- *$O(m + n \log n)$ computational operations performed by all parties; and*
- *$O(\ell)$ latency.*

(2) The expected total number of times COMPUTE returns a corrupted output is $O(t(\log^* m)^2)$.

[2] We note that any gate of any fixed in-degree and out-degree can be converted into a fixed number of gates with in-degree 2 and out-degree at most 2.

[3] In particular, if we call *COMPUTE* $\mathcal{L}$ times, then the expected total number of messages sent will be $O(\mathcal{L}(m+n \log n)+t(m \log^2 n))$. Since t is fixed, for large $\mathcal{L}$, the expected number of messages per *COMPUTE* is $O(m+n \log n)$. Similar for the cost of computational operations.

1.3 Technical Overview

Our algorithms make critical use of quorums and a quorum graph.

Quorums and the Quorum Graph: We define a quorum to be a set of $\Theta(\log n)$ parties, of which at most $1/4$-fraction are bad. Many results show how to create and maintain a network of quorums [1,2,3,4,5,6,7]. All of these results maintain what we will call a *quorum graph* in which each vertex represents a quorum. The properties of the quorum graph are:

(1) each party is in $\Theta(\log n)$ quorums;
(2) for any quorum Q, any party in Q can communicate directly to any other party in Q; and
(3) for any quorums Q and Q' that are connected in the quorum graph, any party in Q can communicate directly with any party in Q' and vice versa.

Moreover, we assume that for any two parties x and y in a quorum, x knows all quorums that y is in.

Computing with Quorums: We maintain a quorum graph with $m + n$ nodes: m nodes for the gates of the circuit and n nodes for the inputs of the parties. The input nodes are connected to the gates using these inputs, and the gate nodes are connected as in the circuit. Quorums are mapped to nodes in this quorum graph as described above. For simplicity of presentation, we let the computation be performed from the left to the right, where the input quorums are leftmost and the output quorum is rightmost.

Naive Algorithm: A correct but inefficient way to solve RC is as follows. Each party s_i sends its input to all parties of the appropriate input quorum. Then the computation is performed from left to right. All parties in each quorum compute the appropriate gate operation on their inputs, and send their outputs to all parties in the right neighboring quorums via all-to-all communication. At the next level, all parties in each quorum take the majority of the received messages in order to determine the correct input for their gate. At the end, the parties in the rightmost quorum will compute the correct output of the circuit. They then forward this output back from right to left through the quorum graph using the same all-to-all communication and majority filtering.

Unfortunately, this naive algorithm requires $O(m \log^2 n)$ messages and $O(m \log n)$ computational operations. Our main goal is to remove the logarithmic factors. [4]

Our Approach: A more efficient approach is for each quorum to have a leader, and for this leader to receive inputs, perform gate computations, and send off the output. Unfortunately, a single bad leader can corrupt the entire computation.

To address this issue, we provide *CHECK* (Section 2.3). This algorithm determines if there has been a corruption, and if so, it calls *UPDATE* (Section 2.4), which identifies at least one pair of parties that are in *conflict*. Informally, we say that a pair of parties are in conflict if they each accuse the other of malicious behavior. In such a situation, we know that at least one party in the pair is bad. Our approach is to mark both parties

[4] We note that such asymptotic improvements can be significant for large networks. For example, if $n = 64{,}000$, then we would expect our algorithm to reduce message costs by a factor of $\log^2 n = 255$.

in each conflicting pair, and these marked parties are prohibited from participating in future computation but they still can provide the inputs of the circuit. [5]

The basic idea of *CHECK* is to redo the computation through subsets of parties so that one subset for each gate. *CHECK* runs in multiple rounds. Initially, all subsets are empty; and in each round, a new party is selected uniformly at random from each quorum to be added to each subset. We call these parties the *checkers*. For convenience of presentation, we will refer to the leaders as the checkers for round 0. For each round $i \geq 1$, all i checkers at gate g: 1) receive inputs to g from the checkers at each input gate for g; 2) compute the gate output for g based on these inputs; and 3) send this output to the checkers at each output gate for g. If a good checker ever receives inconsistent inputs, it calls *UPDATE*. Unfortunately, waiting until a round where each gate has had at least one good checker would require $O(\log n)$ rounds.

To do better, we use the following approach. Let G be the quorum graph as defined above and let the checkers be selected as above. Call a subgraph of G bad in a given round if all checkers in the nodes of that subgraph are bad; note that such a subgraph consists of the new checkers that are added to the subsets in that round. When the adversary corrupts an output of a bad subgraph of G in one round, it has to keep corrupting this output by nesting levels of bad subgraphs of G in all subsequent rounds.

Recall that in each round, new checkers are selected uniformly at random. When *CHECK* selects a good checker at a quorum, it is as removing the node associated with this quorum from the quorum graph. Thus, we can view *CHECK* as repeatedly removing nodes from increasingly smaller subgraphs of G until no nodes remain, at which the corruption is detected. A key lemma (Lemma 2) shows that for any rooted directed acyclic graph (DAG), with m nodes and maximum indegree 2, when each node is deleted independently with probability at least $1/2 + \epsilon$, for any constant $\epsilon > 0$, the probability of having a connected DAG, rooted at one node, with surviving nodes of size $\Omega(\log m)$, is at most $1/2$. By this lemma, we show that *CHECK* requires only $O(\log^* m)$ rounds to detect a corruption with constant probability.[6]

CHECK requires $O((m + n \log n)(\log^* m)^2)$ messages. Then, we can call it with probability $1/(\log^* m)^2$ and obtain asymptotically optimal resource costs for the RC problem, while incurring an expected $O(t(\log^* m)^2)$ corruptions.

1.4 Related Work

Our results are inspired by recent work on self-healing algorithms. Early work of [8,9,10,11,12] discusses different restoration mechanisms to preserve network performance by adding capacity and rerouting traffic streams in the presence of node or link failures. This work presents mathematical models to determine global optimal restoration paths, and provides methods for capacity optimization of path-restorable networks.

More recent work [13,14,15,16,17,18] considers models where the following process repeats indefinitely: an adversary deletes some nodes in the network, and the al-

[5] A technical point is that we may need to unmark all parties in a quorum if too many parties in that quorum become marked. However, a potential function argument (Lemma 8) shows that after $O(t)$ markings, all bad parties will be marked.

[6] This probability can be made arbitrarily close to 1 by adjusting the hidden constant in the $O(\log^* m)$ rounds.

gorithm adds edges. The algorithm is constrained to never increase the degree of any node by more than a logarithmic factor from its original degree. In this model, researchers have presented algorithms that ensure the following properties: the network stays connected and the diameter does not increase by much [13,14,15]; the shortest path between any pair of nodes does not increase by much [16]; expansion properties of the network are approximately preserved [17]; and keeping network backbones densely connected [18].

This paper particularly builds on [19]. That paper describes self-healing algorithms that provide reliable communication, with a minimum of corruptions, even when a Byzantine adversary can take over a constant fraction of the nodes in a network. While our attack model is similar to [19], reliable *computation* is more challenging than reliable communication, and hence this paper requires a significantly different technical approach. Additionally, we improve the fraction of bad parties that can be tolerated from $1/8$ to $1/4$.

Reliable multiparty computation (RC) is closely related to the problem of secure multiparty computation (MPC) which has been studied extensively for several decades (see e.g. [20,21,22,23] or the recent book [24]). RC is simpler than MPC in that it does not require inputs of the parties to remain private. Our algorithm for RC is significantly more efficient than current algorithms for MPC, which require at least polylogarithmic blowup in communication and computational costs in order to tolerate a Byzantine adversary. We reduce these costs through our self-healing approach, which expends additional resources only when corruptions occur, and is able to "quarantine" bad parties after $O(t(\log^* m)^2)$ corruptions.

1.5 Organization of Paper

The rest of this paper is organized as follows. In Section 2, we describe our algorithms. The analysis of our algorithms is shown in Section 3. Finally, we conclude and describe problems for future work in Section 4.

