A Theoretical and Empirical Evaluation of an Algorithm for Self-Healing Computation[*]

George Saad and Jared Saia

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

Abstract. In the problem of reliable multiparty computation (RMC), there are n parties, each with an individual input, and the parties want to jointly compute a function f over n inputs; note that it is not required to keep the inputs private. 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 RMC 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. Empirical results show that our algorithm can reduce message cost by a factor of 425 when compared with algorithms that are not self-healing.

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 (RMC)*. In the RMC 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.

[*] This research is partially supported by NSF Award SATC 1318880.

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]

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

Our main result is an algorithm for RMC 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.

This paper is organized as follows. In Section 2, we describe our model. Our main theorem is given in Section 3, and we provide a technical overview in Section 4. The related work is discussed in Section 5. Section 6 describes our algorithms. The analysis of our algorithms is shown in Section 7. Section 8 gives empirical results showing how our algorithms improve the efficiency of the butterfly networks of [1]. Finally, we conclude and describe problems for future work in Section 9.

The theoretical result of this paper was first presented as an extended abstract in [2]. This paper is the full version of that extended abstract.

2 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, where 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. Note that we assume partial synchronous communication, but with a *rushing* adversary. The adversary is rushing in the sense that the bad nodes can wait to receive all messages in a round, before they need to send out their own messages for that round.

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 directly communicate with node q in the overlay network.

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

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

[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.

function multiple times (with changing inputs). However, we can also compute multiple functions with our algorithm.

3 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 RMC 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 $O(t(\log^* m)^2)$.

Our experimental results in Section 8 show that our algorithms (Section 6) reduce the message cost, compared to the naive algorithm (Section 4.3), by a factor of 425 for $n = 8{,}191$.

4 Technical Overview

In this section, we briefly describe a quorum graph as well as a naive (no self-healing) computation algorithm and our self-healing approach.

4.1 Quorums and Quorum Graph

Our algorithms make critical use of quorums and a quorum graph. We define a quorum to be a set of $\Theta(\log n)$ parties, of which at most $1/4$ are bad. Many results show how to create and maintain a network of quorums [1,3,4,5,6,7,8]. 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 directly 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.

4.2 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.

[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.

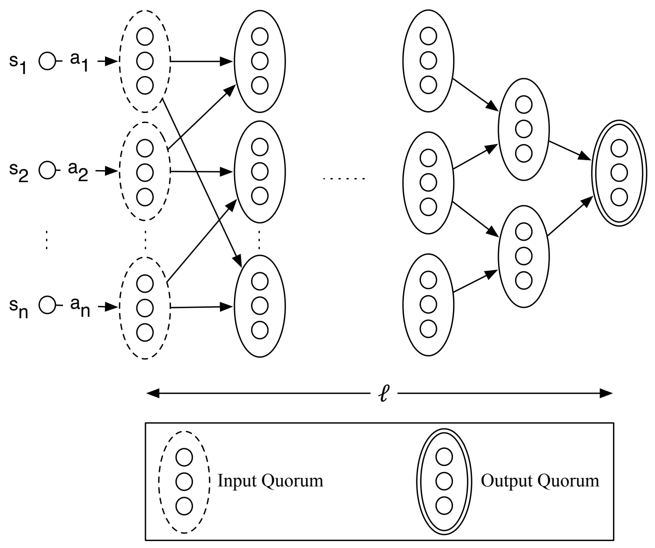

Fig. 1. Quorum Graph

Figure 1 shows the quorum graph in which the computation is performed from the left to the right. In particular, the input quorums are the leftmost quorums and the output quorum is the rightmost quorum in the quorum graph. Note that s_i is the i^{th} party that seeds an input a_i to the quorum network, for $1 \leq i \leq n$.

4.3 Naive Computation

A correct but inefficient way to solve RMC 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 (See Figure 2). They then forward this output back from right to left through the quorum graph using the same all-to-all communication and majority filtering (See Figure 3).

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

4.4 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.

[4] We note that such asymptotic improvements can be significant for large networks. For example, if $n = 4{,}095$, then our algorithm reduces message cost by a factor of $O(\log^2 n) = 336$.

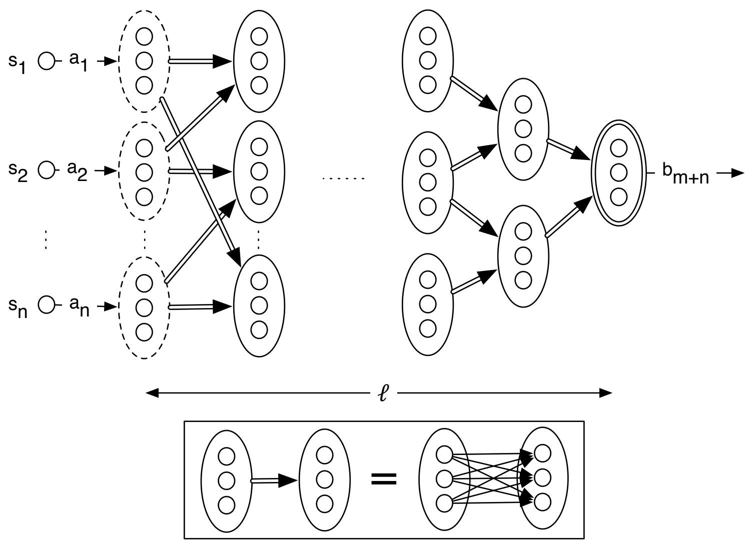

Fig. 2. The parties provide n inputs to a circuit of quorums via all-to-all communication in the Naive Algorithm

To address this issue, we provide *CHECK* (Section 6.3). This algorithm determines if there has been a corruption, and if so, it calls *RECOVER* (Section 6.4), 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 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; 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 *RECOVER*. 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

[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 9) shows that after $O(t)$ markings, all bad parties will be marked.

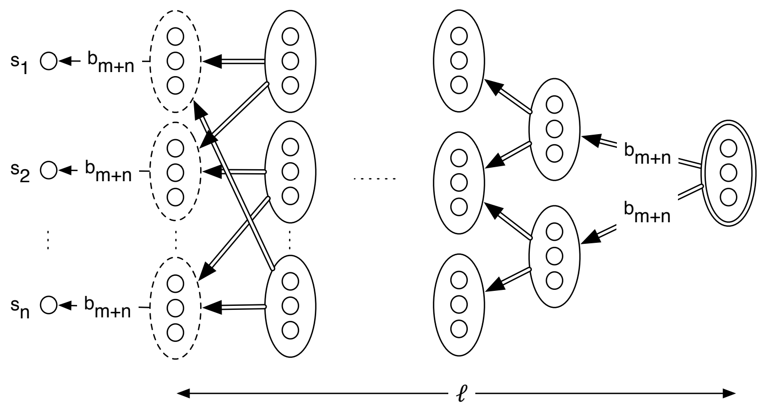

Fig. 3. The output quorum sends back the result, b_{m+n}, to the n parties through a circuit of quorums via all-to-all-communication in the Naive Algorithm

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 3) 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 RMC problem, while incurring an expected $O(t(\log^* m)^2)$ corruptions.

5 Related Work

Our results are inspired by recent work on self-healing algorithms. Early work of [9,10,11,12,13] 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.

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

More recent work [14,15,16,17,18,19] considers models where the following process repeats indefinitely: an adversary deletes some nodes in the network, and the algorithm 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 [14,15,16]; the shortest path between any pair of nodes does not increase by much [17]; expansion properties of the network are approximately preserved [18]; and keeping network backbones densely connected [19].

This paper particularly builds on [20]. 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 [20], 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 (RMC) is closely related to the problem of secure multiparty computation (MPC) which has been studied extensively for several decades (see e.g. [21,22,23,24,25] or the recent book [26]). RMC is simpler than MPC in that it does not require inputs of the parties to remain private. Our algorithm for RMC 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.

6 Our Algorithms

In this section, we describe our algorithms: *COMPUTE*, *EVALUATE*, *CHECK* and *RECOVER*.

