Breaking the $O(n^2)$ Bit Barrier: Scalable Byzantine Agreement with an Adaptive Adversary

Valerie King*
Department of Computer Science, University of Victoria
P.O. Box 3055; Victoria, BC, Canada V8W 3P6
val@cs.uvic.ca

Jared Saia†
Department of Computer Science, University of New Mexico,
Albuquerque, NM 87131-1386
saia@cs.unm.edu

ABSTRACT

We describe an algorithm for Byzantine agreement that is scalable in the sense that each processor sends only $\tilde{O}(\sqrt{n})$ bits, where n is the total number of processors. Our algorithm succeeds with high probability against an *adaptive adversary*, which can take over processors at any time during the protocol, up to the point of taking over arbitrarily close to a 1/3 fraction. We assume synchronous communication but a *rushing* adversary. Moreover, our algorithm works in the presence of flooding: processors controlled by the adversary can send out any number of messages. We assume the existence of private channels between all pairs of processors but make no other cryptographic assumptions. Finally, our algorithm has latency that is polylogarithmic in n. To the best of our knowledge, ours is the first algorithm to solve Byzantine agreement against an adaptive adversary, while requiring $o(n^2)$ total bits of communication.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems*

General Terms

Theory

Keywords

Byzantine agreement, consensus, Samplers, Peer-to-peer, secret-sharing, Monte Carlo Algorithms, distributed computing

*This research was supported by an NSERC grant.

†This research was partially supported by NSF CAREER Award 0644058, NSF CCR-0313160, and AFOSR MURI grant FA9550-07-1-0532.

PODC'10, July 25–28, 2010, Zurich, Switzerland.

1. INTRODUCTION

Recent years have seen rapid growth in networks that are characterized by large sizes and little admission control. Such networks are open to attacks by malicious users, who may subvert the network for their own gain. To address this problem, the research community has been recently revisiting techniques for dealing with nodes under the control of a malicious adversary [21, 8, 9, 5].

The Byzantine agreement problem, defined in 1982, is the *sine qua non* of handling malicious nodes. With a solution to Byzantine agreement, it is possible to create a network that is reliable, even when its components are not. Without a solution, a compromised network cannot perform even the most basic computations reliably. A testament to the continued importance of the problem is its appearance in modern domains such as sensor networks [26]; mediation in game theory [1, 2]; grid computing [5]; peer-to-peer networks [25]; and cloud computing [27]. However, despite decades of work and thousands of papers, we still have no practical solution to Byzantine agreement for large networks. One impediment to practicality is suggested by the following quote from a recent systems papers (see also [6, 22, 4, 3]): *"Eventually batching cannot compensate for the quadratic number of messages [of Practical Byzantine Fault Tolerance (PBFT)]"* [10]

In this paper, we describe an algorithm for Byzantine agreement with only $\tilde{O}(n^{1/2})$ bit communication per processor overhead. Our techniques also lead to solutions with $\tilde{O}(n^{1/2})$ bit complexity for generating a distributed sequential version of a bit-fixing random source which generates a polylogarithmic length string, most of which are global coinflips generated uniformly and independently at random and agreed upon by all the good processors. Our protocols are polylogarithmic in time and, succeed with high probability.[1]

We overcome the lower bound of [11] by allowing for a small probability of error. This is necessary since this lower bound also implies that any randomized algorithm which always uses no more than $o(n^2)$ messages must necessarily err with positive probability, since the adversary can guess the random coinflips and achieve the lower bound if the guess is correct.

1.1 Model and Problem Definition

We assume a fully connected network of n processors $p_1, p_2, ..., p_n$ whose ID's are common knowledge. Each pro-

[1]That is probability $1 - 1/n^k$ for any fixed k

cessor has a private random coin. We assume that all communication channels are *private* and that whenever a processor sends a message directly to another, the identity of the sender is known to the recipient, but we otherwise make no cryptographic assumptions. We assume an *adaptive* adversary. That is, the adversary can take over processors at any point during the protocol, up to a total of $1/3 - \epsilon$ fraction of the processors for any positive constant ϵ. When an adversary takes over a processor, it learns the processor's state. The adversary is malicious: it chooses the input bits of every processor, bad processors can engage in any kind of deviations from the protocol, including false messages and collusion, or crash failures, while the remaining processors are good and follow the protocol. Bad processors can send *any* number of messages.

We assume a synchronous model of communication. In particular, we assume there is a known upper bound on the transit time of any message and communication proceeds in rounds determined by this transit time. The time complexity of our protocols are given in the number of rounds. However, we assume a *rushing* adversary which receives all messages sent by good processors to bad processors before sending out its own messages.

In the *Byzantine agreement* problem, each processor begins with either a 0 or 1. An execution of a protocol is *successful* if all processors terminate and, upon termination, agree on a bit held by at least one good processor at the start.

We define a distributed, sequential variant of a bit-fixing (random) source, see [7], which we call an (s, t) *random source* or *random source*, for short. This is a distributed protocol which generates a stream of *words* $w_1, ..., w_s$ such that t are generated uniformly and independently at random, and are agreed upon by all good processors. An adversary chooses the indices of these t words and the values received by the processors of each remaining word (these values may not be agreed upon). In fixing the value(s) of a word at index j the adversary knows the values only of the words which appear in positions $i < j$. We also consider a non-sequential version in which the sequence is generated in parallel and the adversary knows all the random words before fixing its words. An example is a simple $(n, 2n/3)$ $O(n)$ time protocol, $O(wn)$ bit complexity per processor for words of length w: For $i = 1, .., n$, each p_i randomly picks a word and sends it to all other processors. The challenge here is to reduce the length of the stream and the communication cost per processor.

1.2 Results

We use the phrase *with high probability* (*w.h.p.*) to mean that an event happens with probability at least $1 - 1/n^c$ for every constant c and sufficiently large n. For readability, we treat $\log n$ as an integer throughout.

In all of our results, n is the number of processors in a synchronous message passing model with an adaptive, rushing adversary that controls less than $1/3 - \epsilon$ fraction of processors, for any positive constant ϵ We have three main results. The first result makes use of the second and third ones, but these latter two results may be of independent interest. First, we show:

THEOREM 1. [*BA*] *There exists a protocol which w.h.p. computes Byzantine agreement, runs in polylogarithmic time, and uses* $\tilde{O}(n^{1/2})$ *bits of communication per processor.*

Our second result concerns *almost-everywhere* ("a.e.") Byzantine agreement and, *almost-everywhere* ("a.e.") random source where a $(1 - 1/\log n)$ fraction of the good processors come to agreement on a good processor's input bit, or the random words, resp.