2 Our Algorithms

In this section, we describe our algorithms: *COMPUTE*, *COMPUTE-CIRCUIT*, *CHECK* and *UPDATE*.

Our algorithms aim at detecting corruptions and marking the bad parties. Parties that are marked are not allowed to participate in the computation. Initially, all parties are unmarked.

Recall that there are n parties, each provides an input to an input quorum, Q_i, for $1 \leq i \leq n$; and then the computation is performed through m quorums, Q_j's, for $n+1 \leq j \leq m+n$. The result is produced at an output quorum Q_{m+n}, and it is sent back to the senders through the m quorums.

Before discussing our main *COMPUTE* algorithm, we describe that when a party x broadcasts a message msg, signed by a quorum Q, to a set of parties S, it calls $BROADCAST(msg, Q, S)$.

2.1 *BROADCAST*

In *BROADCAST* (Algorithm 1), we use threshold cryptography to avoid the overhead of Byzantine Agreement. In a (η, η')-threshold cryptographic scheme, a private key is

distributed among η parties in such a way that 1) any subset of more than η' parties can jointly reassemble the key; and 2) no subset of at most η' parties can recover the key. The private key can be distributed using a *Distributed Key Generation* (DKG) protocol [25]. DKG generates the public/private key shares of all parties in every quorum. The public key of each quorum is known to all parties in the quorum, and to all parties in all neighboring quorums in the circuit.

In particular, we use a $(|Q|, \frac{3|Q|}{4} - 1)$-threshold scheme, where $|Q|$ is the quorum size. A party x calls $BROADCAST$ in order to send a message msg to all parties in S after signing it by the private key of quorum Q. We sign the message using Algorithm 2 to fulfill that: 1) at least $3/4$-fraction of the parties in quorum Q have received the same message msg; 2) they agree upon the content of msg; and 3) they give a permission to party x to broadcast this message.

Algorithm 1 BROADCAST(msg, Q, S) ▷ A party x sends message msg to a set of parties S, after signing it by the private key of quorum Q.

1: Party x calls *SIGN* (msg, Q). ▷ signs msg by the private key of quorum Q.
2: Party x sends this signed-message to all parties in S.

Any call to *BROADCAST* has $O(\log n + |S|)$ messages and $O(\log n)$ computational operations for signing the message msg by $O(\log n)$ parties in Q, with latency $O(1)$.

Algorithm 2 SIGN(msg, Q) ▷ Signs message msg by the private key of quorum Q.

1: Party x sends message msg to all parties in Q.
2: Each party in Q signs msg by its private key share to obtain its message share.
3: Each party in Q sends its message share back to party x.
4: Party x interpolates at least $\frac{3|Q|}{4}$ message shares to obtain a signed-message of Q.

2.2 *COMPUTE*

Now we describe our main algorithm, *COMPUTE* (Algorithm 3), which calls *COMPUTE-CIRCUIT* (Algorithm 4). In *COMPUTE-CIRCUIT*, the n parties broadcast their inputs to the input quorums. The input quorums forward these inputs to a circuit of m leaders in order to perform the computation and provide the result to the output quorum. Then this result is sent back to all senders through the same circuit. Recall that a leader of a quorum, is a party in this quorum, that is: 1) a representative of all parties in the quorum; and 2) it is known to all parties in the quorum and neighboring quorums. We assume that all parties provide their inputs to the circuit in the same round.

In the presence of an adversary, *COMPUTE-CIRCUIT* is vulnerable to corruptions. Thus, *COMPUTE* calls *TRIGGER-CHECK* (Algorithm 5), in which the parties of the output quorum decide together, to trigger *CHECK* (Algorithm 7) with probability $1/(\log^* m)^2$, using secure multiparty computation (MPC) [23]. *CHECK* is triggered in

Algorithm 3 COMPUTE ▷ performs a reliable computation and sends the result reliably to all parties.

1: COMPUTE-CIRCUIT ▷ computes and sends back the result through a circuit of leaders.
2: TRIGGER-CHECK ▷ The output quorum triggers *CHECK* with probability $1/(\log^* m)^2$.

order to detect if a computation was corrupted in the last call to *COMPUTE-CIRCUIT*, with probability at least $1/2$.

Unfortunately, while *CHECK* can determine if a corruption occurred, it does not specify the location where the corruption occurred. Thus, when *CHECK* detects a corruption, *UPDATE* (Algorithm 11) is called. In each call to *UPDATE*, two neighboring quorums in the circuit are identified such that at least one pair of parties in these quorums is in conflict and at least one party in this pair is bad. Then the parties that are in conflict are marked in all quorums they are in and in their neighboring quorums. Moreover, for each pair of leaders that are in conflict, their quorums elect a new pair of unmarked leaders uniformly at random. If $(1/2 - \gamma)$-fraction of parties in any quorum have been marked, for any constant $\gamma > 0$, e.g., $\gamma = 0.01$, they are set unmarked in all their quorums and the neighboring quorums.

Moreover, we use *BROADCAST* in *COMPUTE-CIRCUIT* and *CHECK* in order to handle any accusation issued in *UPDATE* against the parties that provide the inputs to the input quorums, or those that receive the result in the output quorum.

Our model does not directly consider concurrency. In a real system, concurrent executions of *COMPUTE* that overlap at a single quorum may allow the adversary to achieve multiple corruptions at the cost of a single marked bad party. However, this does not effect correctness, and, in practice, this issue can be avoided by serializing concurrent executions of *COMPUTE*. For simplicity of presentation, we leave the concurrency aspect out of this paper.

2.3 *CHECK*

In this section, we describe *CHECK* algorithm, which is stated formally as Algorithm 7. In this algorithm, we make use of subquorums, where a subquorum is a subset of unmarked parties in a quorum. Let U_k be the set of all unmarked parties in quorum Q_k, for $1 \leq k \leq m + n$.

CHECK runs for $O(\log^* m)$ rounds. For each round i, the parties of the output quorum Q_{m+n} elect an unmarked party $\mathbf{r}$ from Q_{m+n} to be in charge of the recomputation in round i. This election process is stated formally in *ELECT* (Algorithm 6). The elected party $\mathbf{r}$ calls *REQUEST* (Algorithm 8) to send a request through a DAG of subquorums, S_j^A's, to the n senders to recompute. Then the recomputation is performed by *RECOMPUTE* (Algorithm 9), in which each sender that receives such request provides its input to redo the computation through a DAG of subquorums, S_j^B's. Finally, when $\mathbf{r}$ receives the result of the computation, it calls *RESEND-RESULT* (Algorithm 10) in order to send back this result to the senders through a DAG of subquorums S_j^C's, for $n + 1 \leq j \leq m + n$.

Moreover, in *ELECT* (Q), the parties of quorum Q use MPC to elect an unmarked party uniformly at random from Q. Note that at any moment at least half of the un-

Algorithm 4 COMPUTE-CIRCUIT ▷ performs a computation through a circuit of leaders producing a result at the output quorum; then the result is sent back through same circuit to all senders.

for $i = 1, \ldots, n$ **do** ▷ provides the inputs to the circuit.
  Party s_i calls *BROADCAST* (a_i, Q_i, Q_i). ▷ s_i broadcasts its input a_i to all nodes in Q_i.
  All parties in Q_i send a_i to the leaders of the right neighboring quorums of Q_i.
end for
for $i = n+1, \ldots, m+n-1$ **do** ▷ performs the computation.
  for all $j : i < j \leq m+n$ and $(Q_i, Q_j) \in Circuit$ **do**
    if leader $q_i \in Q_i$ receives all its inputs **then**
      q_i performs an operation on its inputs producing an output, b_i.
      q_i sends b_i to leader $q_j \in Q_j$.
    end if
  end for
end for
if leader $q_{m+n} \in Q_{m+n}$ receives all its inputs **then**
  q_{m+n} performs an operation on its inputs producing an output, b_{m+n}.
  q_{m+n} broadcasts b_{m+n} to all parties in Q_{m+n}.
end if
for $i = m+n, \ldots, n+2$ **do** ▷ sends back the result to the leftmost leaders.
  for all $j : n+1 \leq j < i$ and $(Q_j, Q_i) \in Circuit$ **do**
    Leader $q_i \in Q_i$ sends b_{m+n} to leader $q_j \in Q_j$.
  end for
end for
for $i = 1, \ldots, n$ **do** ▷ sends result to all parties after broadcasting it to the input quorums.
  The leaders of Q_i's right neighboring quorums call $BROADCAST(b_{m+n}, Q_i, Q_i)$.
  All parties in Q_i send b_{m+n} to sender s_i.
end for

Algorithm 5 TRIGGER-CHECK ▷ The parties of the output quorum Q_{m+n} trigger *CHECK* with probability $1/(\log^* m)^2$.