Our algorithms aim at detecting corruptions and marking the bad parties. Note that the parties that are marked are not allowed to participate in the computation; but they still can provide inputs to the circuit. Note further that all parties are initially 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 the private key of a quorum Q, to a set of parties S, it calls $BROADCAST(msg, Q, S)$.

6.1 *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 [27].

In particular, we use ($|Q|$, $\frac{3|Q|}{4} - 1$)-DKG to generate for each quorum Q the following: 1) a (distributed) private key of Q, where a private key share is generated for each party in Q; 2) a public key of Q to verify each message signed by the (distributed) private key of Q; and 3) a public key share for each party in Q in order to verify any message signed by the private key share of this party.

Note that for each quorum, Q, the public key of Q and the public key share of each party in Q are known to all parties in Q and all parties in the neighboring quorums.

Recall that a party x calls $BROADCAST(msg, Q, S)$ in order to send a message msg to all parties in S after signing msg by the private key of quorum Q. Signing a message msg, by the private key of Q, is formally stated in *SIGN* (msg, Q) (Algorithm 2). Note that we let the message msg be signed by the private key of Q in order to fulfill the following: 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 permission to 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 $S$.
```

Lemma 1. *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 $O(1)$.*

Proof. The proof is immediate from the algorithm description of *BROADCAST* and *SIGN*.

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 $Q$.
```

6.2 *COMPUTE*

Now we describe our main algorithm, *COMPUTE* (Algorithm 3), which calls *EVALUATE* (Algorithm 4). In *EVALUATE*, the n parties broadcast their inputs to the input quorums; note that we assume that all parties provide their inputs to the circuit in the same round. The input quorums forward these inputs to a circuit of m leaders in order

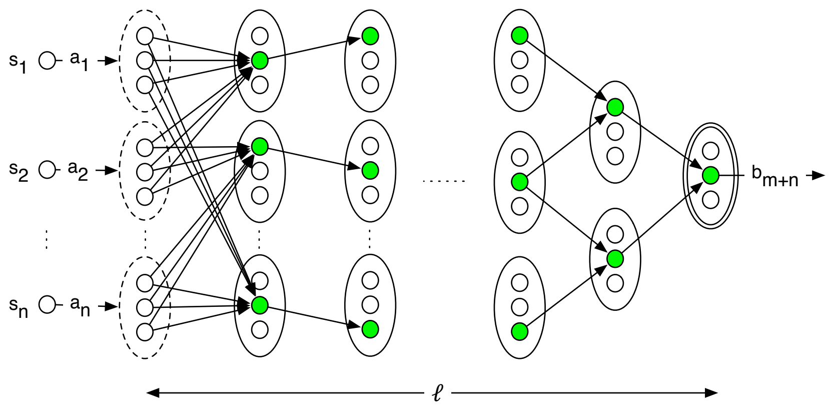

Fig. 4. The senders provide the network with the inputs to perform the computation through a circuit of leaders

to perform the computation and provide the result to the output quorum (See Figure 4). Then this result is sent back to all senders (all parties) through the same circuit (See Figure 5). Note that we define a leader of a quorum as a representative party of all parties in this quorum, and its leadership is known to all parties in this quorum and the neighboring quorums.

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

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

In the presence of an adversary, *EVALUATE* 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) [23,24,25]. Note that we assume the standard RAM model of computation: the number of bits in the real number chosen, in step 1 of Algorithm 5, is logarithmic in $m + n$. *CHECK* is triggered in order to detect with probability at least $1/2$ if a computation was corrupted in the last call to *EVALUATE*.

Unfortunately, while *CHECK* can determine if a corruption occurred, it does not locate where the corruption originally occurred. Thus, when *CHECK* detects a corruption, *RECOVER* (Algorithm 11) is called. In each call to *RECOVER*, 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. Note that if at least $(1/2 - \gamma)$-fraction of

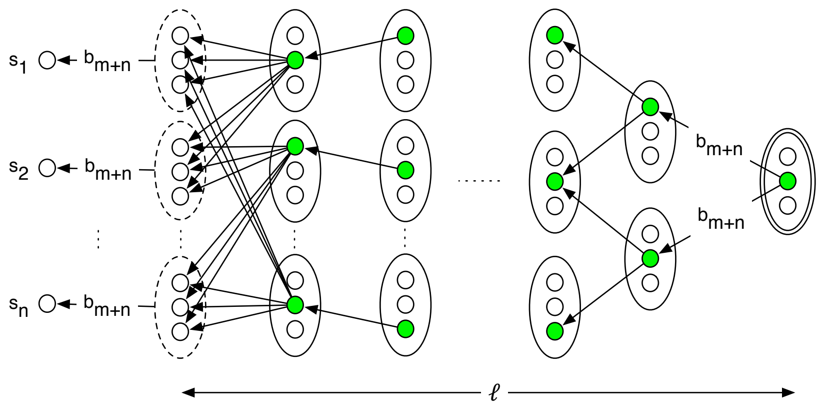

Fig. 5. The output quorum sends back the result of the computation to the senders through the same circuit of leaders

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 in all their neighboring quorums.

Moreover, we use *BROADCAST* in *EVALUATE* and *CHECK* in order to handle any accusation issued in *RECOVER* 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.

6.3 *CHECK*

In this section, we describe the *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. The election process is stated formally in *ELECT* (Algorithm 6).

The elected party $\mathbf{r}$ selects u.a.r. a set of unmarked parties to participate in round i. It may not know how many unmarked parties in each quorum, but we assume that it knows m', which is the maximum number of parties in any quorum. Party $\mathbf{r}$ constructs an m by m' array of random integers, A_i, selected uniformly at random. A_i represents a selection to a set of unmarked parties to be added to a DAG of subquorums, S_j^A's, for $n+1 \leq j \leq m+n$, in round i. Note that $A^i[k,k']$ refers to an unmarked party in a

Algorithm 4 *EVALUATE* ▷ 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.

```
 1: for $i = 1, \ldots, n$ do                                        ▷ provides the inputs to the circuit.
 2:     Party $s_i$ calls BROADCAST $(a_i, Q_i, Q_i)$.  ▷ $s_i$ broadcasts its input $a_i$ to all parties in $Q_i$.
 3:     All parties in $Q_i$ send $a_i$ to the leaders of the right neighboring quorums of $Q_i$.
 4: end for
 5: for $i = n + 1, \ldots, m + n - 1$ do                             ▷ performs the computation.
 6:     Let $Q_{i'}$ and $Q_{i''}$ be the right neighboring quorums of $Q_i$ in the circuit.
 7:     if leader $q_i \in Q_i$ receives all its inputs then
 8:         $q_i$ performs an operation on its inputs producing an output, $b_i$.
 9:         $q_i$ sends $b_i$ to leader $q_{i'} \in Q_{i'}$ and to leader $q_{i''} \in Q_{i''}$.
10:     end if
11: end for
12: if leader $q_{m+n} \in Q_{m+n}$ receives all its inputs then
13:     $q_{m+n}$ performs an operation on its inputs producing an output, $b_{m+n}$.
14:     $q_{m+n}$ broadcasts $b_{m+n}$ to all parties in $Q_{m+n}$.
15: end if
16: for $i = m + n, \ldots, n + 1$ do                  ▷ sends back the result to the leftmost leaders.
17:     Let $Q_{i'}$ and $Q_{i''}$ be the left neighboring quorums of $Q_i$ in the circuit, for $n+1 \leq i', i'' \leq$
    $m + n$. *
18:     Leader $q_i \in Q_i$ sends $b_{m+n}$ to leader $q_{i'} \in Q_{i'}$ and to leader $q_{i''} \in Q_{i''}$.
19: end for
20: for $i = 1, \ldots, n$ do    ▷ sends result to all parties after broadcasting it to the input quorums.
21:     The leaders of $Q_i$'s right neighboring quorums call $BROADCAST(b_{m+n}, Q_i, Q_i)$.
22:     All parties in $Q_i$ send $b_{m+n}$ to sender $s_i$.
23: end for
```

* Recall that there are no leaders in the input quorums.

