Journal on Information Security manuscript No.
(will be inserted by the editor)

Byzantine Set-Union Consensus using Efficient Set Reconciliation

Florian Dold · Christian Grothoff

Received: June 23, 2017

Abstract Applications of secure multiparty computation such as certain electronic voting or auction protocols require Byzantine agreement on large sets of elements. Implementations proposed in the literature so far have relied on state machine replication, and reach agreement on each individual set element in sequence.

We introduce *set-union consensus*, a specialization of Byzantine consensus that reaches agreement over whole sets. This primitive admits an efficient and simple implementation by the composition of Eppstein's set reconciliation protocol with Ben-Or's ByzConsensus protocol.

A free software implementation of this construction is available in GNUnet. Experimental results indicate that our approach results in an efficient protocol for very large sets, especially in the absence of Byzantine faults. We show the versatility of set-union consensus by using it to implement distributed key generation, ballot collection and cooperative decryption for an electronic voting protocol implemented in GNUnet.

This is a revised and extended version of a paper published under the same title at ARES 2016.

Keywords Byzantine agreement, secure multiparty computation, complexity

F. Dold
Inria
Inria Rennes Bretagne Atlantique
263 Avenue du General Leclerc
F-35042 Rennes
E-mail: florian.dold@inria.fr

C. Grothoff
Inria Rennes Bretagne Atlantique
263 Avenue du General Leclerc
F-35042 Rennes
E-mail: christian.grothoff@inria.fr

1 Introduction

Byzantine consensus is a fundamental building block for fault-tolerant distributed systems. It allows a group of peers to reach agreement on some value, even if a fraction of the peers are controlled by an active adversary. Theory-oriented work on Byzantine consensus often focuses on finding a single agreement on a binary flag or bit string. [1] More recent approaches for practical applications are mainly based on state machine replication (SMR), wherein peers agree on a sequence of state machine transitions. State machine replication makes it relatively easy to lift existing, non-fault-tolerant services to a Byzantine fault-tolerant implementation [2]. Each request from a client triggers a state transition in the replicated state machine that provides the service.

A major shortcoming of SMR is that all requests to the service need to be individually agreed upon in sequence by the replica peers of the state machine. This is undesireable since in unoptimized SMR protocols, such as PBFT [2], a single transition requires $O(n^2)$ messages to be exchanged for n replicas. Some implementations [3] try to address this inefficiency by optimistically processing requests and falling back to individual Byzantine agreements only when Byzantine behavior is detected. In practice this leads to very complex implementations whose correctness is hard to verify and that have weak progress guarantees [4].

The canonical example for a service where this inefficiency becomes apparent is the aggregation of values submitted by clients into a set. This scenario is relevant for the implementation of secure multiparty computation protocols such electronic voting [5], where ballots must be collected, and auctions [6], where bids must be collected. A direct implementation that reaches agreement on a set of m elements with SMR requires m sequential agreements, each consisting of $O(n^2)$ messages.

We introduce Byzantine set-union consensus (BSC) as an alternative communication primitive that allows this aggregation to be implemented more efficiently. In order to implement the set aggregation service described above, the peers first reconcile their sets using an efficient set reconciliation protocol that is not fault-tolerant but where the complexity is bounded even in the case of failures. Then, they use a variant of ByzConsensus [7] to reach Byzantine agreement on the union.

We assume a partially synchronous communication model, where non-faulty peers are guaranteed to successfully receive values transmitted by other non-faulty peers within an existing but unknown finite time bound [8]. Peers communicate over pairwise channels that are authenticated. Message delivery is reliable (i.e. messages arrive uncorrupted and in the right order) but the receipt of messages may be delayed. We make the same assumption as Castro and Liskov [2,9] about this delay, namely that it does not grow faster than some (usually exponential) function of wall clock time. We assume a computationally unbounded adversary that can corrupt at most $t = \lceil n/3 \rceil - 1$ peers creating Byzantine faults. The adversary is static, that is the set of corrupted peers is fixed before the protocol starts, but this set is not available to the correct peers. The actual number of faulty peers is denoted by f, with $f \leq t$.