Each party in Q_{m+n} chooses an input: a real number uniformly distributed between 0 and 1.
The parties of Q_{m+n} perform MPC to find the output, $prob$, which is the sum of all their inputs modulo 1.
if $prob \leq 1/(\log^* m)^2$ **then**
  CHECK
end if

Algorithm 6 ELECT(Q) ▷ Parties in Q elect an unmarked party in Q using MPC.

Let each party in the set of unmarked parties, $U \subset Q$, is assigned a unique integer from 0 to $|U| - 1$.
Each party in Q chooses an input: an integer uniformly distributed between 0 and $|U| - 1$.
The parties of Q perform MPC to find the output: the sum of all their inputs modulo $|U|$.
The party in U associated with this output number is the elected party.

Algorithm 7 CHECK ▷ Party $\mathbf{r}$ calls *CHECK* to check for corruptions.

Declaration: Let U_k be the set of all unmarked parties in quorum Q_k, for $1 \leq k \leq m+n$. Also let m' be the maximum number of nodes in any quorum. Further, let subquorums, S_j^A, S_j^B and S_j^C, be initially empty, for all $n+1 \leq j \leq m+n$.

1: **for** $i \leftarrow 1, \ldots, 8(\log^* m + 2(\log c + 1))$* **do**
2: ELECT(Q_{m+n}) ▷ elects an unmarked party $\mathbf{r} \in Q_{m+n}$.
3: Party $\mathbf{r}$ constructs A^i, B^i and C^i to be three, m by m', arrays of random integers.**
4: REQUEST(i, A^i, B^i) ▷ $\mathbf{r}$ requests all senders to recompute.
5: RECOMPUTE ▷ recomputes producing the result, b_{m+n}^i, at $\mathbf{r}$.
6: RESEND-RESULT(i, C^i, b_{m+n}^i) ▷ $\mathbf{r}$ sends back b_{m+n}^i to all parties.
7: **end for**

* $c = \frac{2(1+2p)}{\log e(1-2p)^2}$; for any quorum, $p \leq 1/2 - \epsilon$ is the probability of selecting a bad party u.a.r. from the unmarked nodes in this quorum, for a constant $\epsilon > 0$.
** $A^i[k,k']$, $B^i[k,k']$ and $C^i[k,k']$ are uniformly random integers between 1 and k', for $1 \leq k \leq m$ and $1 \leq k' \leq m'$.
Note that: if a party has previously received k_p, then it verifies each subsequent message with it; also if a node receives inconsistent messages or fails to receive and verify an expected message, then it initiates a call to *UPDATE*.

Algorithm 8 REQUEST(i, A^i, B^i) ▷ $\mathbf{r}$ requests n senders through a DAG of subquoums, S_j^A's, for $n+1 \leq j \leq m+n$, to redo the computation.

1: Party $\mathbf{r}$ calls $SIGN([i, A^i, B^i, \mathbf{r}], Q_{m+n})$. ▷ signs $[i, A^i, B^i, \mathbf{r}]$ by Q_{m+n}'s private key.
2: Party $\mathbf{r}$ sets $REQ^i = ([i, A^i, B^i, \mathbf{r}]_{k_s}, k_p)$. ▷ (k_p, k_s) : public/private key pair of Q_{m+n}.
3: Party $\mathbf{r}$ sends REQ^i to all parties of quorum Q_{m+n}.
4: All parties in Q_{m+n} calculate party, $q_{m+n}^i \in U_{m+n}$, of index A_{m+n}^i to be added to S_{m+n}^A.*
5: **for** $j \leftarrow m+n, \ldots, n+1$ **do** ▷ sends REQ^i through a DAG of subquorums.
6: Let $Q_{j'}$ and $Q_{j''}$ be the left neighboring quorums of Q_j in the circuit, for $n+1 \leq j', j'' \leq m+n$. **
7: All i parties in S_j^A calculate parties, $q_{j'}^i$ and $q_{j''}^i$, of indices $A_{j'}^i$ and $A_{j''}^i$, to be added to $S_{j'}^A$ and $S_{j''}^A$ respectively.
8: Party q_j^i calculate all parties in $S_{j'}^A$ and $S_{j''}^A$ using $A_{j'}^1, \ldots, A_{j'}^i$ and $A_{j''}^1, \ldots, A_{j''}^i$.
9: **for** $k \leftarrow 1, \ldots, i$ **do** ▷ k refers to the rounds prior to round i.
10: Party q_j^k sends REQ^k to parties $q_{j'}^i$ and $q_{j''}^i$.
11: Party q_j^i sends REQ^i to parties $q_{j'}^k$ and $q_{j''}^k$.
12: **end for**
13: **end for**
14: **for** $k \leftarrow n, \ldots, 1$ **do** ▷ The input quorums forward REQ^i to all senders.
15: Let $Q_{k'}$ and $Q_{k''}$ be the right neighboring quorums of Q_k in the circuit.
16: All i parties in $S_{k'}$ and all parties in $S_{k''}$ call $BROADCAST(REQ^i, Q_k, Q_k)$.
17: All parties in Q_k send REQ^i to sender s_k.
18: **end for**

* $A_j^i = A^i[j-n, |U_j|]$ is the index of the party, q_j^i, which is selected u.a.r. from the parties in U_j in round i of *REQUEST*; note that the nodes of U_j are sorted by their IDs, for $n+1 \leq j \leq m+n$.
** Recall that there are no subquorums for the input quorums.

marked parties in Q are good. Thus, the elected party is good with probability at least $1/2$. Finally, this election protocol runs in $O(1)$ time, and requires $O(\log^4 n)$ messages and $O(\log^4 n)$ computational operations.

During *CHECK*, if any party receives inconsistent messages or fails to receive and verify any expected message in any round, it initiates a call to *UPDATE*.

Algorithm 9 RECOMPUTE ▷ n senders provide inputs to a DAG of subquorums, S_j^B's, for $n+1 \leq j \leq m+n$, to recompute, producing a result, b_{m+n}^i, at $\mathbf{r}$.