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

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

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

```
1: Let each party in the set of unmarked parties, $U \subset Q$, be assigned a unique integer from 0
   to $|U| - 1$. Note that these unique integers are assigned in ascending order to the unmarked
   parties as their IDs are ascendingly ordered.
2: Each party in $Q$ chooses an input: an integer uniformly distributed between 0 and $|U| - 1$.
3: The parties of $Q$ perform MPC to find the output: the sum of all their inputs modulo $|U|$.
4: 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 $1 \le k \le m+n$. Also let m' be the maximum number of parties in any quorum. Further, let subquorum, S_j^A, be initially empty, for all $n+1 \le j \le m+n$.

```
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 an array of $m$ by $m'$ random integers, $A^i$.**
4:     REQUEST ($i, A^i$)                         ▷ $\mathbf{r}$ requests all senders to recompute.
5:     RECOMPUTE                                  ▷ recomputes, producing the result, $b^i_{m+n}$, at $\mathbf{r}$.
6:     RESEND ($i, A^i, b^i_{m+n}$)                ▷ $\mathbf{r}$ sends back $b^i_{m+n}$ to all parties.
7: end for
```

* $c = \frac{2(1+2p)}{\log e(1-2p)^2}$; note that for any quorum Q_k, $p \le 1/2 - \epsilon$, is the probability of selecting a bad party u.a.r. from U_k, for any constant $\epsilon > 0$.
** $A^i[k, k']$ is a uniformly random integer between 1 and k', for $1 \le k \le m$ and $1 \le k' \le m'$.

Note that: if a party has previously received k_p, then it verifies each subsequent message with it; also if a party receives inconsistent messages or fails to receive and verify an expected message, then it initiates a call to *RECOVER*.

quorum Q_k, of size $|Q_k| = k'$, for $1 \le k \le m$ and $1 \le k' \le m'$. Note further that A^i has a size of $O(m \log n \log \log n)$ bits.

After $\mathbf{r}$ constructs A_i, it calls *REQUEST* to send a request through the DAG of subquorums, to the n senders in order to recompute (See Figure 6).

The recomputation process is stated formally as *RECOMPUTE* (Algorithm 9) and is shown in Figure 7. In this process, each sender that receives this request provides its input to redo the computation through a DAG of subquorums, S_j^A's, producing the result at the output quorum. When $\mathbf{r}$ receives this result, it calls *RESEND* (Algorithm 10) in order to send the result back to the senders through a DAG of subquorums S_j^A's, for $n+1 \le j \le m+n$ (See Figure 8).

Note that in *ELECT* (Q), the parties of quorum Q perform MPC [23,24,25] to elect an unmarked party uniformly at random from Q. We know that at least half of the unmarked parties in Q are good. Thus, the elected party is good with probability at least $1/2$. MPC requires a message cost and a number of computational operations that are polylogarithmic functions in n, and it runs in $O(1)$ time.

Note further that 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 *RECOVER*.

CHECK detects message corruptions with probability at least $1/2$. It requires $O((m+n\log n)(\log^* n)^2)$ message cost and $O(\ell \log^* n)$ latency. But since *CHECK* is triggered with probability $1/(\log^* m)^2$, it has expected message cost $O(m + n\log n))$ with expected latency $O(\ell/\log^* n)$.

An example run of *CHECK* is illustrated in Figures 9 and 10. These figures show that in each round, a circuit of parties is formed, where one party is selected u.a.r. from each quorum in the quorum graph. Each of these parties performs the appropriate gate operation on its inputs providing an output which is an input for the next gate in the circuit.

Algorithm 8 *REQUEST* (i, A^i) $\triangleright$ **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 $SIGN([i, A^i, \mathbf{r}], Q_{m+n})$.    $\triangleright$ signs $[i, A^i, \mathbf{r}]$ by $Q_{m+n}$'s private key
2:  Party $\mathbf{r}$ sets $REQ^i = ([i, A^i, \mathbf{r}]_{k_s}, k_p)$.    $\triangleright$ $(k_p, k_s)$ : 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, $q^i_{m+n} \in U_{m+n}$, of index $A^i_{m+n}$ to be added to $S^A_{m+n}$.*
5:  for $j \leftarrow m+n, \ldots, n+1$ do    $\triangleright$ 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, $q^i_{j'}$ and $q^i_{j''}$, of indices $A^i_{j'}$ and $A^i_{j''}$, to be added to $S^A_{j'}$ and $S^A_{j''}$ respectively.
8:      Party $q^i_j$ calculates all parties in $S^A_{j'}$ and $S^A_{j''}$ using $A^1_{j'}, \ldots, A^i_{j'}$ and $A^1_{j''}, \ldots, A^i_{j''}$.
9:      for $k \leftarrow 1, \ldots, i$ do    $\triangleright$ $k$ refers to the rounds prior to round $i$.
10:         Party $q^k_j$ sends $REQ^k$ to parties $q^i_{j'}$ and $q^i_{j''}$.
11:         Party $q^i_j$ sends $REQ^i$ to parties $q^k_{j'}$ and $q^k_{j''}$.
12:     end for
13: end for
14: for $k \leftarrow n, \ldots, 1$ do    $\triangleright$ 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
```

* $A^i_j = A^i[j-n, |U_j|]$ 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 *REQUEST*; note that all parties in 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 9 *RECOMPUTE* ▷ n senders provide inputs to a DAG of subquorums, S_j^A'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 $Q_j$ recalculate parties, $q_{j'}^i$ and $q_{j''}^i$, of indices $A_{j'}^i$ and $A_{j''}^i$, and make sure
   that they are already added to $S_{j'}^A$ and $S_{j''}^A$ respectively, in *REQUEST*.
6:     All parties in $Q_j$ send $REC^i$ to all parties in $S_{j'}^A$ and to all parties in $S_{j''}^A$.
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 $S_j^A$ recalculate parties, $q_{j'}^i$ and $q_{j''}^i$, of indices $A_{j'}^i$ and $A_{j''}^i$, and make
   sure that they are already added to $S_{j'}^A$ and $S_{j''}^A$ respectively, in *REQUEST*.
12:    Party $q_j^i$ recalculates all parties in $S_{j'}^A$ and $S_{j''}^A$ using $A_{j'}^1, \ldots, A_{j'}^i$ and $A_{j''}^1, \ldots, A_{j''}^i$.
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.
```

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