The BSC protocol has message complexity $O(mn + n^2)$ when no peers show Byzantine behavior. When f peers show Byzantine behavior, the message complexity is $O(mnf + kfn^2)$, where k is the number of valid set elements exclusively available to the adversary. We will show how k can be bounded for common practical applications, since in the general case k is only bounded by the bandwidth available to the adversary. In practice, we expect kf to be significantly smaller than m. Thus, $O(mnf + kfn^2)$ is an improvement over using SMR-PBFT which would have complexity $O(mn^2)$.

We have created an implementation of the BSC protocol by combining Ben-Or's protocol for Byzantine consensus [7] with a bounded variant of Eppstein's protocol for efficient set reconciliation [10]. We demonstrate the practical applicability of our resulting abstraction by using BSC to implement distributed key generation, ballot collection and cooperative decryption from the Cramer-Gennaro-Schoenmakers remote electronic voting scheme [5] in the GNUnet framework.

2 Background

The Byzantine consensus problem [11] is a generalization of the consensus problem where the peers that need to reach agreement on some value might also exhibit Byzantine faults.

Many specific variants of the agreement problem (such as interactive consistency [12], k-set consensus [13], or leader election [14] and many others [15]) exist. We will focus on the consensus problem, wherein each peer in a set of peers $\{P_1, \dots, P_n\}$ starts with an initial value $v_i \in M$ for an arbitrary fixed set M. At some point during the execution of the consensus protocol, each peer irrevocably decides on some output value $\hat{v}_i \in M$. Informally, a protocol that solves the consensus problem must fulfill the following properties:[1]

– *Agreement:* If peers P_i, P_j are correct, then $v_i = v_j$.
– *Termination:* The protocol terminates in a finite number of steps.
– *Validity:* If all correct peers have the same input value $\tilde{v}$, then all correct peers decide on $\tilde{v}$.

Some definitions of the consensus problem also include *strong validity*, which requires the value that is agreed upon to be the initial value of some correct peer [16]. The consensus protocol presented in this paper does not offer strong validity; in fact, for a set union operation this is not exactly desirable as the goal is to have all peers agree a union of all of the original sets, not on some peer's initial subset.

[1] Different variations and names can be found in the literature. We have chosen a definition that extends to our generalization to sets later on.

2.1 The FLP Impossibility Result

A fundamental theoretical result (often called FLP impossibility for the initials of the authors) states, informally, that no deterministic protocol can solve the consensus problem in the asynchronous communication model, even in the presence of only one crash-fault [17].

While this result initially seems discouraging, the conditions in which FLP impossibility holds are quite specific and subtle [18]. They have been challenged in a number of ways, including the following:

- *Common coin:* Some protocols introduce a shared source of randomness that the adversary cannot predict or bias. This breaks the assumption that the protocol must be deterministic. In practice, these protocols are very complex and often use variants of secret-sharing and weaker forms of Byzantine agreement to implement the common coin [19–21]. Implementing a common coin oracle resilient against an active adversary is non-trivial and usually required extra assumptions such as a trusted dealer in the startup phase [22] or shared memory [23]. Recent designs to implement a Byzantine fault-tolerant bias-resistant public random generator only scale to hundreds of participants and still have relatively high failure rates (reported at 0.08% for and adversary power bounded at $\frac{1}{3}$ and 32 participants) [24].
- *Failure oracles:* Approaches based on unreliable failure detectors [25] augment the model with oracles for the detection of faulty nodes. Much care has to be taken not to violate correctness of the protocol by classifying too many correct peers as faulty; this is problem is present in early systems such as Rampart [26] and SecureRing [27] as noted by Castro and Liskov [2,9]. While the theory of failure detectors is quite established for the non-Byzantine case, it is not clear whether they are still useful in the presence of Byzantine faults.
- *Partial synchrony:* A model where a bound on the message delay or clock shift exists but is unknown or is known but only holds from an unknown future point in time is called partial synchrony. The FLP result does not hold in this model [8].
- *Minimal synchrony:* The definition of synchrony used by the FLP impossibility result can be split into three types of synchrony: Processor synchrony, communication synchrony and and message ordering synchrony. Dolev et al. [28] show that consensus is still possible if only certain subsets of these three synchrony assumptions are fulfilled.