1: **for** each sender s_j that receives REQ_i, for $1 \leq j \leq n$ and $n+1 \leq j', j'' \leq m+n$ **do**
2: s_j sets REC^i to be a message consisting of its input a_j and REQ^i.
3: s_j broadcasts REC^i to all parties in Q_j.
4: Let $Q_{j'}$ and $Q_{j''}$ be the right neighboring quorums of Q_j in the circuit.
5: All parties in Q_j calculate parties, $q_{j'}^i$ and $q_{j''}^i$, of indices $B_{j'}^i$ and $B_{j''}^i$, to be added to $S_{j'}^B$ and $S_{j''}^B$ respectively.*
6: All parties in Q_j send REC^i to all parties in $S_{j'}^B$ and to all parties in $S_{j''}^B$.
7: All parties in Q_j send $REC^1, \ldots, REC^{i-1}$ to $q_{j'}^i$ and $q_{j''}^i$.
8: **end for**
9: **for** $j \leftarrow n+1, \ldots, m+n-1$ **do** ▷ recomputes
10: Let $Q_{j'}$ and $Q_{j''}$ be the right neighboring quorums of Q_j in the circuit.
11: All i parties in S_j^B calculate parties, $q_{j'}^i$ and $q_{j''}^i$, of indices $B_{j'}^i$ and $B_{j''}^i$, to be added to $S_{j'}^B$ and $S_{j''}^B$ respectively.
12: Party q_j^i calculate all parties in $S_{j'}^B$ and $S_{j''}^B$ using $B_{j'}^1, \ldots, B_{j'}^i$ and $B_{j''}^1, \ldots, B_{j''}^i$.
13: for all $1 \leq k \leq i$, q_j^k performs its operation on its inputs producing an output, b_j^k.
14: **for** $k \leftarrow 1, \ldots, i$ **do**
15: q_j^k sends b_j^k and REC^k to parties $q_{j'}^i$ and $q_{j''}^i$.
16: q_j^i sends b_j^i and REC^i to parties $q_{j'}^k$ and $q_{j''}^k$.
17: **end for**
18: **end for**
19: All i parties in S_{m+n} broadcast b_{m+n}^i and REC^i to all parties in Q_{m+n}.
20: All parties in Q_{m+n} send b_{m+n}^i and REC^i to party $\mathbf{r}$. ▷ $\mathbf{r}$ receives the result.

* $B_j^i = B^i[j-n, |U_j|]$ is the index of the party, q_j^i, which is selected u.a.r. from the parties in U_j in round i of *RECOMPUTE*; note that the nodes of U_j are sorted by their IDs, for $n+1 \leq j \leq m+n$.

2.4 *UPDATE*

When a computation is corrupted and *CHECK* detects this corruption, *UPDATE* is called. The *UPDATE* algorithm is described formally as Algorithm 11. When *UPDATE* starts, all parties in each quorum in the circuit are notified.

The main purpose of *UPDATE* is to 1) determine the location in which the corruption occurred; and 2) mark the parties that are in conflict.

To determine the location in which the corruption occurred, *UPDATE* calls *INVESTIGATE* (Algorithm 12) to investigate the corruption situation by letting each party involved in *COMPUTE-CIRCUIT* or *CHECK* broadcast all messages they have received

Algorithm 10 RESEND-RESULT(i, C^i, b^i_{m+n}) ▷ Party **r** sends back the result, b^i_{m+n}, through a DAG of subquorums, S^C_j's, to n senders, for $n+1 \leq j \leq m+n$.

1: Party **r** calls $SIGN([i, C^i, b^i_{m+n}, \mathbf{r}], Q_{m+n})$. ▷ signs it by Q_{m+n}'s private key.
2: Party **r** sets $RES^i = ([i, C^i, b^i_{m+n}, \mathbf{r}]_{k_s}, k_p)$. ▷ (k_p, k_s) : public/private key pair of Q_{m+n}.
3: Party **r** sends RES^i to all parties of quorum Q_{m+n}.
4: All parties in Q_{m+n} calculate party, $q^i_{m+n} \in U_{m+n}$, of index C^i_{m+n} to be added to S^C_{m+n}.*
5: **for** $j \leftarrow m+n, \ldots, n+1$ **do** ▷ sends back the result through a DAG of subquorums.
6: Let $Q_{j'}$ and $Q_{j''}$ be the left neighboring quorums of Q_j in the circuit, for $n+1 \leq j', j'' \leq m+n$. **
7: All i parties in S^C_j calculate parties, $q^i_{j'}$ and $q^i_{j''}$, of indices $C^i_{j'}$ and $C^i_{j''}$, to be added to $S^C_{j'}$ and $S^C_{j''}$ respectively.
8: Party q^i_j calculate all parties in $S^C_{j'}$ and $S^C_{j''}$ using $C^1_{j'}, \ldots, C^i_{j'}$ and $C^1_{j''}, \ldots, C^i_{j''}$.
9: **for** $k \leftarrow 1, \ldots, i$ **do** ▷ k refers to the rounds prior to round i.
10: Party q^k_j sends RES^k to parties $q^i_{j'}$ and $q^i_{j''}$.
11: Party q^i_j sends RES^i to parties $q^k_{j'}$ and $q^k_{j''}$.
12: **end for**
13: **end for**
14: **for** $k \leftarrow n, \ldots, 1$ **do** ▷ The input quorums forward RES^i to all senders.
15: Let $Q_{k'}$ and $Q_{k''}$ be the right neighboring quorums of Q_k in the circuit.
16: All i parties in $S_{k'}$ and all parties in $S_{k''}$ call $BROADCAST(RES^i, Q_k, Q_k)$.
17: All parties in Q_k send RES^i to sender s_k.
18: **end for**

* $C^i_j = C^i[j-n, |U_j|]$ is the index of the party, q^i_j, which is selected u.a.r. from the parties in U_j in round i of *RESEND-RESULT*; note that the nodes of U_j are sorted by their IDs, for $n+1 \leq j \leq m+n$.
** Recall that there are no subquorums for the input quorums.

Algorithm 11 UPDATE ▷ Party $q' \in Q'$ calls *UPDATE* after it detects a corruption.

1: q' broadcasts to all parties in Q' the fact that it calls *UPDATE* along with the messages it has received in this call to *COMPUTE*.
2: The parties in Q' verify that q' received inconsistent messages before proceeding.
3: Q' notifies all quorums in the circuit via all-to-all communication that *UPDATE* is called.
4: INVESTIGATE ▷ investigates all participants to determine corruption locations.
5: MARK-IN-CONFLICTS ▷ marks the parties that are in conflict.

Algorithm 12 INVESTIGATE ▷ investigates the parties that have participated.

1: **for** each party, q, involved in the last call to *COMPUTE-CIRCUIT* or *CHECK* **do**
2: q compiles all messages they have received (and from whom) and they have sent (and to whom) in the last call to *COMPUTE-CIRCUIT* or *CHECK*.
3: q broadcasts these messages to all parties in its quorum and neighboring quorums.
4: **end for**

or sent. Then, *UPDATE* calls *MARK-IN-CONFLICTS* (Algorithm 13) in order to mark the parties that are *in conflict*, where a pair of parties is in conflict if at least one of these parties broadcasted messages that conflict with the messages broadcasted by the other party in this pair. Note that each pair of parties that are in conflict has at least one bad party. Recall that if $(1/2 - \gamma)$-fraction of parties in any quorum are marked, for any constant $\gamma > 0$, e.g., $\gamma = 0.01$, they are set unmarked. Also, for each pair of leaders that get marked, their quorums elect another pair of unmarked leaders.

Algorithm 13 MARK-IN-CONFLICTS ▷ marks the parties that are in conflict.

for each pair of parties, (q_x, q_y), that is in conflict*, in quorums (Q_x, Q_y) **do**
    party q_y broadcasts a *conflict* message, $\{q_x, q_y\}$, to all parties in Q_y.
    each party in Q_y forwards $\{q_x, q_y\}$ to all parties in Q_x.
    all parties in Q_x (or Q_y) send $\{q_x, q_y\}$ to the other quorums that has q_x (or q_y).
    each quorum has q_x or q_y sends $\{q_x, q_y\}$ to its neighboring quorums.
end for
for each party q that receives conflict message $\{q_x, q_y\}$ **do**
    q marks q_x and q_y in its marking table.
end for
if $(1/2 - \gamma)$-fraction of parties in any quorum have been marked, for $\gamma = 0.01$ **then**
    each of these parties is set unmarked in all its quorums.
    each of these parties is set unmarked in all its neighboring quorums.
end if
for each pair of leaders, (q_x, q_y), that is in conflict, in quorums (Q_x, Q_y) **do**
    ELECT(Q_x) and ELECT(Q_y) to elect a pair of unmarked leaders, (q'_x, q'_y).
    Q_x and Q_y notify their neighboring quorums with (q'_x, q'_y).
end for

* A pair of parties, (q_x, q_y), is *in conflict* if: 1) q_x was scheduled to send an output to q_y at some point in the last call to *COMPUTE-CIRCUIT* or *CHECK*; and 2) q_y does not receive an expected message from q_x in *INVESTIGATE*, or q_y receives a message in *INVESTIGATE* that is different than the message that it has received from q_x in the last call to *COMPUTE-CIRCUIT* or *CHECK*.