```
1: Party r calls $SIGN([i, A^i, b^i_{m+n}, \mathbf{r}], Q_{m+n})$.    ▷ signs it by $Q_{m+n}$'s private key.
2: Party r sets $RES^i = ([i, A^i, b^i_{m+n}, \mathbf{r}]_{k_s}, k_p)$. ▷ $(k_p, k_s)$ : 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 $Q_{m+n}$ recalculate party, $q^i_{m+n} \in U_{m+n}$, of index $A^i_{m+n}$, and make sure that
   it is already added to $S^A_{m+n}$, in REQUEST.
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 $n+1 \le$
   $j', j'' \le m+n$. *
7:     All $i$ parties in $S^A_j$ recalculate parties, $q^i_{j'}$ and $q^i_{j''}$, of indices $A^i_{j'}$ and $A^i_{j''}$, and make
   sure that they are already added to $S^A_{j'}$ and $A^C_{j''}$ respectively, in REQUEST.
8:     Party $q^i_j$ recalculates all parties in $S^A_{j'}$ and $S^A_{j''}$ using $A^1_{j'}, \ldots, A^i_{j'}$ and $A^1_{j''}, \ldots, A^i_{j''}$.
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
```

* Recall that there are no subquorums for the input quorums.

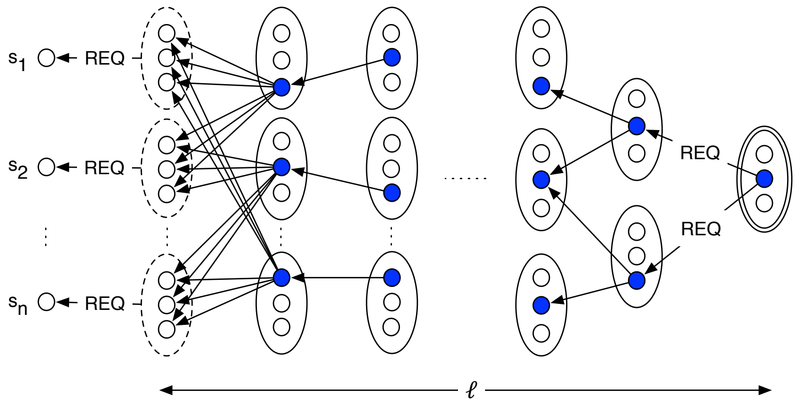

Fig. 6. The output quorum sends a request to the senders to provide the circuit with the inputs in order to redo the computation through a circuit of unmarked parties selected u.a.r.

For a given circuit of parties and a given round, there is a white or black node depending on whether the party selected in that particular round and that particular gate (quorum) is good (white) or bad (black).

Informally, we define a *deception DAG*, D_i, in a round, i, as a rooted DAG of bad nodes that are selected in this round to be added to the subquorums in the quorum graph.

In Figure 9, we show the deception DAGs that are chosen by the adversary to corrupt the computation over rounds. In particular, the adversary's strategy is 1) to corrupt the output of the maximum deception DAG in the first round; and 2) to keep corrupting this output by nesting levels of deception DAGs in all subsequent rounds. These nesting levels of deception DAGs are outlined in this figure.

There are two key points by which *CHECK* detects corruptions: 1) any deception DAG in any round never extends in any subsequent round; and 2) any deception DAG is expected to shrink logarithmically in size from round to round. This will imply that any deception DAG shrinks to size zero after $O(\log^* n)$ rounds, at which the corruption is detected.

Deception DAGs never extend. We know that each good party that receives an input message in any round has to receive the same input message in all subsequent rounds; otherwise, it will call *RECOVER*. Moreover, for each round, we know that all parties in each subquorum send their output message to the new party that is added in this round to the next subquorum in the circuit. Thus, those good parties that provide their output message to the deception DAG of this round will provide the same output message to all subsequent deception DAGs.

Recall that in each round, every party that is added to each subquorum, S, sends its output message to all parties in the next subquorum, S', in the quorum graph. Thus, all good parties in S' that receive an input message through a deception DAG in any round expect to receive the same input message in all subsequent rounds. Also, each

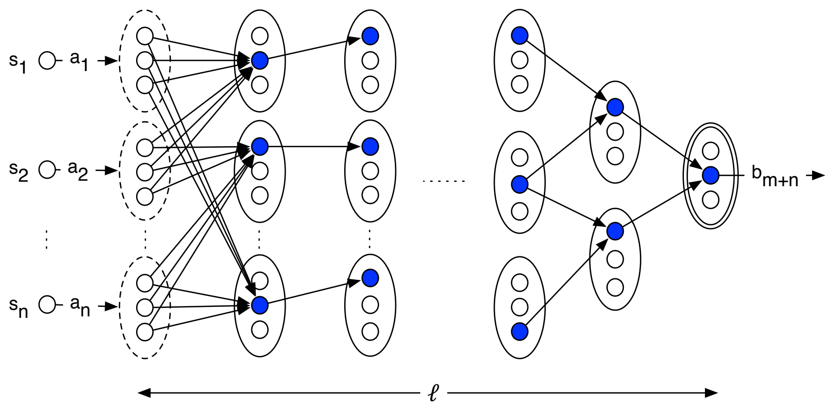

Fig. 7. The n parties provide n inputs to a circuit of unmarked parties selected u.a.r. to redo the computation producing the result at the output quorum.

good party that did not receive the correct input message in any round must not receive this message in all subsequent rounds; otherwise, it will initiate a call to *RECOVER*.

Deception DAGs shrink logarithmically. A key lemma (Lemma 3) shows that given a rooted DAG of size m, in which each bad node is selected u.a.r. with probability at most $1/2$, the probability of having a rooted subgraph of $\Omega(\log m)$ bad nodes is at most $1/2$. Intuitively, we could expect that $O(\log^* m)$ rounds will suffice to shrink any deception DAG to size zero in a quorum graph of m gates.

When any deception DAG shrinks to size zero, the corruption is detected. Figure 10 shows that when the deception DAG shrinks over rounds to size zero, node x in the last round receives correct input messages. Then node x computes a correct output and sends it to node y; however, node y has not previously received it as an input in this call to *CHECK*. As a result, node y calls *RECOVER* declaring that it has received inconsistent input messages.

Even if all parties in some subquorums are bad, *CHECK* is still being able to detect corruptions. Recall that *CHECK* runs in $O(\log^* m)$ rounds. In each round, new parties are selected u.a.r. to be added to the subquorums. This limits the adversary to know, before all rounds finish, if all parties of any particular subquorum are bad. Thus, the adversary would rather corrupt the output of the maximum deception DAG in the first round and keeps corrupting this output over all subsequent deception DAGs. Note that if the adversary corrupts more than one output in the same round, it will increase the chance of detecting corruptions. Figure 10 shows that even though all parties in some subquorum are bad, some of these parties behave as good parties since they are out of the deception DAGs selected by the adversary.

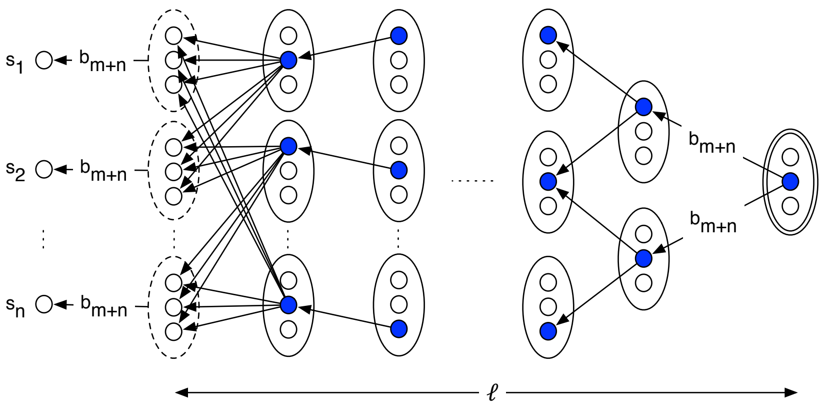

Fig. 8. The output quorum sends back the result of the computation to the senders through a circuit of unmarked parties selected u.a.r.

6.4 *RECOVER*

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

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