This work follows the path of [8] in relaxing the full asynchrony assumption behind the FLP impossibility result.

2.2 Byzantine consensus in the partially synchronous model

The protocols presented in this paper operate within the constraints of the partially synchronous model, where participants have some approximate information about time.

A fundamental result is that no Byzantine consensus protocol with n peers can support $\lceil n/3 \rceil$ or more Byzantine faults in the partially synchronous model [8].

Early attempts at implementing Byzantine consensus with state machine replication are SecureRing [27] and Rampart [26]. A popular design in the partially synchronous model is Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) [2,9]. PBFT uses a leader to coordinate peers (called *replicas* in BPFT terminology). When replicas detect that the leader is faulty, they run a leader-election protocol to appoint a new leader.

PBFT guarantees progress as long as the message delay does not grow indefinitely for some fixed growth function[2]. The approach taken by BPFT (and several derived protocols) has several problems [4]: In practice, malicious participants are able to slow down the system significantly. In fact, PBFT can fail to make progress entirely when facing an adversarial scheduler that violates PBFT's weak synchrony requirements [29].

Some more recent Byzantine state machine replication protocols such as Q/U [30] or Zyzzyva [3] have less overhead per request since they optimize for the non-Byzantine case. This comes, however, often at the expense of robustness in the presence of Byzantine faults [4], not to mention that correctness proofs for the respective protocols and the implementation of state machine replication are notoriously difficult [31].

2.3 Gradecast

A key building block for our protocol is Feldman's Gradecast protocol [20]. In contrast to an unreliable broadcast, Gradecast provides correctness properties to the receivers, even if the leader is exhibiting Byzantine faults.

In a Gradecast, a leader P_L broadcasts a message m among a fixed set $\mathcal{P} = \{P_1, \dots, P_n\}$ of peers. For notational convenience, we assume that $P_L \in \mathcal{P}$. These are the communication steps for peer P_i:

1. LEAD: If $i = L$, send the input value v_L to $\mathcal{P}$
2. ECHO: Send the value received in LEAD to $\mathcal{P}$.
3. CONFIRM: If a common value $\overline{v}$ was received at least $n-t$ times in round ECHO, send $\overline{v}$ to $\mathcal{P}$. Otherwise, send nothing.

Afterwards, each peer assigns a confidence value $c_i \in \{0, 1, 2\}$ that "grades" the correctness of the broadcast. The result is a graded result tuple $\langle \hat{v}_i, c_i \rangle$ containing the output value $\hat{v}_i$ and the confidence c_i. The grading is done with the following rules:

– If some $\hat{v}$ was received at least $n-t$ times in CONFIRM, output $\langle \hat{v}, 2 \rangle$.
– Otherwise, if some $\hat{v}$ was received at least $t+1$ times in CONFIRM, output $\langle \hat{v}, 1 \rangle$.

[2] In practice, exponential back-off is used.

– Otherwise, output $\langle \perp, 0 \rangle$. Here, $\perp$ denotes a special value that indicates the absence of a meaningful value.

For the c_i, the following correctness properties must hold:

1. If $c_i \geq 1$ then $\hat{v}_i = \hat{v}_j$ for correct P_i and P_j
2. If P_L is correct, then $c_i = 2$ and $\hat{v}_i = v_L$ for correct P_i.
3. $|c_i - c_j| \leq 1$ for correct P_i and P_j.

When a correct peer P_i receives a Gradecast with confidence 2, it can deduce that all other peers received the same message, but some other peers might have only received it with a confidence of 1. Receiving a Gradecast with confidence 1 also guarantees that all other correct peers received the same message. However, it indicates that the leader behaved incorrectly. No assumption can be made about the confidence of other peers. Receiving a Gradecast with confidence 0 indicates that the leader behaved incorrectly and, crucially, that all other correct peers *know* that the leader behaved incorrectly.