3 Analysis

In this section, we sketch the proof of Theorem 1. Due to space constraints, all proofs are provided in the full paper. Throughout this section all logarithms are base 2.

Recall that in each round of *CHECK*, a new DAG of unmarked parties is formed, where a new unmarked party is selected u.a.r. from each quorum in the circuit.

Definition 1. *A* Deception DAG*, D_i, is the maximal subgraph of the new DAG of unmarked parties that are selected u.a.r. in round i, with the following properties: 1) it has only bad parties; 2) it receives all its inputs, and each input is provided correct by at least one good party; 3) it is rooted at one party, which does not provide the correct output to at least one good party; and 4) all other outputs this DAG has are provided correct.*

If the adversary corrupts the output of the root party in a deception DAG in any round, then it has to keep corrupting this output by a deception DAG in each subsequent round; otherwise, the good parties that expect to receive this output in each round will call *UPDATE* due to receiving inconsistent output messages.

We say that a deception DAG, D_i, in round i extends in round $i+1$ if there exists a deception DAG, D_{i+1}, in round $i+1$ such that at least 1) there is a subquorum that has one party in D_i and one party in D_{i+1}; and 2) there is a subquorum that has one party in D_{i+1} but has no party in D_i.

Also, we say that a deception DAG, D_i, in round i shrinks in round $i+1$ if there exists a deception DAG, D_{i+1}, in round $i+1$ such that 1) for each subquorum, S, in which D_{i+1} has a party, D_i has a party in S; and 2) at least there is a subquorum that has one party in D_i but has no party in D_{i+1}.

Further, we say that a deception DAG, D_i, shrinks logarithmically from round i to round $i+1$ if $|D_{i+1}| = O(\log |D_i|)$.

3.1 *CHECK*

In the following lemmas, we first show that any deception DAG in any round never extends in any subsequent round. Then we show that with probability at least $1/2$, any deception DAG shrinks logarithmically from round to round. This will imply that the expected number of rounds to shrink any deception DAG to size zero is $O(\log^* m)$.

Note that in any round i, if a deception DAG, D_i, shrinks to a deception DAG, D_{i+1}, of size zero in round $i+1$, then the good party that did not receive the correct output from D_i in round i will receive the correct output in round $i+1$. As a result, this good party will call *UPDATE* declaring that it has received inconsistent output messages.

Lemma 1. *Any deception DAG in any round never extends in any direction.*

Proof. We know by definition that the deception DAG is bordered by the good parties that provide the inputs to the DAG, and the good parties that receive the outputs from the DAG.

In each round, all parties of each subquorum in round i send their outputs to the new added party in the next subquorum. Thus, the good parties that provide the correct inputs to the deception DAG of round i, will provide the correct inputs to the deception DAGs in all subsequent rounds.

Moreover, in each round, the new added party in each subquorum forwards its output to all parties in its subquorum. Note that each good party has previously received a message, it verifies this message with all subsequent messages it receives, and if it receives inconsistent messages or fails to receive an expected message, then it calls *UPDATE*.

Therefore, all good parties that border a deception DAG in any round will border all subsequent deception DAGs. □

Now we show that any deception DAG shrinks logarithmically from round to round with probability at least $1/2$.

Definition 2. Rooted Directed Acyclic Graph (R-DAG) *is a DAG in which, for a vertex u called the root and any other node v, there is at least one directed path from v to u.*

Lemma 2. *Given any R-DAG, of size n, in which each node has indegree of at most d and survives independently with probability at most p such that $0 < p \leq \frac{1}{d} - \epsilon$, for any constant $\epsilon > 0$, then the probability of having a subgraph, rooted at some node, with surviving nodes, of size $\Omega(\frac{\log n}{(1-pd)^2})$ is at most $1/2$.*

Proof. This proof makes use of the following three propositions, but first we define some notations.

Given an R-DAG, $D(V, E)$, with size n and maximum indegree d, after each node survives independently with probability at most p such that $0 < p \leq \frac{1}{d} - \epsilon$, for any constant $\epsilon > 0$, we explore D to find a subgraph with only surviving nodes, of size more than k, rooted at an arbitrary node v (assuming that node v survives).

Let $D'(v)$ be the maximal subgraph of surviving nodes, rooted at node v. Let each node in D have a status, which is either *inactive*, *active* or *neutral*. During the exploration process, the status of nodes is changed. A node x is inactive if $x \in D'(v)$ and its children are explored determining which one is in $D'(v)$. A node x is active if $x \in D'(v)$ but its children are not explored yet. A node x is neutral if it is neither active nor inactive, i.e., node x and its children are not explored yet.

The exploration process runs in at most $k > 0$ steps. Initially, we set an arbitrary surviving node, v, active and all other nodes neutral. At each step i, we choose an active node, w_i, in an arbitrary way, and we explore all its children. For all $(w_i, w'_i) \in E$ and w'_i survives and is neutral, we set w'_i active, otherwise w'_i remains as it is. Then, we set w_i inactive. Note that at any step, if there is no active node, the exploration process terminates. Now let d_i be the maximum number of children of node w_i for $1 \leq i \leq k$, i.e.,

$$d_i = \begin{cases} deg(w_i) - 1 & \text{if } w_i \in V - root(D), \\ deg(w_i) & otherwise. \end{cases}$$

where $deg(w_i)$ is the degree of node w_i and $root(D)$ is the root node of D. For $1 \leq i \leq k$, let X_i be a non-negative random variable for the number of surviving neutral children of w_i, and let Y_i be a non-negative random variable for the number of surviving non-neutral children of w_i. Note that $Y_1 = 0$. So X_i follows a binomial distribution with parameters $(d_i - Y_i)$ and p, i.e., $X_i \sim Bin(d_i - Y_i, p)$. Let A_i be a non-negative random variable for the total number of active nodes after i steps, for $1 \leq i \leq k$.

Proposition 1. $A_i = \begin{cases} \sum_{i=1}^{k} X_i - (k-1) & \text{if } A_{i-1} > 0, \\ 0 & otherwise. \end{cases}$

Proof. Since the process starts initially with one active node v, $A_0 = 1$. Now we have two cases of A_{i-1} to compute A_i, $1 \leq i \leq k$:

Case 1 (process terminates before i steps): If $A_{i-1} = 0$, then $A_j = 0$ for $i \leq j \leq k$.

Case 2 (otherwise): If $A_{i-1} > 0$, then $A_i = A_{i-1} + X_i - 1$, where after exploring w_i, the total number of active nodes is the number of new active nodes (X_i) due to the exploration of w_i in addition to the total number of active nodes of previous steps (A_{i-1}) excluding w_i that becomes inactive at the end of step i. □

Now let $|D'(v)|$ be the number of nodes in $D'(v)$.

Proposition 2. $Pr(|D'(v)| > k) \leq Pr(\sum_{i=1}^{k} X_i \geq k)$.

Proof. To prove this proposition, we first prove that $Pr(|D'(v)| > k) \leq Pr(A_k > 0)$.