THEOREM 2. [ALMOST EVERYWHERE BYZANTINE AGREEMENT] *For any* $\epsilon > 0$*, there exists a protocol that w.h.p. computes a.e. Byzantine agreement; uses* $\tilde{O}(n^{4/\epsilon})$ *bits of communication per processor; and runs in time* $O((\log^{4+\epsilon} n/ \log\log n)$*. In addition, this protocol can be used as an a.e.* $(s, 2s/3)$ *random source for an additional cost of* $O(\log n/ \log\log n)$ *time and* $\tilde{O}(n^{4/\epsilon})$ *bits of communication per bit of the stream, for* $s = \theta(\log^{3+\epsilon} n)$*.*

Our third result concerns going from a.e. Byzantine agreement to Byzantine agreement. It makes use of the a.e. (s, t) random source in Lemma 2. We actually prove a result below that is stronger than what is necessary to establish Theorem 1.

THEOREM 3. *Assume* $n/2 + \epsilon n$ *good processors agree on a message* M *and have access to an a.e.* (s, t) *random source, where* $t > c \log n$*, which generates* s *words of length* $(1/2) \lg_2 n$ *in* $O(f(s))$ *time and* $g(s)$ *bits of communication per processor. Then there is a protocol that ensures with probability* $1 - 1/n^c$ *that all good processors output* M *and* n*. Moreover this protocol runs in* $O(s + f(s))$ *time and uses* $\tilde{O}(sn^{1/2} + g(s))$ *bits of communication per processor.*

1.3 Techniques

Our protocol uses a sparse network construction and tournament tree similar to the network and tournament network in [19]. This past result gives a bandwidth efficient a.e. Byzantine agreement algorithm for a *non-adaptive adversary*, which must take over all its processors at the start of the algorithm. The basic idea of the algorithm from [19] is that processors compete in local elections in a tournament network, where the winners advance to the next highest level, until finally a small set is elected that is representative in the sense that the fraction of bad processors in this set is not much more than the fraction of bad processors in the general population.

This approach is *prima facie* impossible with an adaptive adversary, which can simply wait until a small set is elected and then can take over all processors in that set. To avoid this problem, we make use of two novel techniques. First, instead of electing processors, we elect *arrays* of random numbers, each generated initially by a processor. Second, we use secret sharing on the contents of the arrays to make sure that 1) the arrays are split among an increasingly larger numbers of processors as the array is elected higher up in the tournament. As the nodes higher in the tree have more processors (and the memories of previously held shares are erased), the adversary must corrupt an increasingly larger number of processors to learn the secret; 2) the secrets in the arrays cannot be reconstructed except at the appropriate time in the protocol. Critical to our approach is the need to iteratively reapply secret sharing on shares of secrets that were computed previously, in order to increase the number of processors the adversary must corrupt as elections occur higher up in the tournament.

Another contribution of this paper is a method *a.e.BA-with-random-source* for computing a.e. Byzantine Agreement

given an a.e. distributed random source. In [19], each election was run by a small group of participants using Feige's bin selection protocol [12]. Adapting Feige's bin selection protocol from an atomic broadcast model to a message-passing model requires a Byzantine agreement protocol. In [19], this was run by the small group of election participants. Because we are now faced with an adaptive adversary which can adaptively corrupt small groups, we instead need to implement BA on a much larger sets of processors. To achieve this, we use our algorithm, *a.e.BA-with-random-source*, which is an a.e. version of Rabin's algorithm [24] (which requires a global coin) run on a sparse network. To run an a.e. version of Rabin's algorithm in a node on a particular level, we supply coinflips using an *a.e.random-source* generated from the arrays which won on the previous tournament level.

Our final new technique is a simple but non-obvious protocol for going from a.e. Byzantine agreement and an a.e. random source to Byzantine agreement, with an adaptive adversary (Section 4). A past result [18] shows that it is possible to do this with a non-adaptive adversary, even without private channels. However, the technique presented in this paper for solving the problem with an adaptive adversary is significantly different from the approach from [18].

1.4 Map of the algorithms

Initially, processor p_i provides an array of random bits to leaf i of the tournament tree. There are five algorithms: *BA*, *a.e.BA*, *a.e.BA-with-random-source*, *a.e.random-source*, and *a.e.BA-to-BA*.

- *BA* (Section 5) solves Byzantine agreement and yields Theorem 1.
- *BA* calls *a.e.BA* (Section 3) which solves a.e. Byzantine agreement, yielding Theorem 2, and then calls *a.e.BA-to-BA* (Section 4) which yields Theorem 3.
- *a.e.BA-to-BA* uses *a.e.random-source* to go from a.e. Byzantine agreement to Byzantine agreement.
- *a.e.random-source* (Section 3.6) is a simple variant of *a.e.BA* which gives an a.e. sequential (s, t) random source.
- *a.e.BA* calls *a.e.BA-with-random-source* (Section 3.3) for each set of processors in each node of the tournament tree. This protocol produces a.e. Byzantine agreement for each set, given an a.e. random source for that set.
- Each call of *a.e.BA-with-random-source* uses *a.e.random-source* generated from the subtree of the node in question. This implementation of *a.e.random-source* is implicit in the description of *a.e.BA* in Section 3.

2. RELATED WORK

As mentioned previously, this paper builds on a main idea from [19] which gives a polylogarithmic time protocol with polylogarithmic bits of communication per processor for a. e. Byzantine agreement, leader election, and universe reduction in the synchronous full information message passing model with a *nonadaptive* rushing adversary. The result from [18] shows how the a.e. universe reduction problem (i.e., to select a small representative subset of processors known to almost all the processors) can be used to go from a.e. Byzantine agreement to Byzantine agreement with a *non-adaptive* adversary. Empirical results in [17] suggest that the algorithms from [19, 18] use less bandwidth in practice than other popular Byzantine agreement algorithms once the network size reaches about 1,000 nodes.

A.e. agreement in sparse networks has been studied since 1986. See [19, 20] for references. The problem of almost everywhere agreement for secure multiparty computation on a partially connected network was defined and solved in 2008 in [13], albeit with $\Omega(n^2)$ message cost.

In [20], the authors give a sparse network implementation of their protocols from [19]. It is easy to see that everywhere agreement is impossible in a sparse network where the number of faulty processors t is sufficient to surround a good processor. A protocol in which processors use $o(n)$ bits may seem as vulnerable to being isolated as in a sparse network, but the difference is that without access to private random bits, the adversary can't anticipate at the start of the protocol where communication will occur. In [14], it is shown that even with private channels, if a processor must pre-specify the set of processors it is willing to listen to at the start of a round, where its choice in each round can depend on the outcome of its random coin tosses, at least one processor must send $\Omega(n^{1/3})$ messages to compute Byzantine agreement with probability at least $1/2+1/\log n$. Hence the only hope for a protocol where every processor sends $o(n^{1/3})$ messages is to design outside this constraint. We note that *a.e.BA* falls within this restrictive model, but *a.e.BA-to-BA* does not, as the decision of whether a message is listened to (or acted upon) depends on how many messages carrying a certain value are received so far.