```
1: $q'$ broadcasts to all parties in $Q'$ the fact that it calls RECOVER 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 RECOVER is called.
4: INVESTIGATE              ▷ investigates all participants to determine corruption locations.
5: MARK-IN-CONFLICTS                              ▷ marks the parties that are in conflict.
```

The main purpose of *RECOVER* 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, *RECOVER* calls *INVESTIGATE* (Algorithm 12) to investigate the corruption situation by letting each party involved in *EVALUATE* or *CHECK* broadcast all messages they have received or sent. Note that the message size that each participating party broadcasts is $O((m \log n \log \log n + b)(\log^* m)^2)$ bits, given that any input message a_i that is provided by sender s_i has a size of $O(b)$ bits.

Then, *RECOVER* 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 at least $(1/2 - \gamma)$-fraction of parties in any quorum are marked, for

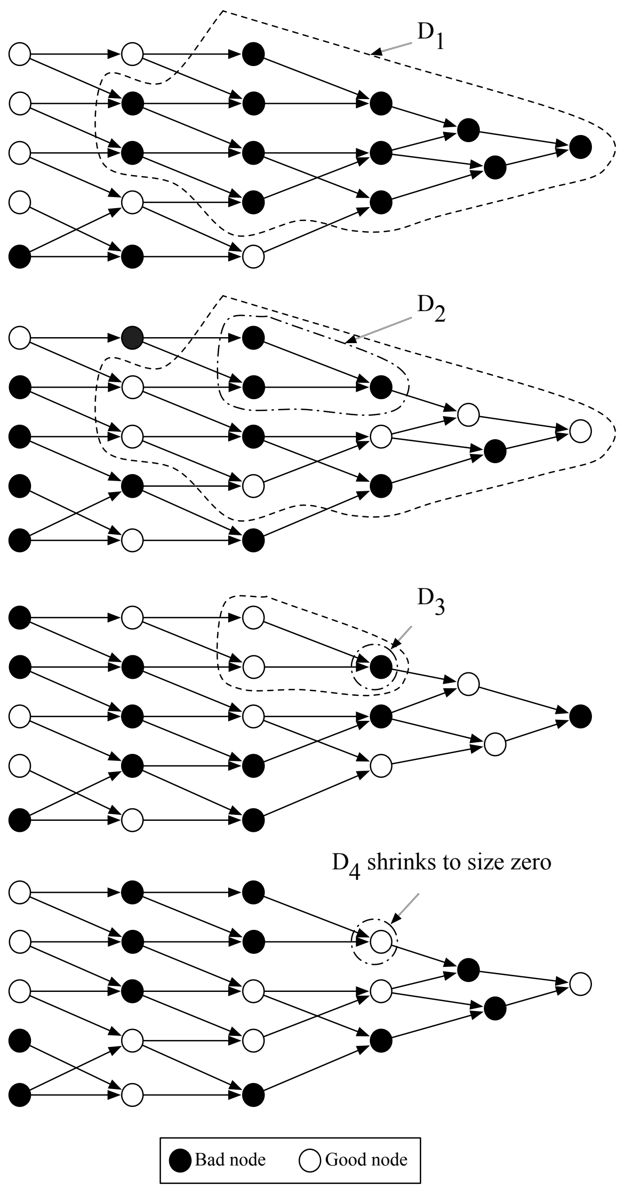

Fig. 9. Example run of *CHECK*

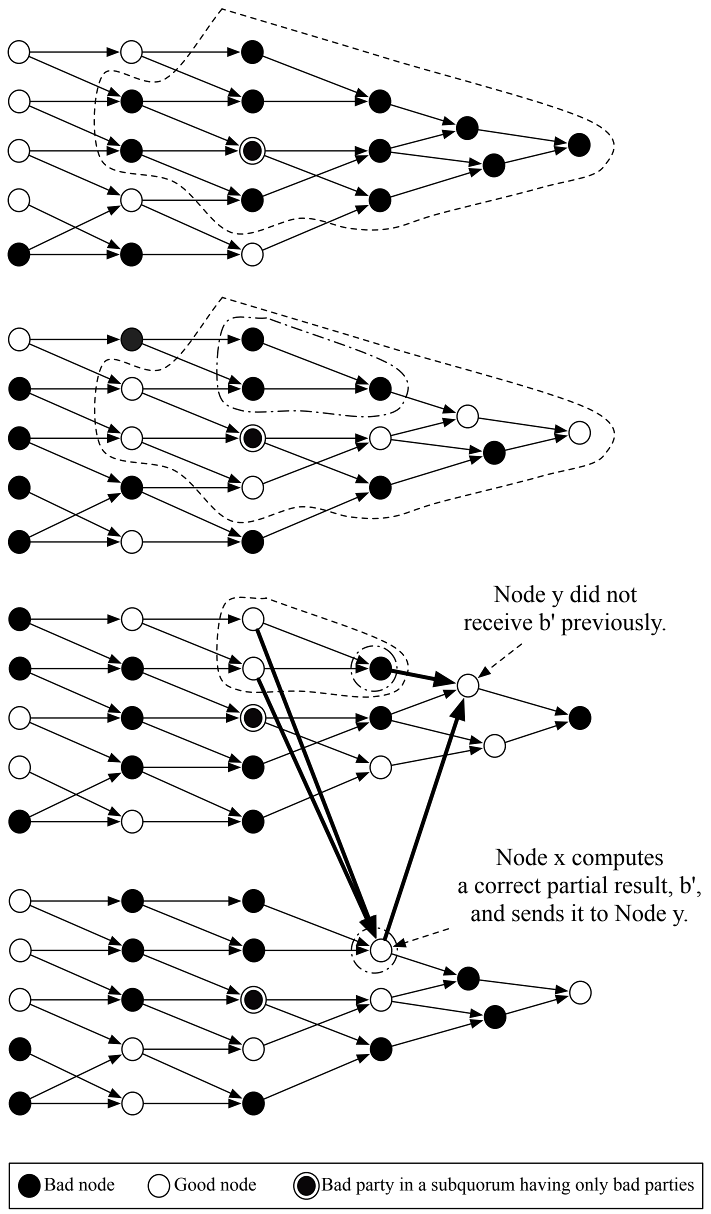
Node y did not receive b' previously.
Node x computes a correct partial result, b', and sends it to Node y.
Bad node
Good node
Bad party in a subquorum having only bad parties

Fig. 10. A corruption is detected after the deception DAG shrinks to size zero.

Algorithm 12 *INVESTIGATE* ▷ investigates the parties that have participated.