A simple counting argument proves that the above protocol satisfies the three Gradecast properties. [20]

2.4 ByzConsensus

ByzConsensus [7] uses Gradecast to implement a consensus protocol for simple values. Each peer begins with a starting value $s_i^{(1)}$ and the list of all n participants $\mathcal{P}$. Each peer also starts with an empty blacklist of corrupted peers. If a peer is ever blacklisted, it is henceforth excluded from the protocol. In ByzConsensus, Gradecast is used to force corrupt peers to either expose themselves as faulty—and consequently be excluded—by gradecasting a value with low confidence, or to follow the protocol and allow all peers to reach agreement.

ByzConsensus consists of at most $f+1$ sequentially executed super-rounds $r \in 1 \ldots f+1$ where $f \leq t$. In each super-round, each peer leads a Gradecast using their candidate value $s_i^{(r)}$; these n Gradecasts can be executed in parallel. Leaders where the Gradecast results in a confidence of less than 2 are put on the blacklist. Recall that different correct peers might receive a Gradecast with different confidence; thus peers do not necessarily agree on the blacklist.

At the end of each super-round, each peer computes a new candidate value $s_i^{(r+1)}$ using the value that was received most often from the Gradecasts with a confidence of as least 1. If $s_i^{(r)}$ was received more than $n - t$ times, then $r = f$ and the next round is the last round.

If the final candidate value does not receive a majority of at least $2t + 1$ among the n Gradecasts, or if the blacklist has more than t entries, then the protocol failed: either more than t faults happened or, in the partially synchronous model, correct peers did not receive a message within the designated round due to the delayed delivery.

ByzConsensus has message complexity $O(fn^3)$. While the asymptotic message complexity is obviously worse than the $O(n^2)$ of PBFT, there is a way to use set reconciliation to benefit from the parallelism of the Gradecast rounds and thereby reduce the complexity to $O(fn^2)$.

2.5 Set reconciliation

The goal of set reconciliation is to identify the differences between two large sets, say S_a and S_b, that are stored on two different machines in a network. A simple but inefficient solution would be to transmit the smaller of the two sets, and let then receiver compute and announce the difference. Research has thus focused on protocols that are more efficient than this naive approach with respect to the amount of data that needs to be communicated when the sets S_a and S_b are large, but their symmetric difference $S_a \oplus S_b$ is small.

An early attempt to efficiently reconcile sets [32] was based on representing sets by their characteristic polynomial over a finite field. Conceptually, dividing the characteristic polynomials of two sets cancels out the common elements, leaving only the set difference. The characteristic polynomials are transmitted as a sequence of sampling points, where the number of sampling points is proportional to the size of the symmetric difference of the sets S_a and S_b. The number of sampling points can be approximated with an upper bound, or increased on the fly should a peer be unable to interpolate a polynomial. While theoretically elegant, the protocol is not efficient in practice. The computational complexity of the polynomial interpolation grows as $O(|S_a \oplus S_b|^3)$ and uses rather expensive arithmetic operations over large finite fields.

A practical protocol was first proposed by Eppstein et al. in 2011. [10] It is based on invertible Bloom filters (IBFs), a probabilistic data structure that is related to Bloom filters [33], and stratas for difference estimation.

2.5.1 Invertible Bloom Filters

An IBF is a constant-size data structure that supports four basic operations, *insert*, *delete*, *decode* and *subtract.*

Insert and *delete* operations are commutative operations encoding a *key* that uniquely identifies a set element, typically derived from the element via a hash function.

The *decode* operation can be used to extract some or all of the updates, returning the key and the sign of the operation, that is either *insert* or *delete.* Since the data structure uses constant space, decoding cannot always succeed. Decoding is a probabilistic operation that is more likely to succeed when the IBF is sparse, that is the number of encoded operations (*excluding* the operations that canceled each other out) is small. The decoding process can also be partially successful, if some elements could be extracted but the remaining IBF is non-empty. Extracting an update by decoding an IBF is only possible if the key was recorded only once in the IBF. However, storing a deletion or insertion of the same key twice or more (not counting operations that canceled each other out) makes both updates impossible to decode.