3. ALMOST EVERYWHERE BYZANTINE AGREEMENT

We now outline our main algorithm, *a.e.BA*. The processors are arranged into nodes in a q-ary tree. Each processor appears in polylogarithmic places in each level of the tree, so that a large fraction of nodes on each level must contain a majority of good processors. The levels of the tree are numbered from the leaf nodes (level 1) to the root (level ℓ^*). Nodes at higher levels contain more processors; the root contains all processors.

At the start, each processor p_i, generates an *array* of random bits, consisting of one *block* for each level of the network and secret shares each block with the processors in the i^{th} node on level 1.

Beginning with the lowest level of the tree, (the processors of) each node runs an election among r arrays from which a subset of w arrays are selected. To run this election at level ℓ, the ℓ block of each array supplies a random bin choice and random bits to run *a.e.BA-with-random-source* to agree on each bin choice of every competing array. The shares of the remaining blocks of arrays which remain in the competition are further subdivided into more shares and split among the larger number of processors in the parent node (and erased from the current processors' memories.) In this way, as an array becomes more important, an adversary cannot learn its secret value by taking over the smaller number of processors on lower levels which used to share the secret.

Random bits are revealed as needed by sending the iterated shares of secrets down paths to *all* the leaves of the

subtree rooted at the node where the election is occurring. In particular, at each level ℓ, ℓ-shares are collected to reconstruct $\ell-1$-shares. In the level 1 nodes, each processor sends the other processors its 1-share to reconstruct the original secret.

The winning arrays of a node's election compete in elections at the next higher level. At the root which contains all processors, there are a small number of arrays which can be used to run *a.e.BA* or *a.e.random-source*, to produce the output of the protocol.

The method of secret sharing and iterative secret sharing is described in Section 3.1. Networks and communication protocols are described in Section 3.2; the election routine is described in Section 3.4. The procedure for running *a.e.BA-with-random-source* is described in Section 3.3. The main procedure for *a.e.BA* is in 3.5. The extension of the almost everywhere Byzantine Agreement protocol to a solution for *a.e.random-source* is in Section 3.6. Finally the analysis and correctness proof can be found in Sections 3.7 and 3.8, respectively.

3.1 Secret sharing

We assume any secret sharing scheme which is a (n, τ) threshold scheme, for $\tau = n/3$. That is, each of n players are given shares of size proportional to the message M and τ shares are required to reconstruct M. Every message which is the size of M is consistent with any subset of τ or fewer shares, so no information as to the contents of M is gained about the secret from holding fewer than τ shares. Furthermore, we require that if a player possesses all the shares and less than $n/3$ are falsified by an adversary, the player can reconstruct the secret. See [23] for details on constructing such a scheme. We will make extensive use of the following definition.

DEFINITION 1. *secretShare(s): To share a sequence of secret words s with n_1 processes (including itself) of which $\tau_1 - 1$ may be corrupt, a processor (dealer) creates and distributes shares of each of the words using a (n_1, τ_1) secret sharing mechanism. Note that if a processor knows a share of a secret, it can treat that share as a secret. To share that share with n_2 processors of which at most $\tau_2 - 1$ processors are corrupt, it creates and distributes shares of the share using a (n_2, τ_2) mechanism and deletes its original share from memory. This can be iterated many times. We define a 1-share of a secret to be a share of a secret and an i-share of a secret to be a share of an $i-1$-share of a secret.*

To reveal a secret sequence s, all processors which receive a share of s from a dealer send this shares to a processor p which computes the secret. This also may be iterated to first reconstruct $i-1$ shares from i shares, etc., and eventually the secret sequence.

LEMMA 1. *If a secret is shared in this manner up to i iterations, then an adversary which possesses less than τ_1 1-shares of a secret and for $1 < j \leq i$, less than τ_j j-shares of each $j-1$-share that it does not possess, learns no information about the secret.*

PROOF. The proof is by induction. For level 1, it is true by definition of secret sharing. Suppose it is true up to i iterations.

Let v be any value. By induction, it is consistent with the known $\tau_j - 1$ shares on all levels $j \leq i$ and some assignment S_i of values to sets of unknown $n_i - \tau_i + 1$ i-shares. Then consider the shares of these shares that have been spread to level $i+1$. For each value of an i-share given by S_i, there is an assignment S_{i+1} of values to the unknown $n_{i+1} - \tau_{i+1} + 1$ shares consistent with that value. Hence knowing in addition the $\tau_{i+1} - 1$ $i+1$-shares of each i-share does not reveal any information about the secret. □

3.2 Network and Communication

We first describe the topology of the network and then the communications protoocols.

3.2.1 Samplers

Key to the construction of the network is the definition of an averaging sampler which was also used heavily in [16, 20]. We repeat the definition here for convenience. Our protocols rely on the use of averaging (or oblivious) samplers to determine the assignment of processors to nodes on each level and to determine the communication links between processors. Samplers are families of bipartite graphs which define subsets of elements such that all but a small number contain at most a fraction of "bad" elements close to the fraction of bad elements of the entire set. Intuitively, they are used in our algorithm in order to generate samples that do not have too many bad processors. We assume either a nonuniform model in which each processor has a copy of the required samplers for a given input size, or else that each processor initializes by constructing the required samplers in exponential time.

DEFINITION 2. *Let $[r]$ denote the set of integers $\{1, \ldots, r\}$, and $[s]^d$ the multisets of size d consisting of elements of $[s]$. Let $H : [r] \to [s]^d$ be a function assigning multisets of size d to integers. We define the size of the intersection of a multiset A and a set B to be the number of elements of A which are in B.*

Then we say H is a (θ, δ) sampler *if for every set $S \subset [s]$ at most a δ fraction of all inputs x have $\frac{|H(x) \cap S|}{d} > \frac{|S|}{s} + \theta$.*

The following lemma establishing the existence of samplers can be shown using the probabilistic method. For $s' \in [s]$, let $deg(s') = |r' \in [r] \mid s.t.\ s \in H(r')\}|$. A slight modification of Lemma 2 in [16] yields:

LEMMA 2. *For every $r, s, d, \theta, \delta > 0$ such that $2\log_2(e) \cdot d\theta^2\delta > s/r + 1 - \delta$, there exists a (θ, δ) sampler $H : [r] \to [s]^d$ and for all $s \in [s]$, $deg(s) = O((rd/s)\log n)$.*

Note that limiting the degree of s limits the number of subsets that any one element appears in.

For this paper we will use the term *sampler* to refer to a $(1/\log n, 1/\log n)$ sampler, where $d = O((s/r + 1)\log^3 n)$.