```
1: for each party, $q$, involved in the last call to EVALUATE or CHECK do
2:     Party $q$ compiles all messages it has received (and from whom) and has sent (and to
       whom) in the last call to EVALUATE or CHECK.
3:     Party $q$ broadcasts these messages to all parties in its quorum and neighboring quorums.
4: end 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.

```
1: for each pair of parties, $(q_x, q_y)$, that is in conflict*, in quorums $(Q_x, Q_y)$ do
2:     party $q_y$ broadcasts a conflict message, $\{q_x, q_y\}$, to all parties in $Q_y$.
3:     each party in $Q_y$ forwards $\{q_x, q_y\}$ to all parties in $Q_x$.
4:     all parties in $Q_x$ (or $Q_y$) send $\{q_x, q_y\}$ to the other quorums that have $q_x$ (or $q_y$).
5:     each quorum that has $q_x$ or $q_y$ sends $\{q_x, q_y\}$ to its neighboring quorums.
6: end for
7: for each party $q$ that receives conflict message $\{q_x, q_y\}$ do
8:     $q$ marks $q_x$ and $q_y$ in its marking table.
9: end for
10: if a $(1/2 - \gamma)$-fraction of parties in any quorum has been marked, for $\gamma = 0.01$ then
11:     each of these parties is set unmarked in all its quorums.
12:     each of these parties is set unmarked in all its neighboring quorums.
13: end if
14: for each pair of leaders, $(q_x, q_y)$, that is in conflict, in quorums $(Q_x, Q_y)$ do
15:     ELECT($Q_x$) and ELECT($Q_y$) to elect a pair of unmarked leaders, $(q'_x, q'_y)$.
16:     $Q_x$ and $Q_y$ notify their neighboring quorums with $(q'_x, q'_y)$.
17: 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 *EVALUATE* 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 *EVALUATE* or *CHECK*.

7 Analysis

In this section, we prove the lemmas required for Theorem 1. Throughout this section, all logarithms are base 2.

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

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 a correct* *output to at least one good party; and 4) all other outputs of this DAG 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 *RECOVER* 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 1) there is at least one subquorum that has a party in D_i and a party in D_{i+1}; and 2) there is at least one subquorum that has a 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) each subquorum that has a party in D_{i+1} has a party in D_i; and 2) there is at least one subquorum that has a 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 $|D_{i+1}| = O(\log |D_i|)$.

7.1 *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 *RECOVER* declaring that it has received inconsistent output messages.

Lemma 2. *Any deception DAG in any round never extends.*

Proof. We know by definition that any deception DAG is confined by 1) the good parties that provide the inputs to this deception DAG; and 2) the good parties that receive the outputs from it.

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

Moreover, in each round i, each new party that is added to a subquorum in this round sends its output to all i parties in the next subquorum. Thus, the good parties that receive an output from a deception DAG in round i, must receive the same output from all nesting levels of deception DAGs in all subsequent rounds. Similarly, the good parties that did not receive the correct output from a deception DAG in round i must not receive this output from any nesting level of deception DAG in any subsequent round; otherwise, they will call *RECOVER*.

Further, if a good party has previously received k_p (the public key of Q_{m+n}), then it verifies each subsequent message with it; also if a party receives inconsistent messages or fails to receive and verify an expected message, then it initiates a call to *RECOVER*.

Therefore, all good parties that confine a deception DAG in any round will restrict 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 3. *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. The basic ideas of this proof come from [28,29,30]. 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).

Each node in D has a status. The node status is either *inactive*, *active* or *neutral*. A node x is inactive if node x and its children are explored. Note that a node is explored when it is determined whether this node survives or dies. A node x is active if node x is explored 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, all nodes are set neutral, and we pick an arbitrary node, $v \in D$, and set it active (assuming that node v survives).

In each exploration step i, we pick an active node, w_i, in an arbitrary way, and we explore all its children as follows. For all $(w_i, w'_i) \in E$, if w'_i survives and is neutral, we set it active; otherwise w'_i remains as it is. After all children of w_i are explored, we set w_i as inactive. Note that in any exploration step, if there is no active node, the exploration process terminates.

Now we define a list of notations to be used in the following propositions. 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 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. *For $1 \leq i \leq k$, $A_i = \begin{cases} \sum_{j=1}^{i} X_j - (i-1) & \text{if } A_{i-1} > 0, \\ 0 & otherwise. \end{cases}$*

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. □

Let $D'(v)$ be the maximal subgraph of surviving nodes, rooted at node v. Also 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 [31,32]. 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 [33], 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

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

Note that we initially assumed that node v survives. However, node v survives with probability at most p. By definition if node v does not survive, $|D'(v)| = 0$. Note further that $k > 0$. Thus, we obtain

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

Union bound over n nodes, we have

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

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

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

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

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

Base case: for $i = 1$, by definition, $g(n) = c\log n \leq 2c\log n$.
Induction hypothesis: for $\log^{(j)}(n) \geq \log c + 1$, $\forall j < i, g^{(j)}(n) \leq 2c\log^{(j)}(n)$.
Induction step: by definition, $g^{(i)}(n) = g(g^{(i-1)}(n))$. By induction hypothesis, for $\log^{(i-1)}(n) \geq \log c + 1$, $g^{(i-1)}(n) \leq 2c\log^{(i-1)}(n)$. Then, we have

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

or equivalently,

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

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

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

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

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

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

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

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

We know that $\log^{(k+1)}(n) \leq 2c$. Thus, we obtain $g^{(k+1)}(n) \leq c(1 + \log c + 2c)$. Therefore, by definition, $g^*(n) \leq k + 1 = \log^* n - \log^* (\log c + 1)$. □

Lemma 4. *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 5. *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 2, the deception DAG never extends over rounds. For shrinking deception DAGs over rounds, we make use of Lemma 3 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 4, 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 3, $X_i = 1$ 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 *RECOVER* is at least $1/2$. □

Now we show that for the adversary to maximize the number of rounds in which no corruption is detected, is to select the maximum deception DAG of the first round.

Lemma 6. *The best strategy to maximize the expected number of rounds in which no corruption is detected, is for the adversary to select the largest deception DAG present in the first round.*

Proof. Let D_m be the maximum deception DAG in the first round, and let d_m be the expected number of rounds that D_m shrinks to size zero.

For any deception DAG, D, that the adversary selects in the first round, by Lemma 5, D shrinks to size zero in an expected number of rounds that is at most d_m. If the adversary selects two or more deception DAGs in the first round, then each of these deception DAGs shrinks in an expected number of rounds that is at most d_m.

Therefore, for the adversary to maximize the expected number of rounds in which no corruption detected, is to select the maximum deception DAG in the first round of *CHECK*. □

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

Lemma 7. *Assume some party selected uniformly at random in the last call to EVALUATE has corrupted a computation. Then when the algorithm CHECK is called, with probability at least* $1/2$*, some party will call RECOVER.*