In order to do that, we prove that $|D'(v)| > k \implies A_k > 0$. If $|D'(v)| > k$, then the exploration process does not terminate before k steps. This implies that after k steps, there are k inactive nodes and at least one active node remains. This follows that $A_k > 0$. Thus, we have

$$Pr(|D'(v)| > k) \leq Pr(A_k > 0). \tag{1}$$

Now we prove that $Pr(A_k > 0) \leq Pr(\sum_{i=1}^{k} X_i - (k-1) > 0)$. To do that, we prove that $A_k > 0 \implies \sum_{i=1}^{k} X_i - (k-1) > 0$. If $A_k > 0$, then $A_j > 0$ for all $1 \leq j \leq k$. By Proposition 1, we obtain that $\sum_{i=1}^{j} X_i - (j-1) > 0$ for all $1 \leq j \leq k$. This follows that

$$Pr(A_k > 0) \leq Pr(\sum_{i=1}^{k} X_i - (k-1) > 0). \tag{2}$$

By Inequalities 1 and 2, we obtain

$$Pr(|D'(v)| > k) \leq Pr(\sum_{i=1}^{k} X_i - (k-1) > 0),$$

or equivalently,

$$Pr(|D'(v)| > k) \leq Pr(\sum_{i=1}^{k} X_i > k-1).$$

Since k is a positive integer, we have

$$Pr(|D'(v)| > k) \leq Pr(\sum_{i=1}^{k} X_i \geq k).$$

□

Proposition 3. $Pr(\sum_{i=1}^{k} X_i \geq k) \leq e^{-\frac{(1-pd)^2 k}{1+pd}}$.

Proof. To prove this proposition, we first make use of stochastic dominance. For $1 \leq i \leq k$, let $X_i^+ \sim Bin(d, p)$, and let $X_1^+, ..., X_k^+$ be independent random variables. We know that $Y_i \geq 0$ and $d_i \leq d$ for $1 \leq i \leq k$.

By Theorem (1.1) part (a) of [26], for all $1 \leq i \leq k$, X_i^+ first-order stochastically dominates X_i, i.e., X_i^+ is stochastically larger than X_i. Hence, $\sum_{i=1}^{k} X_i^+$ is stochastically larger than $\sum_{i=1}^{k} X_i$. Thus, we have

$$Pr(\sum_{i=1}^{k} X_i \geq k) \leq Pr(\sum_{i=1}^{k} X_i^+ \geq k).$$

Now let $S_k = \sum_{i=1}^{k} X_i^+$. By Chernoff bounds, for $\delta > 0$, we obtain

$$Pr(S_k \geq (1+\delta)E(S_k)) \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{E(S_k)} \leq e^{-\frac{\delta^2}{2+\delta}E(S_k)}.$$

We know that $S_k \sim Bin(kd, p)$. Thus, $E(S_k) = pdk$. Therefore, we have

$$Pr(S_k \geq (1+\delta)pdk) \leq e^{-\frac{\delta^2}{2+\delta}pdk}.$$

For $\delta = \frac{1-pd}{pd}$, we obtain

$$Pr(S_k \geq k) \leq e^{-\frac{(1-pd)^2 k}{1+pd}}.$$

□

Now by Propositions 2 and 3, we have

$$Pr(|D'(v)| > k) \leq e^{-\frac{(1-pd)^2 k}{1+pd}}.$$

We know that node v survives with probability at most p. Thus, we obtain

$$Pr(|D'(v)| > k) \leq pe^{-\frac{(1-pd)^2 k}{1+pd}}.$$

Union bound over n nodes, then the probability that there exists a subgraph of D, rooted at one node, having only surviving nodes, of size more than k is at most

$$nPr(|D'(v)| > k) \leq npe^{-\frac{(1-pd)^2 k}{1+pd}}.$$

Note that $npe^{-\frac{(1-pd)^2 k}{1+pd}} \leq 1/2$ when $k \geq \frac{1+pd}{(1-pd)^2 \log e}\log(2pn)$. Thus, the probability of having such a subgraph of size more than $\frac{1+pd}{(1-pd)^2 \log e}\log(2pn)$, or equivalently, $\Omega\left(\frac{\log n}{(1-pd)^2}\right)$, is at most $1/2$. □

Corollary 1. *For any R-DAG, of size n, the probability of having a subgraph, rooted at one node, with surviving nodes, of size at least $n/2$ is at most $1/2$.*

Now, if a deception DAG shrinks logarithmically in a successful step, then how many successful steps to shrink this deception DAG to a deception DAG of size zero or even of a constant size?

Let $f(n) = c\log n$, and let $f^{(i)}(n)$ be the function of applying function f, i times, over n. Also, we let $\log^{(i)}(n)$ be the function of applying logarithm i times over n.

Fact 1 $\forall i \geq 1 : \log^{(i)}(n) \geq \log c + 1$, $f^{(i)}(n) \leq 2c\log^{(i)}(n)$.

Proof. We prove by induction over $i \geq 1$ that for $\log^{(i)}(n) \geq \log c + 1$,

$$f^{(i)}(n) \leq 2c\log^{(i)}(n).$$

Base case: for $i = 1$, by definition,

$$f(n) = c\log n \leq 2c\log n.$$

Induction hypothesis: for $\log^{(j)}(n) \geq \log c + 1$,

$$\forall j < i, f^{(j)}(n) \leq 2c\log^{(j)}(n).$$

Induction step: by definition,

$$f^{(i)}(n) = f(f^{(i-1)}(n)).$$

By induction hypothesis, for $\log^{(i-1)}(n) \geq \log c + 1$,

$$f^{(i-1)}(n) \leq 2c \log^{(i-1)}(n).$$

Then,

$$f^{(i)}(n) \leq f(2c \log^{(i-1)}(n)) = c \log(2c \log^{(i-1)}(n)),$$

or equivalently,

$$f^{(i)}(n) \leq c(1 + \log c + \log^{(i)}(n)) \leq 2c \log^{(i)}(n),$$

for $\log^{(i)}(n) \geq \log c + 1$. □

Now let $f^*(n)$ be the smallest value i such that $f^{(i)}(n) \leq c(2c + \log c + 1)$.

Fact 2 $\forall n > c(2c + \log c + 1), f^*(n) \leq \log^* n - \log^*(\log c + 1)$.

Proof. Let $k = \log^* n - \log^* (\log c + 1) - 1$. Then, $\log^{(k)}(n) \geq \log c + 1$. By Fact 1,

$$f^{(k)}(n) \leq 2c \log^{(k)}(n).$$

With a further application of f to $f^{(k)}(n)$, we have

$$f^{(k+1)}(n) \leq c \log(2c \log^{(k)}(n)) = c(1 + \log c + \log^{(k+1)}(n)).$$

We know that $\log^{(k+1)}(n) \leq 2c$. Thus, we obtain

$$f^{(k+1)}(n) \leq c(1 + \log c + 2c).$$

Therefore, by definition,

$$f^*(n) \leq k + 1 = \log^* n - \log^* (\log c + 1).$$

□

Lemma 3. *Assume that any deception DAG of size n' shrinks to a deception DAG of size $c \log n'$ in a successful step, for any constant $c \geq 1$. Then, for a deception DAG of size $n > c(2c + \log c + 1)$, after $\log^* n - \log^* (\log c + 1)$ successful steps, it shrinks to a deception DAG of size at most $c(2c + \log c + 1)$.*

Proof. Fact 2 proves this lemma. □

Let p be the probability of selecting an unmarked bad party uniformly at random in any quorum. Recall that the fraction of bad parties in any quorum is at most $1/4$, and at any time the fraction of unmarked parties in any quorum is at least $1/2 + \gamma$, for any constant $\gamma > 0$. Thus, $p \leq \frac{1/2}{1+2\gamma}$.

Now we show the expected number of rounds to shrink any deception DAG to size zero.

Lemma 4. *With probability at least* $1/2$*, any deception DAG of size* m *shrinks to size zero in* $8(\log^* m + 2(\log c + 1))$ *rounds, where* $c = \frac{2(1+2p)}{\log e(1-2p)^2}$ *and* $p \leq \frac{1/2}{1+2\gamma}$*, for any constant* $\gamma > 0$*.*

Proof. Given a deception DAG, of size m. By Lemma 1, the deception DAG never extends over rounds. For shrinking deception DAGs over rounds, we make use of Lemma 2 to shrink logarithmically any deception DAG of size more than $c(2c + \log c + 1)$; otherwise, deception DAGs shrink geometrically using Corollary 1.