3.2.2 Network structure

Let P be the set of all n processors. The network is structured as a complete q-ary tree. The level 1 nodes (leaves) contain $k_1 = \log^3 n$ processors. Each node at height $\ell > 1$ contains $k_\ell = q^\ell k_1$ processors; there are $(n/k_\ell)\log^3 n$ nodes on level ℓ; and the root node at height $\ell^* = \log_q(n/k_1)$ contains all the processors. There are n leaves, each assigned to a different processor. The contents of each node on level ℓ is determined by a sampler where $[r]$ is the set of nodes, $[s] = P$ and $d = k_\ell$.

The edges in the network are of three types:

1. *Uplinks:* The *uplinks* from processors in a child node on level ℓ to processors in a parent node on level $\ell+1$ are determined by a sampler of degree $d = q\log^3 n$, $[r]$ is the set of processors in the child node and $[s]$ is the set of processors in the parent node. Let C, C' be child nodes of a node A. Then the mapping of processors in C, C' to $[r]$ (and A to $[s]$) determines a *correspondence* between the uplinks of C and C'.

2. $\ell - links$: The $\ell - links$ between processors in a node C at any level $\ell > 1$ to C's descendants at level 1 are determined by a sampler where $[r]$ is the set of processors in the node C and $[s]$ is C's level 1 descendants. Here, $r = q^\ell k_1$; $s = q^\ell$; $d = O(\log^3 n)$ and the maximum number of $\ell - links$ incident to a level 1 node is $O(k_1 \log^4 n)$.

3. *Links between processors in a node* are also determined by a sampler of polylogarithmic degree. These are described in *a.e.BA-with-random-source*.

Definition 3. *Call a node* good *if it contains at least a* $2/3 + \epsilon$ *fraction of good processors. Call it* bad *otherwise.*

From the properties of samplers, we have:

1. Less than a $1/\log n$ fraction of the nodes on any level are bad.

2. Less than a $1/\log n$ fraction of processors in every node whose uplinks are connected to fewer than a $2/3 + \epsilon - 1/\log n$ fraction of good processors, unless the parent or child, resp. is a bad node. We call such a set of uplinks for a processor *bad*.

3. If a level ℓ node C has less than a $1/2 - \epsilon$ fraction of bad level 1 descendants, then less than a $1/\log n$ fraction of processors in C are connected through $\ell - links$ to a majority of bad nodes on level 1.

3.2.3 Communication protocols

We use the following three subroutines for communication. Initially each processor p_i shares its secret with all the processors in the i^{th} node at level 1.

sendSecretUp(s): To send up a sequence s of secret words, a processor in a node uses *secretShare*(w) to send to each of its neighbors in its parent node (those connected by *uplinks*) a share of each word w of s. Then the processor erases s from its own memory.

sendDown(w, i): After a secret w has been passed up a path to a node C, the secret can be recovered by passing it down to the processors in the 1-nodes in the subtree. To send a secret word w down the tree, each processor in a node C on level i sends its i-shares of w *down* the uplinks it came from plus the corresponding uplinks from each of its other children. The processors on level $i-1$ receiving the i-shares use these shares to reconstruct $i-1$-shares of w. This is repeated for lower levels until all the 1-shares are reconstructed by the processors in all the 1-nodes in C's subtree. The processors in the 1-node each send each other all their shares and reconstruct the secrets received. Note that a processor may have received an i-share generated from more than one $i-1$ share because of the overlapping of sets (of uplinks) in the sampler.

sendOpen(w, ℓ) : This procedure is used by a node C on any level ℓ to learn a word w held by the set of level 1 nodes in C's subtree. Each processor in each level 1 node A sends w up the $\ell - links$ from A to a subset of processors in C. A processor in C receiving a version of w from each of the processors in a level 1 node takes a majority to determine the node A's version of w. Then it takes a majority over the values obtained from each of the level 1 nodes it is linked to.

3.2.4 Correctness of communications

Definition 4. *A good path up the tree is a path from leaf to root which has no nodes which become bad during the protocol.*

Lemma 3. *1. If sendSecretUp(s) is executed up a path in the tree and if the adversary learns a word of the secret s, there must be at least one bad node on that path.*

2. Assume that s is generated by a good processor and sendSecretUp(s) is executed up a good path in a tree to a node A on level ℓ, followed by sendDown(w, ℓ) where w is a word of s, and then sendOpen(w). Further assume there are at least a $1/2 + \epsilon$ fraction of nodes among A's descendants on level 1 which are good, and whose paths to A are good. Then a $1 - 1/\log n$ fraction of the good processors in A learn w.

Proof. Proof of (1): If *sendSecretUp*(s) is executed up a good path of length ℓ, then all secrets passed up uplinks incident to a 2/3 majority of good processors will remain secret, by Lemma 1.

We consider the effect of secrets passed up uplinks incident to less than 2/3 majority of good processors. Let us define a processor in a node as "bad" if it is bad or it is good and its share is learned by the adversary. We show by induction on the level number that when secrets are passed from a leaf to level ℓ for any level ℓ, the level i node on a good path up to level ℓ contains no more than $1/3 - \epsilon + 1/\log n$ fraction of processors which are "bad".

For the basis case, $i = \ell$, only the shares received by bad processors in the level ℓ node are learned by the adversary. Since the node is good, there are no more than a $1/3 - \epsilon$ fraction of "bad" processors.

We assume by induction that no more than $1/3 - \epsilon + 1/\log n$ processors in nodes on levels i through ℓ are "bad". We show it is true for level $i-1$. Since the uplinks are determined by a $(1/\log n, 1/\log n)$-sampler, when the processors in a node on level $i-1$ pass the secret shares up the uplinks, no more than a $1/\log n$ fraction of the uplink sets are incident to more than $1/3 - \epsilon + 2/\log n$ fraction of "bad" processors on level i. The adversary through these processors may thus learn a $1/\log n$ fraction of $i-1$-shares sent by processors in the level $i-1$ node in the path. Thus no more than a $1/\log n$ fraction of processors holding the $i-1$-shares are "bad" in addition to the bad processors in the level $i-1$ node, for a total fraction of $1/3 - \epsilon - 1/\log n$ "bad" processors. This completes the induction.

We have shown that no more than $1/3 - \epsilon + 1/\log n$ fraction of 1-shares will be learned by the adversary. Thus, the adversary cannot learn the secret, which completes the proof of part (1) of the lemma.