Proof. Recall that the number of gates in the circuit is m.

Case 1: ***REQUEST* has corruptions:**
When a corruption occurs in any round of *REQUEST*, by Lemmas 5 and 6, this corruption is detected with probability at least $1/2$.

Recall that in *REQUEST*, for each round j, REQ^j that has $([j, A^j, \mathbf{r}]_{k_s}, k_p)$ is sent from the output quorum to the senders through a graph of subquorums. If any good party in any subquorum receives REQ^j and has not received all REQ^i, for $1 \leq i < j$, then it will call *RECOVER*.

Case 2: ***REQUEST* has no corruptions and *RECOMPUTE* has corruptions:**
When no corruption occurs in any round of *REQUEST*, all request messages REQs are correctly received at the input quorums in all $O(\log^* m)$ rounds and *RECOMPUTE* must be called $O(\log^* m)$ times by all input quorums.

When a corruption occurs in any round of *RECOMPUTE*, by Lemmas 5 and 6, this corruption is detected with probability at least $1/2$.

Recall that in *RECOMPUTE*, for every round j, REC^j that has $([j, A^j, \mathbf{r}]_{k_s}, k_p)$ is propagated with the computation result from the senders to the output quorum through a graph of subquorums. If any good party in any subquorum receives a computation result with REC^j and has not received $(j-1)$ results with all REC^i, for $1 \leq i < j$, then it will call *RECOVER*.

Case 3: ***REQUEST*** **has no corruptions,** ***RECOMPUTE*** **has no corruptions and** ***RESEND*** **has corruptions:**
When no corruption occurs in any round of *REQUEST* and *RECOMPUTE*, all parties in the output quorum receive all computation results correctly in all $O(\log^* m)$ rounds and *RESEND* must be called $O(\log^* m)$ times by the output quorum.

When a corruption occurs in any round of *RESEND*, by Lemmas 5 and 6, this corruption is detected with probability at least $1/2$.

Recall that in *RESEND*, for each round j, RES^j that has $([j, A^j, b^j_{m+n}, \mathbf{r}]_{k_s}, k_p)$ is sent from the output quorum to the senders through a graph of subquorums. If any good party in any subquorum receives RES^j and has not received all RES^i, for $1 \leq i < j$, then it will call *RECOVER*.

Otherwise: no corruption occurs.

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

7.2 *RECOVER*

Lemma 8. *If some party selected uniformly at random in the last call to EVALUATE or CHECK has corrupted a computation, then RECOVER 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 *EVALUATE*. By Lemma 7, with probability at least $1/2$, some party, $q' \in Q'$, will call *RECOVER*. Recall that in *RECOVER* q' broadcasts all messages it has received to all parties in Q'. These parties verify if q' received inconsistent messages before proceeding.

In *RECOVER*, 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 that 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 *RECOVER* 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 *RECOVER*. Thus, *RECOVER* 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 *RECOVER* before all bad parties are marked.

Lemma 9. *RECOVER is called $O(t)$ times before all bad parties are marked.*

Proof. By Lemma 8, if a corruption occurred in the last call to *EVALUATE*, and it is caught by *CHECK*, then *RECOVER* is called. *RECOVER* identifies at least one pair of parties that is in conflict, and each of such pairs has at least one bad party.

Recall that p is the probability of selecting an unmarked bad party uniformly at random in a quorum. Now let b be the number of marked bad parties; and let g be the number of marked good parties. Also, let

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

Note that at least $(1/2+\gamma)$-fraction of parties is unmarked in any quorum at any time, for any constant $\gamma > 0$. Note further that $t \leq (\frac{1}{4} - \epsilon)n$ parties, for any constant $\epsilon > 0$. Thus, for any constant $\delta > 0$,

$$0 < p \leq 1/2 - \delta.$$

This implies that

$$0 < \frac{p}{1-p} < 1.$$

Now we show that for any corruption detected, $f(b,g)$ monotonically increases, i.e., $\Delta f(b,g) > 0$. For each corruption caught, at least one bad party and at most one good party are marked. Thus, b increases by at least 1, and g increases by at most 1. This implies that

$$\Delta f(b,g) \geq 1 - \frac{p}{1-p} = \left(\frac{1-2p}{1-p}\right) > 0.$$

After the corruption is detected and the in-conflict parties are marked, if the fraction of marked parties in any quorum Q reaches or exceeds $(1/2-\gamma)$, then these parties are set unmarked. This implies that b decreases by at most $p|Q|(1/2-\gamma)$ and g decreases by at least $(1-p)|Q|(1/2-\gamma)$, or equivalently, $f(b,g)$ increases further by at least 0.

Hence, for each corruption caught,

$$\varDelta f(b,g) \geq \left(\frac{1-2p}{1-p}\right) > 0.$$

Since $0 \leq b \leq t$ and $g \geq 0$, $f(b,g)$ never exceeds t; and when all bad parties are marked, $f(b,g)$ is at most t. Therefore, all bad parties are marked after *RECOVER* is called at most $\left(\frac{1-p}{1-2p}\right) t$ times, or equivalently, at most $\left(\frac{1+2\delta}{4\delta}\right) t$ times, for any constant $\delta > 0$. □

7.3 Proof of Theorem 1

We first show the message cost, the number of operations and the latency of our algorithms. By Lemma 9, the number of calls to *RECOVER* is at most $O(t)$. Thus, the resource cost of all calls to *RECOVER* 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 *EVALUATE* and *CHECK*.

When a computation is performed through a circuit of m gates with a circuit depth ℓ, *EVALUATE* 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 *RECOVER* is called $O(t)$ times and each call to *RECOVER* 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 9, 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 *EVALUATE*, then by Lemmas 7 and 8, with probability at least $1/2$, *CHECK* will catch it. As a consequence, it will call *RECOVER*, which marks the parties that are in conflict. $RECOVER$ is thus called with probability $1/(\log^* m)^2$, so the expected total number of corruptions is $O(t(\log^* m)^2)$.

8 Empirical Results

8.1 Setup

In this section, we empirically compare two algorithms via simulation. We evaluate the following resource costs of these algorithms: message cost, latency, probability of computation corruption and expected total number of corruptions.

The first algorithm we simulate is *no-self-healing* (Section 4.3). This algorithm simply computes via all-to-all communication between quorums that are connected in binary-tree circuits. The second algorithm is *self-healing*, wherein we apply our self-healing algorithm in binary-tree circuits (Section 6).

In our experiments, we consider two quorum graphs. The first quorum graph has a number of parties $n = 4{,}095$ with a number of gates $m = 4{,}095$; and the second quorum graph has $n = 8{,}191$ with $m = 8{,}191$. The two quorum graphs are perfect binary-tree of depth $\ell = \lfloor \log n \rfloor$. We let the quorum be of size $\lfloor 4 \log n \rfloor$ and the sub-quorum be of size $\lfloor 2 \log^* n \rfloor$. Also, we let *COMPUTE* call *CHECK* with probability $1/(\log^* n)^2$. Moreover, we do our experiments for several fractions of bad parties such as $f = \{\frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}\}$, where $f = t/n$.

Our simulations consist of a sequence of calls to *COMPUTE* over the network. Note that in each call, we let the output quorum request that the n input quorums provide inputs to a circuit of m gates in the same round.

We simulate an adversary who at the beginning of each simulation chooses uniformly at random without replacement a fixed number of nodes to control. Our adversary attempts to corrupt computations, where it drops messages or it changes the bits of messages sent between parties whenever possible. Aside from attempting to corrupt computations, the adversary performs no other attacks.

8.2 Results

The results of our experiments are shown in Figures 11, 12 and 13. Our results highlight two strengths of our self-healing algorithms (*self-healing*) when compared to algorithms without self-healing (*no-self-healing*). First, the message cost per *COMPUTE* decreases as the total number of calls to *COMPUTE* increases. Second, for a fixed number of calls to *COMPUTE*, the message cost per *COMPUTE* decreases as the total number of bad parties decreases. In particular, when there are no bad nodes, *self-healing* has dramatically less message cost than *no-self-healing*.

Expected number of messages

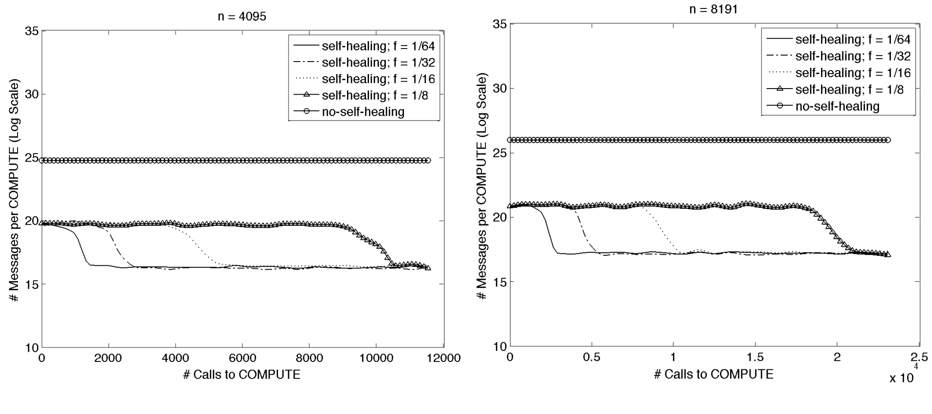

Fig. 11. Number of messages per call to *COMPUTE* versus # calls to *COMPUTE*, for $n = 4{,}095$ and $n = 8{,}191$.