Let X_i be an indicator random variable that is equal 1 if the deception DAG in round i shrinks logarithmically in round $i + 1$; and 0 otherwise.

By Lemma 3, after having at most $\log^* m - 1$ of X_i's equal 1, the deception DAG of size at most m shrinks to a size of at most $c(2c + \log c + 1)$.

Also let Y_j be an indicator random variable that is equal 1 if the deception DAG of size at most $c(2c + \log c + 1) \leq 4c^2$ in round j shrinks geometrically by at most half the size in round $j + 1$; and 0 otherwise.

Thus, in order to shrink the deception DAG of size n to 0, we require at most $\log^* m - 1$ of X_i's equal 1 and at most $2 \log c + 3$ of Y_j's equal 1.

Note that in each round, the receiver that is elected by the output quorum is good with probability at least $1/2$. Then, by Lemma 2, $X_i = 1$ with probability at least $1/4$; and by Corollary 1, $Y_j = 1$ with probability at least $1/4$.

Now let

$$X = \sum_{i=1}^{8(\log^* m - 1)} X_i$$

and

$$Y = \sum_{j=8(\log^* m - 1)+1}^{8(\log^* m + 2(\log c + 1))} Y_j.$$

Also let Z_k be an indicator random variable that is 1 with probability $1/4$; and 0 otherwise, for $1 \leq k \leq 8(\log^* m + 2(\log c + 1))$; and let

$$Z = \sum_{k=1}^{8(\log^* m + 2(\log c + 1))} Z_k.$$

We know that for all i, j, k, both X_i and Y_j are stochastically larger than Z_k. Thus, $X + Y$ is stochastically larger than Z. Therefore,

$$\Pr\left(X + Y \geq \log^* m + 2(\log c + 1)\right) \geq \Pr\left(Z \geq \log^* m + 2(\log c + 1)\right),$$

or equivalently,

$$1 - \Pr\left(X + Y < \log^* m + 2(\log c + 1)\right) \geq 1 - \Pr\left(Z < \log^* m + 2(\log c + 1)\right).$$

Thus, we obtain

$$\Pr\left(X + Y < \log^* m + 2(\log c + 1)\right) \leq \Pr\left(Z < \log^* m + 2(\log c + 1)\right).$$

Note that $E(Z) = 2(\log^* m + 2(\log c + 1))$. Since the Z_k's are independent random variables, by Chernoff bounds,

$$\Pr\left(Z < 2(1-\delta)(\log^* m + 2(\log c + 1))\right) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{2(\log^* m + 2(\log c + 1))}.$$

For $\delta = \frac{1}{2}$ and $m \geq 3$,

$$\Pr\left(Z < \log^* m + 2(\log c + 1)\right) < \frac{1}{2}.$$

Thus, the probability that *CHECK* succeeds in finding a corruption and calling *UPDATE* is at least $1/2$. □

Now we show that for the adversary to maximize the number of rounds without detecting corruptions is to consider the maximum deception DAG in the first round.

Lemma 5. *For the adversary to maximize the expected number of rounds, in which no corruption detected, is to corrupt the output of the root party in the maximum deception DAG of the first round.*

Proof. For the case that the adversary considers multiple deception DAGs that are overlapped in the same round. Then the adversary corrupts more than one output in some round. Now let D' be the maximum deception DAG in this round. By Lemma 4, each of these deception DAGs shrinks to size zero in an expected number of rounds that is at most the expected number of rounds that D' shrinks to size zero.

Similarly, the case that the adversary considers multiple disjoint deception DAGs in the same round.

Therefore, for the adversary to maximize the expected number of rounds without any corruption detected is to consider only the maximum deception DAG in the first round of *CHECK*. □

The next lemma shows that *CHECK* catches corruptions with probability $\geq 1/2$.

Lemma 6. *Assume some party selected uniformly at random in the last call to COMPUTE-CIRCUIT has corrupted a computation. Then when the algorithm CHECK is called, with probability at least* $1/2$*, some party will call UPDATE.*

Proof. Recall that the number of gates in the circuit is m. Thus, by Lemmas 4 and 5, this request is sent reliably to all input quorums in $O(\log^* m)$ rounds with probability at least $1/2$. Note that each request, REQ^i, has a round number i. Hence, at any round, if any good party in any input quorum receives a request of round number i and has not received $(i-1)$ requests of proper round numbers, then it will call *UPDATE*.

If all input quorums receive all requests properly in all $O(\log^* m)$ rounds, then *RECOMPUTE* must be called properly $O(\log^* m)$ times by all input quorums. By Lemmas 4 and 5, the result is computed and sent reliably to the output quorum in $O(\log^* m)$ rounds with probability at least $1/2$.

Similarly, we know that in *RECOMPUTE*, the round number i is enclosed in REC^i, which is propagated with the computation results from the senders to the output quorum.

Thus, at any round, if any good party in the output quorum receives a result with a round number i and has not received $(i-1)$ results with proper rounds numbers, then it will call *UPDATE*.

Finally, if all parties in the output quorum receive all results properly in all $O(\log^* m)$ rounds, then *RESEND-RESULT* must be called $O(\log^* m)$ times by the output quorum. By Lemmas 4 and 5, the result of the computation is sent back reliably to all senders in $O(\log^* m)$ rounds with probability at least $1/2$. Thus, the probability that *CHECK* succeeds in finding a corruption and calling *UPDATE* is at least $1/2$. □

3.2 *UPDATE*

Lemma 7. *If some party selected uniformly at random in the last call to COMPUTE-CIRCUIT or CHECK has corrupted a computation, then UPDATE will identify a pair of neighboring quorums Q and Q' such that at least one pair of parties in these quorums is in conflict and at least one party in such pair is bad.*

Proof. First, we show that if a pair of parties x and y is in conflict, then at least one of them is bad. Assume not. Then both x and y are good. This implies that party x would have truthfully reported what it received and sent; any result that x has computed would have been sent directly to y; and y would have truthfully reported what it received from x. But this is a contradiction, since for x and y to be in conflict, y must have reported that it received from x something different than what x reported sending.

Now consider the case where a selected unmarked bad leader corrupted the computation in the last call to *COMPUTE-CIRCUIT*. By Lemma 6, with probability at least $1/2$, some party, $q' \in Q'$, will call *UPDATE*. Recall that in *UPDATE* q' broadcasts all messages it has received to all parties in Q'. These parties verify if q' received inconsistent messages before proceeding.

In *UPDATE*, we know that each party, $q \in Q$, participated in the last call to *COMPUTE* broadcasts what it has received and sent to all parties in Q. Thus, all parties of Q verify the correctness of q's computation. Thus, if the corruption occurs due to an incorrect computation made by a bad party, this corruption will be detected and all parties will know that this party is bad.

Now if all parties compute correctly and *CHECK* detects a corruption, then we show that there is some pair of parties will be in conflict. Assume this is not the case. Thus, by the definition of corruption, there must be a deception DAG, in which all inputs are provided correct and an output is corrupted at party q'. Then each pair of parties, $(q_j, q_k) \in (Q_j, Q_k)$, in the deception DAG that is rooted at q', is not in conflict, for $n+1 \leq j < k \leq m+n$. Thus, we have that 1) this DAG received all its inputs correct; 2) all parties compute correctly; and 3) no pair of parties is in conflict. This implies that it must be the case that q' received the correct output. But if this is the case, then q' that initially called *UPDATE* would have received no inconsistent messages. This is a contradiction since in such a case, this party would have been unsuccessful in trying to initiate a call to *UPDATE*. Thus, *UPDATE* will find two parties that are in conflict, and at least one of them will be bad. □

The next lemma bounds the number of calls to *UPDATE* before all bad parties are marked.