Proof of (2): The proof of part (1) shows that on each level of a good path, no more than a $1/\log n$ fraction of

good processors in the path have uplinks which are incident to less than a 2/3 fraction of good processors. Hence when the i-shares are returned down the uplinks they were sent, for all but $1/\log n$ processors, each receives enough shares to recover the $i-1$ share that it once had. The same is true for the processors in the other good paths of the subtree rooted at A. In particular, all but $1/\log n$ fraction of processors learn the secret in each of the leaves of the subtree rooted at A, for all leaves on good paths to A. If there are at least $1/2+\epsilon$ fraction of such leaves then by property (3) of the samplers, at least $1-1/\log n$ fraction of the processors in A will be connected by ℓ-links to a majority of such leaves. Thus when *sendOpen* is executed, $1-1/\log n$ fraction of processors in A will learn the secret correctly when they take the majority of values held by each leaf to which they are connected by ℓ-links, where that value is determined by a majority of processors in the leaf. □

3.3 a.e.BA with random source

We present the algorithm and otherwise leave out the proof of Theorem 4 for lack of space. We assume here that the fraction of bad processors is no more than $1/3+\epsilon$ for some fixed $\epsilon > 0$. We assume access to a sequential (s,t) *a.e.random-source*, where s and t are polylogarithmic, which outputs sequences of bits, one per call.

THEOREM 4. *[a.e.BA-with-random-source] Given a sequential* (s,t) a.e.random-source *which generates s bits in $f(s)$ time and $g(s)$ bit complexity per processor, and let C_1 and C_2 be any positive constants. Then there is a protocol which runs in time $O(f(s))$ with bit complexity $O(\log n + g(s))$ per processor, such that with probability at least $1-e^{-C_1 n}+1/2^t$, all but $C_2 n/\log n$ of the good processors commit to the same vote b, where b was the input of at least one good processor.*

The algorithm is an implementation of Rabin's global coin toss Byzantine agreement protocol except that the coin toss is a.e., rather than global. In addition, only some of the coin tosses are random, and broadcast is replaced by sampling a fraction of the other processors according to the edges of a certain type of $O(\log n)$ regular graph. This graph has the property that almost all of the samples contain a majority of processors whose value matches the value of the majority of the whole set.

Algorithm 1 *a.e.BA-with-random-source*

Set $vote \leftarrow b_i$; For s rounds do the following:

1. Send *vote* to all neighbors in G;
2. Collect votes from neighbors in G;
3. $maj \leftarrow$ majority bit among votes received;
4. $fraction \leftarrow$ fraction of votes received for maj;
5. $coin \leftarrow$ result of call to *a.e.random-source*;
6. If $fraction \geq (1-\epsilon_0)(2/3+\epsilon/2)$ then $vote \leftarrow maj$
7. else
   (a) If $coin =$ "heads", then $vote \leftarrow 1$, else $vote \leftarrow 0$;

At the end of all rounds, commit to *vote* as the output bit;

3.4 Arrays and the election subroutine

Here we describe the array of random bits generated by each processor. Each processor generates a sequence of blocks, each except the last to be used for the elections in the nodes along the path to the root are disgarded. The last block is only used to decide the final output bit, and contains only the bits needed to compute *a.e.BA-with-random-source*. To understand the blocks, we first describe Feige's election procedure.

The procedure is simple: each processor announces a bin number in some small range. The winners are the processors which choose the bin picked by the fewest. Here we describe Feige's election procedure [12], adapted to this context. We assume r candidates are competing in the election.

DEFINITION 5. *Let $numBins = r/(5c\log^3 n)$, and let a* word *consist of* $\log numBins$ *bits. In general, a block B is a sequence of bits, beginning with an initial word (bin choice) $B(0)$ followed by r words $B(1), B(2), .., B(r)$, each bit of which is used as coins in running* a.e.BA-with-random-source *on each bit of the bin choices for each of the r candidates. The input to an election is a set of r candidate blocks labelled $B_1, ..., B_r$. The output is a set of $r/numBins$ indices W. Let $w = |W| = 5c\log^3 n$.*

We now describe the election subroutine.

1. In parallel, for $i = 1, ..., r$, the processors run *a.e.BA-with-random-source* on the bin choice of each of the r candidate blocks. Round j of *a.e.BA-with-random-source* to determine i's bin choice is run using the i^{th} word of the j^{th} processor's block $B_j(i)$.
   Let $b_1, ..., b_r$ be the decided bin choices.
2. Let $min = \min\{i \mid \sum_j B_j(0) = i\}$.
   $W \leftarrow \{j \mid B_j(0) = min\}$.
   (If $|W| < r/numBins$ then W is augmented by adding the first $r/numBins - |W|$ indices that would otherwise be omitted.)

Then Feige's result for the atomic broadcast model holds (except agreement on the choices is a.e.):

LEMMA 4. *[12] Let S be the set of bin choices generated independently at random. Then even if the adversary sets the remaining bits after seeing the bin choices of S, with probability at least $1-2^{-\epsilon^2|S|/(3numBins)}$ there are at least $(1/numBins-\epsilon)|S|$ winners from S. In particular, if $|S| > 2/3r$ and $r/numBins > 5c\log^3 n$ then with probability $1-1/n^c$ the fraction of winners from S is at least $|S|/r - 1/\log n$. The winners are known to all but $1/\log n$ fraction of the processors.*

3.5 Main protocol for a.e. BA

The main protocol for *a.e.BA* is given as Algorithm 2. Figure 1, which we now describe, outlines the main ideas behind the algorithm. The left part of Figure 1 illustrates the algorithm when run on a 3-ary network tree. The processors are represented with the numbers 1 through 9 and the ovals represent the nodes of the network, where a link between a pair of nodes illustrates a parent child relationship. The numbers in the bottom part of each node are the processors contained in that nodes. Note that the size of these sets increase as we go up tree. Further note that each processor is contained in more than one node at a given level. The numbers in the top part of each node represent the processors whose blocks are candidates at that node. Note that the size of this set remains constant (3) as we go from level

2 to level 3. Further note that each processor is a candidate in at most one node at a given level.

The right part of Figure 1 illustrates communication in *a.e.BA* for the election subroutine occurring at a fixed node at level ℓ. The figure illustrates three main parts of the algorithm (time moves from left to right). First, on the left, is "Expose bin Choices". Here, the bin choices of the candidates at the level ℓ node are revealed in two parts: (1) hop-by-hop communication down the network using the *sendDown* protocol from level ℓ to level 1, at the end of which the nodes at level 1 know the relevant bin choices; and (2) direct communication up the network using the *sendOpen* protocol of these bin choices from the level 1 nodes to the level ℓ node.

The second part of the algorithm is "Agree Bin Choices". This involves running *a.e.BA-with-random-source* which uses "horizontal" communication among the processors in the node at level ℓ that are running *a.e.BA-with-random-source*. It requires the output of a sequential *a.e.random-source*. This is provided by communication within the subtree rooted at the node via hop-by-hop communication down the network, using *sendDown*, at the end of which the nodes at level 1 know the outcome of the next coin toss; and direct communication up the network, using *sendOpen*, of the coin toss outcome from the level 1 nodes to the level ℓ node.