Figure 11 shows that before all bad parties are marked or all selected leaders are good: 1) the expected number of messages per call to *COMPUTE* decreases as the total number of calls to *COMPUTE* increases; and 2) for a fixed number of calls to *COMPUTE*, the expected number of messages per call to *COMPUTE* decreases as f decreases.

Table 1 shows that when all selected leaders are good, *self-healing* has dramatically less expected number of messages per call to *COMPUTE* than *no-self-healing*.

n	*self-healing*	*no-self-healing*
4,095	84,072	28,297,456
8,191	153,744	66,435,642

Table 1. Expected number of messages per call to *COMPUTE* in *self-healing* and in *no-self-healing* for $n = 4{,}095$ and $n = 8{,}191$.

Expected latency

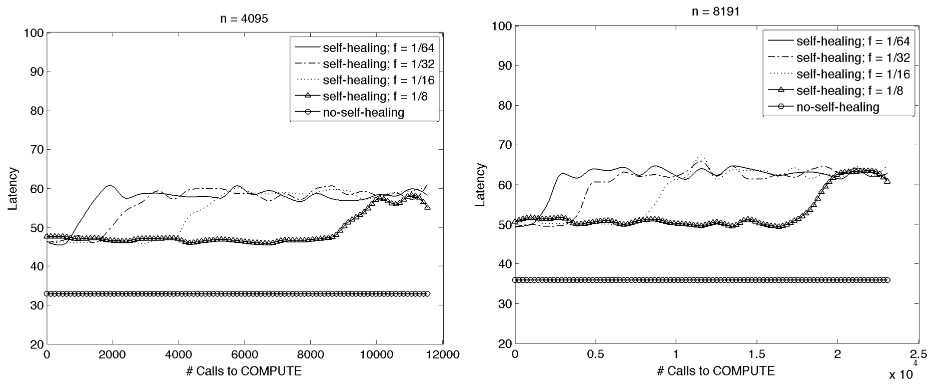

Fig. 12. Latency per call to *COMPUTE* versus # calls to *COMPUTE*, for $n = 4{,}095$ and $n = 8{,}191$.

Figures 12 shows that the latency of *self-healing* is less than the latency of *no-self-healing* due to the expected latency of *CHECK*. Note that in any call to *CHECK*, if a corruption is detected at any round, then *CHECK* is terminated in this round and *RECOVER* is called. This figure shows that when all bad parties are marked (no more corruptions occur), each call to *CHECK* will run all $O(\log^* n)$ rounds.

Table 2 shows that for $n = 4{,}095$ and $n = 8{,}191$, *self-healing* has latency that is at most twofold the latency of *no-self-healing*.

n	*self-healing*	*no-self-healing*
4,095	58	33
8,191	62	36

Table 2. Expected latency per call to *COMPUTE* in *self-healing* and in *no-self-healing* for $n =$ 4,095 and $n =$ 8,191.

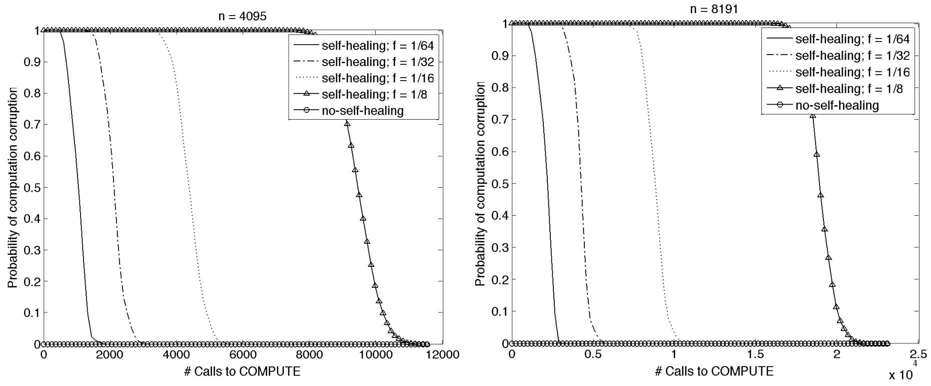

Fig. 13. The probability that the computation is corrupted versus # calls to *COMPUTE*, for $n =$ 4,095 and $n =$ 8,191.

Probability of computation corrupted

Figure 13 shows that the probability of computation corrupted per a call to *COMPUTE* decreases as the total number of calls to *COMPUTE* increases. Also, for a fixed number of calls to *COMPUTE*, this probability decreases as the total number of bad nodes decreases.

Even though our self-healing algorithm has a probability of corrupted computation that declines over the number of calls to *COMPUTE*, it always sends fewer messages than the naive algorithm.

Expected total number of corruptions

In Figure 13, for each network given the number of parties and the fraction of bad parties, if we integrate the corresponding curve, then we obtain $\Sigma(n)$, which is the experimental total number of corruptions occurred in a network of n parties.

f	$\Sigma(4{,}095)$	$\sigma(4{,}095)$	$\Sigma(8{,}191)$	$\sigma(8{,}191)$
1/64	1,024	2,116	2,065	4,232
1/32	2,102	4,388	4,209	8,777
1/16	4,367	9,557	8,771	19,114
1/8	9,440	24,576	18,920	49,152

Table 3. Expected total number of corruptions for $n =$ 4,095 and $n =$ 8,191.

Now we calculate the theoretical total number of corruptions, $\sigma(n)$, given the parameters of our experiments. Then, we compare between $\Sigma(n)$ and $\sigma(n)$, for different fractions of bad parties, f. Recall that p is the probability of selecting an unmarked bad party u.a.r. in a quorum. We know that at least $1/2$-fraction of parties is unmarked in any quorum at any time. Thus, $p \leq 2t/n = 2f$. By Lemma 9, the total number of calls to *RECOVER* is at most $\left(\frac{1-p}{1-2p}\right) t$. Note that in our experiments, *CHECK* is triggered with probability $1/(\log^* n)^2$, and it detects corruptions with probability at least $1/2$. Therefore, the theoretical total number of corruptions is at most

$$\sigma(n) = 2\left(\frac{1-2f}{1-4f}\right) t(\log^* n)^2.$$

Table 3 shows a comparison between the theoretical and experimental results of the expected total number of corruptions. It shows that $\Sigma(n) \leq \sigma(n)$ for $n = \{4095, 8191\}$ and $f = \{\frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}\}$.

9 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 close to 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.

Several open problems remain. 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?

In *CHECK*, we provide an array of random integers in each round to select parties uniformly at random in order to participate for detecting corruptions. Each array has $O(m \log n \log\log n)$ bits. If the input message has $O(b)$ bits, then the communication complexity per call to *COMPUTE* is $O((m + n\log n)(m \log n \log\log n + b))$ and the communication complexity of the naive algorithm is $O((m + n)\log^2 n \cdot b)$. Can we reduce the number of bits required to represent these arrays to $O(m \log\log n)$ in order to improve the communication complexity of our algorithms?

Moreover, in *INVESTIGATE*, each party that has participated in *COMPUTE* broadcasts a message of size $O((m \log n \log\log n + b)(\log^* m)^2)$ bits, how can we optimize the message size?

We allow the inputs of parties to be revealed. Can we modify our algorithms to maintain the privacy of these inputs? If we could, we would have a self-healing algorithm for reliable multiparty computation.

Also, we assume partially synchronous communication, which is crucial for our *CHECK* algorithm to detect computation corruptions over rounds. Can we extend this algorithm to fit for asynchronous computations?

Finally, we assume the presence of a static adversary. Can we modify our algorithms to self-heal computation in the presence of an adaptive adversary? Can we also deal with churn?

References

1. Fiat, A., Saia, J.: Censorship resistant peer-to-peer networks. Theory of Computing **3**(1) (2007) 1–23
2. Saad, G., Saia, J.: Self-healing computation. SSS'14 (2014) 195–210
3. Hildrum, K., Kubiatowicz, J.: Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In: Distributed Computing. Volume 2848. (2003) 321–336
4. Naor, M., Wieder, U.: A simple fault tolerant distributed hash table. IPTPS'03 (2003) 88–97
5. Scheideler, C.: How to spread adversarial nodes? rotate! STOC'05 (2005) 704–713
6. Fiat, A., Saia, J., Young, M.: Making chord robust to Byzantine attacks. ESA'05 (2005) 803–814
7. Awerbuch, B., Scheideler, C.: Towards a scalable and robust DHT. Theory of Computing Systems **45**(2) (2009) 234–260
8. King, V., Lonargan, S., Saia, J., Trehan, A.: Load balanced scalable Byzantine agreement through quorum building, with full information. ICDCN'11 (2011) 203–214
9. Frisanco, T.: Optimal spare capacity design for various protection switching methods in ATM networks. Volume 1 of ICC'97. (1997) 293–298
10. 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
11. Murakami, K., Kim, H.S.: Comparative study on restoration schemes of survivable ATM networks. Volume 1 of INFOCOM'97. (1997) 345–352
12. 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
13. 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
14. Boman, I., Saia, J., Abdallah, C., Schamiloglu, E.: Brief announcement: Self-healing algorithms for reconfigurable networks. Volume 4280 of SSS'06. (2006) 563–565
15. Saia, J., Trehan, A.: Picking up the pieces: Self-healing in reconfigurable networks. IPDPS'08 (2008) 1–12
16. Hayes, T., Rustagi, N., Saia, J., Trehan, A.: The forgiving tree: a self-healing distributed data structure. PODC'08 (2008) 203–212
17. 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
18. Pandurangan, G., Trehan, A.: Xheal: localized self-healing using expanders. Distributed Computing **27**(1) (2014) 39–54
19. 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
20. Knockel, J., Saad, G., Saia, J.: Self-healing of Byzantine faults. SSS'13 (2013) 98–112
21. Yao, A.C.: Protocols for secure computations. SFCS'82 (1982) 160–164
22. Beaver, D.: Efficient multiparty protocols using circuit randomization. CRYPTO'91. (1992) 420–432
23. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. STOC'88 (1988) 1–10
24. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. STOC'89 (1989) 73–85
25. Asharov, G., Lindell, Y.: A full proof of the BGW protocol for perfectly-secure multiparty computation. Electronic Colloquium on Computational Complexity (ECCC) **18** (2011) 36
26. Prabhakaran, M., Sahai, A.: Secure Multi-Party Computation. Volume 10. IOS Press (2013)

27. Kate, A., Goldberg, I.: Distributed key generation for the internet. ICDCS'09 (2009) 119–128
28. Moore, C., Mertens, S.: The Nature of Computation. Oxford University Press, Inc., New York, NY, USA (2011)
29. Van Der Hofstad, R.: Random graphs and complex networks. Available on http://www. win. tue. nl/rhofstad/NotesRGCN. pdf (2009)
30. Janson, S., Luczak, T., Rucinski, A.: Random Graphs. Wiley Series in Discrete Mathematics and Optimization. Wiley (2011)
31. Hadar, J., Russell, W.R.: Rules for Ordering Uncertain Prospects. American Economic Review **59**(1) (1969) 25–34
32. Bawa, V.S.: Optimal rules for ordering uncertain prospects. Journal of Financial Economics **2**(1) (1975) 95–121
33. Klenke, A., Mattner, L.: Stochastic ordering of classical discrete distributions. Advances in Applied Probability **42**(2) (2010) 392 – 410