Lemma 8. *UPDATE is called $O(t)$ times before all bad parties are marked.*

Proof. By Lemma 7, if a corruption occurred in the last call to *COMPUTE-CIRCUIT*, and it is caught by *CHECK*, then *UPDATE* is called. *UPDATE* identifies at least one pair of parties that is in conflict, and each of such pairs has at least one bad party.

Now let g be the number of marked good parties, and let b be the number of marked bad parties. Also let

$$f(b,g) = b - (\frac{p}{1-p})g.$$

Note that since $0 < p \leq \frac{1/2}{1+2\gamma}$, for any constant $\gamma > 0$, $0 < \frac{p}{1-p} \leq \frac{1}{1+4\gamma}$.

For each corruption caught, at least one bad party is marked, and so $f(b,g)$ increases by at least $\frac{1-2p}{1-p}$ since b increases by at least 1 and g increases by at most 1. When $(1/2-\gamma)$-fraction of parties in any quorum Q get unmarked, for any constant $\gamma > 0$, $f(b,g)$ further increases by at least 0 since b decreases by at most $p|Q|(1/2-\gamma)$ and g decreases by at least $(1-p)|Q|(1/2-\gamma)$. Hence, $f(b,g)$ is monotonically increasing by at least $\frac{1-2p}{1-p}$ for each corruption caught. When all bad parties are marked, $f(b,g) \leq t$.

Therefore, after at most $(\frac{1-p}{1-2p})t$, or at most $(1+\frac{1}{2\gamma})t$, calls to *UPDATE*, all bad parties are marked. □

3.3 Proof of Theorem 1

We first show the message cost, the number of operations and the latency of our algorithms. By Lemma 8, the number of calling *UPDATE* is at most $O(t)$. Thus, the resource cost of all calls to *UPDATE* is bounded as the number of calls to *COMPUTE* grows large. Therefore, for the amortized cost, we consider only the cost of the calls to *COMPUTE-CIRCUIT* and *CHECK*.

When a computation is performed through a circuit of m gates with a circuit depth ℓ, *COMPUTE-CIRCUIT* has message cost $O(m+n\log n)$, number of operations $O(m+n\log n)$ and latency $O(\ell)$. *CHECK* has message cost $O((m+n\log n)(\log^* m)^2)$, number of operations $O((m+n\log n)\log^* m)$ and latency $O(\ell\log^* m)$, but *CHECK* is called only with probability $1/(\log^* m)^2$. Hence, the call to *CHECK* has an amortized expected message cost $O(m+n\log n)$, amortized computational operations $O(\frac{m+n\log n}{\log^* m})$ and an amortized expected latency $O(\ell/\log^* m)$.

In particular, if we call *COMPUTE* $\mathcal{L}$ times, then the expected total number of messages sent will be $O(\mathcal{L}(m+n\log n)+t(m\log^2 n))$ with expected total number of computational operations $O(\mathcal{L}(m+n\log n)+t(m\log n\log^* m))$ and latency $O(\ell(\mathcal{L}+t))$. This is true since *UPDATE* is called $O(t)$ times and each call to *UPDATE* has message cost $O(m\log^2 n)$ with computational operations $O(m\log n\log^* m)$ and latency $O(\ell)$.

Recall that by Lemma 8, the number of times *CHECK* must catch corruptions before all bad parties are marked is $O(t)$. In addition, if a bad party caused a corruption during a call to *COMPUTE-CIRCUIT*, then by Lemmas 6 and 7, with probability at least $1/2$, *CHECK* will catch it. As a consequence, it will call *UPDATE*, which marks the parties that are in conflict. $UPDATE$ is thus called with probability $1/(\log^* m)^2$, so the expected total number of corruptions is $O(t(\log^* m)^2)$.

4 Conclusion and Future Work

We have presented algorithms for reliable multiparty computations. These algorithms can significantly reduce message cost and number of computational operations to be

asymptotically optimal. The price we pay for this improvement is the possibility of computation corruption. In particular, if there are $t \leq (\frac{1}{4} - \epsilon)n$ bad parties, for any constant $\epsilon > 0$, our algorithm allows $O(t(\log^* m)^2)$ computations to be corrupted in expectation.

Many problems remain. First, it seems unlikely that the smallest number of corruptions allowable by an attack-resistant algorithm with optimal message complexity is $O(t(\log^* m)^2)$. Can we improve this to $O(t)$ or else prove a non-trivial lower bound? Second, we allow the inputs of parties to reveal. Can we maintain the privacy of these inputs? Finally, we assume a partially synchronous communication model, which is crucial for our *CHECK* algorithm to detect computation corruptions over rounds. Can we extend this algorithm to fit for asynchronous computations?

References

1. Fiat, A., Saia, J.: Censorship resistant peer-to-peer networks. Theory of Computing **3**(1) (2007) 1–23
2. Hildrum, K., Kubiatowicz, J.: Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In: Distributed Computing. Volume 2848. (2003) 321–336
3. Naor, M., Wieder, U.: A simple fault tolerant distributed hash table. IPTPS'03 (2003) 88–97
4. Scheideler, C.: How to spread adversarial nodes? rotate! STOC '05 (2005) 704–713
5. Fiat, A., Saia, J., Young, M.: Making chord robust to Byzantine attacks. ESA'05 (2005) 803–814
6. Awerbuch, B., Scheideler, C.: Towards a scalable and robust DHT. Theory of Computing Systems **45**(2) (2009) 234–260
7. King, V., Lonargan, S., Saia, J., Trehan, A.: Load balanced scalable Byzantine agreement through quorum building, with full information. ICDCN'11 (2011) 203–214
8. Frisanco, T.: Optimal spare capacity design for various protection switching methods in ATM networks. Volume 1 of ICC'97. (1997) 293–298
9. Iraschko, R.R., MacGregor, M.H., Grover, W.D.: Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks. IEEE/ACM Transactions on Networking **6**(3) (1998) 325–336
10. Murakami, K., Kim, H.S.: Comparative study on restoration schemes of survivable ATM networks. Volume 1 of INFOCOM '97. (1997) 345–352
11. Van Caenegem, B., Wauters, N., Demeester, P.: Spare capacity assignment for different restoration strategies in mesh survivable networks. Volume 1 of ICC'97. (1997) 288–292
12. Xiong, Y., Mason, L.G.: Restoration strategies and spare capacity requirements in self-healing ATM networks. IEEE/ACM Transactions on Networking **7**(1) (1999) 98–110
13. Boman, I., Saia, J., Abdallah, C., Schamiloglu, E.: Brief announcement: Self-healing algorithms for reconfigurable networks. Volume 4280 of SSS'06. (2006) 563–565
14. Saia, J., Trehan, A.: Picking up the pieces: Self-healing in reconfigurable networks. IPDPS'08 (2008) 1–12
15. Hayes, T., Rustagi, N., Saia, J., Trehan, A.: The forgiving tree: a self-healing distributed data structure. PODC '08 (2008) 203–212
16. Hayes, T.P., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. PODC '09 (2009) 121–130
17. Pandurangan, G., Trehan, A.: Xheal: localized self-healing using expanders. PODC '11 (2011) 301–310
18. Sarma, A.D., Trehan, A.: Edge-preserving self-healing: keeping network backbones densely connected. In: IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). (2012) 226–231

19. Knockel, J., Saad, G., Saia, J.: Self-healing of Byzantine faults. SSS'13 (2013) 98–112
20. Yao, A.C.: Protocols for secure computations. SFCS '82 (1982) 160–164
21. Beaver, D.: Efficient multiparty protocols using circuit randomization. CRYPTO 91. (1992) 420–432
22. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. STOC '88 (1988) 1–10
23. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. STOC '89 (1989) 73–85
24. Prabhakaran, M., Sahai, A.: Secure Multi-Party Computation. Volume 10. IOS Press (2013)
25. Kate, A., Goldberg, I.: Distributed key generation for the internet. ICDCS '09 (2009) 119–128
26. Klenke, A., Mattner, L.: Stochastic ordering of classical discrete distributions. Advances in Applied Probability **42**(2) (2010) 392 – 410