The third and final part of the algorithm is "Send Shares Winners", where shares of the blocks of the winners at the level ℓ node are sent to the level $\ell + 1$ parent node, via use of the procedure *sendSecretUp*.

3.6 Modification to a.e. random-source

The protocol *a.e.BA* can be modified easily to solve $(s, 2s/3)$-*a.e.random-source* for s a sequence of wq words. Add one more block of the desired length to each processor's array at the start. At level ℓ^*, use sendDown and sendOpen to recover each word, one from each of the wq contestants. The time and bit complexity is given in Theorem 2.

3.7 Bit complexity and running time analysis

To optimize, we set the degree of the tree, q to $q = (\log n)^\delta$, $\delta > 4$. Recall from Section 3.4 that $w = O(\log^3 n)$. The proof of the following lemma is fairly straightforward and is omitted for lack of space. One not-so-straightforward point is that a processor in a parent node may receive uplinks from several different processors in a child node, and this will increase the load on the processor by a d_m factor with each increase in level of the processor, where d_m is the maximum number of such uplinks from one child node. d_m is limited by the construction of the sampler.

LEMMA 5. *For any $\delta > 0$, Almost Everywhere Byzantine Agreement protocol requires $\tilde{O}(n^{4/\delta})$ bits per processor and runs in time $O((\log n)^{4+\delta}/\log\log n))$.*

3.8 Proof of correctness

Call an array at node A *good* if it is generated by good processors and it is known by a $1 - O(1/\log n)$ fraction of good processors at node A.

An election is *good* at a node A if the processors can carry out *a.e.BA-with-random-source*, a $1 - 1/\log n$ fraction of good processors agree on the result, and the election winners are representative, i.e., the fraction of good arrays in the set of winners is no less than the fraction of good arrays among the contestants minus a $1 - 1/\log n$ fraction.

Algorithm 2 *a.e.BA*

1. **Generate Secrets and Send Up Shares:**
   For all i in parallel
   (a) Each processor p_i generates an array of ℓ blocks B_i and uses *secretshare* to share its array with the i^{th} level 1 node;
   (b) Each processor in the i^{th} level 1 node uses *sendSecretUp* to share its 1-share of B_i with its parent node and then erases its shares from memory.

2. **Elect Winners at Root Node:**
   Repeat for $\ell = 2$ to $\ell^* - 1$
   (a) **Expose bin choices:**
   For each processor in each node C on level ℓ:
   for $t = 1, ..., w$ and $i = 1, ..., q - 1$, let $B_{(i-1)w+t}$ be the t^{th} array sent up from child i. (If $\ell = 2$ then $w = 1$)
   $W \leftarrow B_1 \| B_2 \| ... \| B_r$
   Let F be the sequence of first blocks of the arrays of W, i.e., the i^{th} array of F is the first block of the i^{th} array of W. Let S be the sequence of the remaining blocks of each array of W.
   In parallel, for all candidates $i = 1, 2, .., r$
      i. $sendDown(F_i(1))$;
      ii. $sendOpen(F_i(1), \ell)$.
   (b) **Agree on bin choices:**
   If $\ell < \ell^*$ then for rounds $i = 1, ..., r$
      i. **Expose coin flips**: Generate r random words for the i^{th} round of *a.e.BA-with-random-source* to decide each of r bin choices.
      In parallel, for all contestants $j = 1, .., r$
         A. $sendDown(F_i(j))$; upon receiving all 1-shares, level 1 processors compute the secret bits $F_i(j)$);
         B. $sendOpen(F_i(j), \ell)$.
      ii. Run the i^{th} round of *a.e.BA-with-random-source* in parallel to decide the bin choice of all contestants.
   (c) **Send Shares of Winners Up to Next Level**:
   Let W be the winners of the election decided from the previous step (the lightest bin). Let S' be the subsequence of S from W; All processors in a node at level ℓ use $sendSecretUp(S')$ to send S' to its parent node and erase S' from memory.

3. **Root Node Runs BA using Arrays of Winners**:
   All processes in the single node on level ℓ^* run *a.e.BA-with-random-source* once using their initial inputs as inputs to the protocol (instead of bin choices) and the remaining block of each contestant. (Note that only one bit of this block is needed.)
   For rounds $i = 1, 2, ..., qw$ of *a.e.BA-with-random-source*,
   (a) $sendDown(F_i(1), i)$;
   (b) $sendOpen(F_i(1), \ell)$.
   (c) Use $F_i(1)$ as the coin for this round of *a.e.BA-with-random-source*.

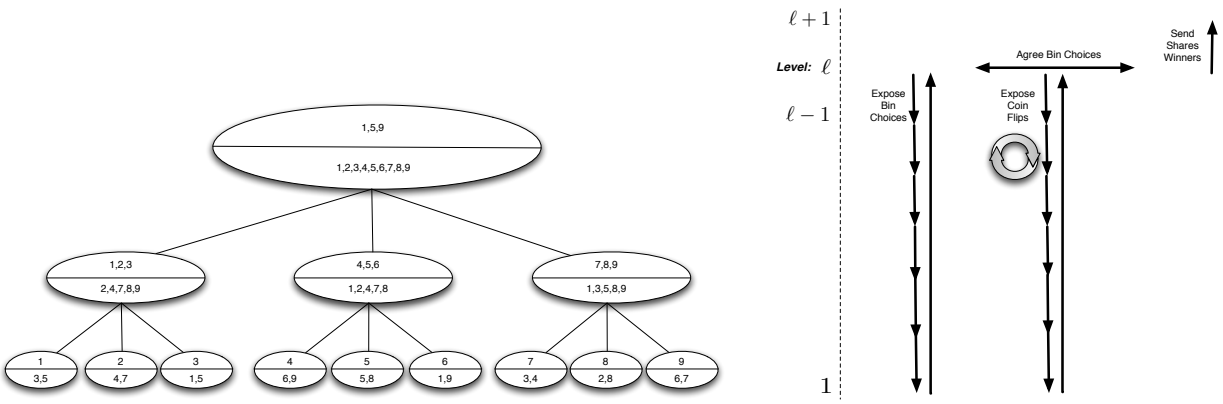

Figure 1: Left: Example run of Algorithm 1 on a small tree; Right: Communication in different phases of Algorithm 1 for a fixed level ℓ.

Recall from Lemma 4 there must be at least $5c\log^3 n$ contestants participating from S to ensure w.h.p. that the election winners are representative of S. Recall (Theorem 4) that *a.e.BA-with-random-source* succeeds w.h.p., if $2/3+\epsilon$ fraction of processors in the node are good and bits can be generated so that at least $c\log n$ are random and for each of these, there is a fraction of $1-1/\log n$ fraction of good processors which agree on it. Thus, an election is good, w.h.p, if the following conditions hold (1) A is a good node; (2) at least $5c\log^3 n$ contestants are candidate arrays with good paths from their assigned level 1 node to A so that that secrets are correctly transmitted up the tree without the adversary learning the secret until it it released; and (3) there must be a $1/2+\epsilon$ fraction of level 1 nodes in A's subtree which have good paths to A, so that, by properties of the network, as described in Section 3.2.2, a $1-1/\log n$ fraction of good processors in A learn the random bin selections and random bits of the good arrays that are competing.

We now lower bound the fraction of arrays that remain that are good.

LEMMA 6. *In the* a.e.BA *algorithm, at least a* $2/3-5\ell/\log n$ *fraction of winning arrays are good on every level* ℓ*. In particular, the protocol can be used to generate a sequence of random words, of length* $r=wq$ *(one from each array at the root of the tournament tree) of which a* $2/3+\epsilon-5/\log\log n$ *fraction are random and known to* $1-1/\log n$ *fraction of good processors.*

PROOF. With high probability, each good election causes an increase of no more than a $1/\log n$ in the fraction of arrays that are bad. We now consider the number of good arrays discarded because of bad elections.

A violation of Condition (1) (the election node is bad) effects at most a $1/\log n$ fraction of arrays per level since at most a $1/\log n$ fraction of nodes are bad per level.

A violation of Condition (2) ($< c\log^3 n$ good arrays participating) cannot happen too often. Let r be the number of contestants in an election at level ℓ. If there are f bad arrays overall on level ℓ then the total number of such lop-sided elections is less than $f/(r-c\log^3 n)$. In these lop-sided elections, the fraction of good arrays is at most $(c\log^3 n)/r$. Since the number of candidates $r=wq>\log^7 n$, the fraction of good arrays lost this way is less than $1/\log^4 n$.

A violation of Condition (3) ($< 1/2+\epsilon$ fraction of level 1 nodes have good paths to the current node) also cannot happen to often. In particular, a $1/\log n$ fraction of bad nodes can be responsible for making a $2/\log n$ fraction of elections bad in every level at above it , by making bad half the paths of those elections, thus eliminating an additional $2/\log n$ fraction of arrays that are good. Note that a bad election at a good node does not make additional paths bad, as information and secrets can still be passed through a good node that held a bad election.

From the possible violation of these three conditions, we lose no more than a $4/\log n$ fraction of arrays that are good. Adding in good arrays lost from good elections, we lose no more than a total of $5/\log n$ fraction of arrays that are good at any level.

Initially a $2/3+\epsilon$ fraction of the arrays are good. As we have shown, at any given level, the fraction of arrays that are good decreases by at most $5/\log n$. Thus, the fraction of arrays that are good at level ℓ^* is at least $2/3+\epsilon-5\ell^*/\log n = 2/3+\epsilon-5/\log\log n$.

Hence there are wq arrays at the top of the election tree of which at least a $2/3$ fraction were generated by good processors, uniformly and independently at random. For each array, there is a block which is secret shared among the processors of the root node, so that for all but $1/\log n$ fraction of the ℓ^*-shares, at least a $2/3$ fraction of ℓ-shares are held by good processors. Sequentially, the protocol exposes a word from each of the arrays. By Lemma 3, each word generated by a good processor is learned by a.e. processor in the root (which contains all the processors). □

The sequential generation of the sequence of words in Lemma 6 ($wq, 2wq/3$) *a.e.random-source* which can be used as input to *a.e.BA-with-random-source*. The correctness of *a.e.BA* follows from Theorem 4. To generate *a.e.random-source* of desired wordsize, it suffices to include a word of that wordsize in each array.

4. A.E. BYZANTINE AGREEMENT TO BYZANTINE AGREEMENT

We call a processor *knowledgeable* if it is good and agrees on a message m. Otherwise, if it is good, it is *confused*.

In this section, we assume that $(1/2+\epsilon)n$, of the processors are knowledgeable. We use the sequential (s,t) *a.e.random-source* which generates a sequence of bits, with $t \geq c \log n$, and s is polylogarithmic, described in Section 3.6. We assume private channels. Here is the protocol. The constants a, c, and ϵ in this protocol will be specified in the proofs of correctness.

Algorithm 3 *a.e.BA-to-BA*

Repeat s times:

1. Each processor p does the following in parallel: Randomly pick a set of $a\sqrt{n}\log n$ processors *without replacement*; for each of these processors j, randomly pick $i \in [1,\ldots,\sqrt{n}]$ (with replacement) and send a *request label* i to processor j.
2. Almost all good processors agree on the next number k in $[1,...,\sqrt{n}]$ generated by *a.e.random-source*.
3. For each processor p, if p receives request label i from q and $i = k$ then if p has not received more than $\sqrt{n}\log n$ such messages (it is not *overloaded*), p returns a message to q.
4. Let k_i be the number of messages returned to p by processors sent the request label i. Let i_{max} be an i such that $k_i \geq k_j$ for all j. If the same message m is returned by $(1/2+3\epsilon/8)a\log n$ processors which were sent the request label i_{max} then p decides m.

4.1 Proof of correctness

LEMMA 7. *Assume at the start of the protocol $n/2+\epsilon n$ good processors agree on a message m and can generate a random bit. Let c be any positive constant. Then after a single execution of the loop:*

1. *With probability $4/(\epsilon\log n) - 1/n^c$, this protocol results in agreement on m.*
2. *With probability $1-1/n^c$, every processor either agrees on m or is undecided.*

To prove Lemma 7 we first prove two other lemmas.

LEMMA 8. *Suppose there are $(1/2+\epsilon)n$ knowledgeable processors. W.h.p., for any one loop, for every processor p and every request label i, at least $A = (1/2+\epsilon/2)a\log n$ processors which are sent i by p are knowledgeable and fewer than $B = (1/2-\epsilon/2)a\log n$ processors which are sent i by p are corrupt or confused.*

PROOF. Since there are private channels, the adversary does not know p's requests other than those sent to bad processors. Hence the choice of the set of processors which are not knowledgeable is independent of the queries, and each event consisting of a processor querying a knowledgeable processor is an independent random variable.

Let X be the number of knowledgeable processors sent a value i by processor p. X is a hypergeometric distribution and $E(X) = a\log n(1/2+\epsilon)$. By concentration bounds on the hypergeometric distribution (see e.g. [15]), we know that: $Pr(X \leq E(X) - \epsilon a\log n) \leq e^{-(\epsilon a\log n)^2/2E(X)}$. This is less than n^{-2c} for $a = 2c(1+2\epsilon)(\ln 2)/\epsilon)$ Taking a union bound over all i and processors p, for all X, $Pr(X \leq E(X) - \epsilon a\log n)$ is less than $\sqrt{n}(n)n^{-2c} < 1/n^{-c}$. The second part of Lemma 8 is shown similarly. □

Lemma 8 immediately implies statement (2) of Lemma 7.

We now show Lemma 7 (1). A knowledgeable processor p which is sent $i = k$ will respond unless overloaded. Each processor can receive no more than $n-1$ requests, or the sender is evidently corrupt. Then there can be no more than $\sqrt{n}/\log n$ values of i for which there are more than $\sqrt{n}\log n$ requests labelled i. Then we claim:

LEMMA 9. *The probability that more than $\epsilon n/4$ knowledgeable processors are overloaded in any one loop of the algorithm is less than $4/(\epsilon\log n)$.*

PROOF. We call a value i for a processor overloaded if $\sqrt{n}\log n$ request labels equal i. A processor is only overloaded if $k = i$ and i is overloaded. Since k is randomly chosen, each processor has at most a $1/\log n$ chance of being overloaded. Let X be a random variable giving the number of overloaded knowledgeable processors and y be the number of knowledgeable processors. Then $E[X] = y/logn$. Using Markov's Inequality, $Pr[X \geq y(\epsilon/4)] < (y/\log n)/(y\epsilon/4) = 4/(\epsilon\log n)$. □

LEMMA 10. *Repeating the protocol $(c'/3)\epsilon\ln n$ times, the probability that all processors agree on m and no processor outputs a different message is $1-1/n^{c'}$.*

PROOF. Similar to the argument above, because the adversary does not know the requests and request labels of the requests sent to knowledgeable processors, the event of choosing knowledgeable processors which are not overloaded are independent random variables and Chernoff bounds apply. With probability $4/(\epsilon\log n)$, there are $(1/2+3\epsilon/4n)$ knowledgeable processors which are not overloaded. Setting ϵ to $3\epsilon/4$ in Lemma 8, we have w.h.p., for every processor and request label i that at least $A = (1/2+3\epsilon/8)a\log n$ processors sent i by p are knowledgeable and fewer than $B = (1/2-3\epsilon/8)a\log n$ processors sent i by p are corrupt or confused. Therefore, with probability $4/(\epsilon\log n) - 1/n^{c'}$, one loop of this protocol results in agreement on m. As each repetitions of the loop is independent, the probability that they all fail is the product of their individual failure probabilities, implying the lemma statement. □

5. BYZANTINE AGREEMENT (*BA*)

Algorithm 4 *BA*

1. Run *a.e.BA* to come to almost everywhere consensus on a bit b.
2. Run *a.e.BA-to-BA* to ensure that all processors output b.

The execution of both *a.e.BA* and *a.e.BA-to-BA* take $\tilde{O}(\sqrt{n})$ bits per processor, while both of these algorithms have polylogarithmic latency.

6. CONCLUSION

We have described an algorithm that solves the Byzantine agreement problem with each processor sending only $\tilde{O}(\sqrt{n})$ bits. Our algorithm succeeds against an adaptive, rushing adversary in the synchronous communication model. It assumes private communication channels but makes no other cryptographic assumptions. Our algorithm succeeds with

high probability and has latency that is polylogarithmic in n. Several important problems remain including the following: Can we use $o(\sqrt{n})$ bits per processor, or alternatively prove that $\Omega(\sqrt{n})$ bits are necessary for agreement against an adaptive adversary? Can we adapt our results to the asynchronous communication model? Can we use the ideas in this paper to perform scalable, secure multi-party computation for other functions? Finally, can the techniques in this paper be used to create a practical Byzantine agreement algorithm for real-world, large networks?

7. REFERENCES

[1] Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halper. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Principles of Distributed Computing(PODC)*, 2006.

[2] Ittai Abraham, Danny Dolev, and Joe Halper. Lower bounds on implementing robust and resilient mediators. In *IACR Theory of Cryptography Conference(TCC)*, 2008.

[3] A. Agbaria and R. Friedman. Overcoming Byzantine Failures Using Checkpointing. *University of Illinois at Urbana-Champaign Coordinated Science Laboratory technical report no. UILU-ENG-03-2228 (CRHC-03-14)*, 2003.

[4] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *Proc. Int. Conf. on Dependable Systems and Networks*, pages 105–114. Citeseer, 2006.

[5] DP Anderson and J. Kubiatowicz. The worldwide computer. *Scientific American*, 286(3):28–35, 2002.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[7] Benny Chor, Oded Goldreich, Johan Håstad, Joel Friedman, Steven Rudich, and Roman Smolensky. The bit extraction problem of t-resilient functions (preliminary version). In *FOCS*, pages 396–407. IEEE, 1985.

[8] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2009.

[9] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Byzantine fault tolerance: the time is now. In *LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, New York, NY, USA, 2008. ACM.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *In Proceedings of Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2005.

[11] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.

[12] Uriel Feige. Noncryptographic selection protocols. In *Proceedings of 40th IEEE Foundations of Computer Science(FOCS)*, 1999.

[13] Juan A. Garay and Rafail Ostrovsky. Almost-everywhere secure computation. In *EUROCRYPT*, pages 307–323, 2008.

[14] Dan Holtby, Bruce M. Kapron, and Valerie King. Lower bound for scalable byzantine agreement. *Distributed Computing*, 21(4):239–248, 2008.

[15] Svante Janson. On concentration of probability. *Combinatorics, Probability and Computing*, 11:2002, 1999.

[16] Bruce Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Scalable algorithms for byzantine agreement and leader election with full information. *ACM Transactions on Algorithms(TALG)*, 2009.

[17] Valerie King, Olumuyiwa Oluwasanmi, and Jared Saia. An empirical study of a scalable byzantine agreement aglrotihm. In *19th International Heterogeneity in Computing Workshop*, 2010.

[18] Valerie King and Jared Saia. From almost-everywhere to everywhere: Byzantine agreement in $\tilde{O}(n^{3/2})$ bits. In *International Symposium on Distributed Computing (DISC)*, 2009.

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

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

[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, page 58. ACM, 2007.

[22] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *In Computer Security Foundations Workshop*, pages 116–124, 1997.

[23] R. J. McEliece and D. V. Sarwate. On sharing secrets and reed-solomon codes. *Commun. ACM*, 24(9):583–584, 1981.

[24] M.O. Rabin. Randomized byzantine generals. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 403–409, 1983.

[25] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 1–14, 2003.

[26] E. Shi, A. Perrig, et al. Designing secure sensor networks. *IEEE Wireless Communications*, 11(6):38–43, 2004.

[27] A. Wright. Contemporary approaches to fault tolerance. *Communications of the ACM*, 52(7):13–15, 2009.