IBFs of the same size can also be *subtracted* from each other. When subtracting IBF_b from IBF_a, the resulting structure $\mathrm{IBF}_c := \mathrm{IBF}_a - \mathrm{IBF}_b$ contains all insertions and deletions from IBF_a, and insertions from IBF_b are recorded as deletions and deletions from IBF_b are recorded as insertions in IBF_c. Effectively, the IBF subtraction allows to compute the difference between two sets simply by encoding each set as an IBF using only insertion operations.

Under the hood, an IBF of size n is an array of n buckets. Each bucket holds three values:

- A signed counter that handles overflow via wrap-around. Recording an insertion or deletion adds -1 or $+1$ to the counter, respectively.
- An $\oplus$-sum[3], called the `keySum`, over the keys that identify set the elements that were recorded in the bucket.
- An $\oplus$-sum, called the `keyHashSum`, over a the hash $h(\cdot)$ of each key that was recorded in the bucket.

As with ordinary Bloom filters, encoding an update in an IBF records the update in k different buckets of the IBF. The indices of buckets that record the update are derived via a k independent hash functions from the key of the element that is subject of the update. We write $\mathrm{Pos}(x)$ for the set of array positions that correspond to the element key x.

Before we describe the decoding process, we introduce some terminology. A bucket is called a *candidate bucket* if its counter is -1 or $+1$, which might indicate that the `keySum` field contains the key of an element that was the subject of an update. Candidate buckets that contain the key of an element that was previously updated are called *pure buckets*. Candidate buckets are not necessarily pure buckets, since a candidate bucket could also result from, for example, first inserting an element key e_1 and then deleting e_2 when $\mathrm{Pos}(e_1) \cap \mathrm{Pos}(e_1) \neq \emptyset$ and $\mathrm{Pos}(e_1) \neq \mathrm{Pos}(e_2)$.

The `keyHashSum` provides a way to detect if a candidate bucket is not a pure bucket, namely when $h(\texttt{keySum}) \neq \texttt{keyHashSum}$. The probability of classifying an impure bucket as pure with this method is dependent on the probability of a hash collision. Another method to check for an impure candidate bucket with index i is to check whether $i \notin \mathrm{Pos}(\texttt{keySum})$.

The decoding process then simply searches for buckets that are, with high probability, pure. When the `count` field of the bucket is 1, the key decoding procedure reports the key as "inserted" and exececutes a deletion operation with that key. When the `count` field is -1, the key is reported as "deleted" and subsequently an insertion operation is executed.

With a probability that increases with sparser IBFs, decoding one element may cause one or more other buckets to become pure, allowing the decoding to be repeated. If none of the buckets is pure, the IBF is undecodable, and a larger IBF must be used, or the reconciliation could fall back to the naive approach of sending the whole set.

[3] The $\oplus$ denotes bit-wise exclusive or.

The IBF decoding process is particularly suitable for reconciling large sets with small differences. When the symmetric difference between the sets is small enough compared to the size of the IBFs, the result IBF_c of the subtraction can be decoded, since the common elements encoded in IBF_a and IBF_b cancel each other out. This makes it possible to obtain the elements of the symmetric difference, even when the IBFs that represent the full sets can not be decoded.

As long as the symmetric difference between the original sets S_a and S_b can be approximated well enough, IBFs can be used for set reconciliation by encoding S_a in IBF_a and S_b in IBF_b. One of the IBFs is sent over the network, the $\mathrm{IBF}_c = \mathrm{IBF}_a - \mathrm{IBF}_b$ is computed and decoded. Should the decoding (partially) fail, the same procedure is repeated with larger IBFs.

2.5.2 Difference Estimation with Stratas

In order to select the initial size of the IBF appropriately for the set reconciliation protocol, one needs an estimate of the symmetric difference between the sets that are being reconciled. Eppstein et al. [10] describe a simple technique, called strata estimation, that is accurate for small differences. While Eppstein et al. suggest combining the strata estimator, with a min-wise estimator, which is more accurate for large differences, our work only requires the strata estimators.

A strata estimator is an array of fixed-size IBFs. These fixed-size IBFs are called *strata* since each of them contains a sample of the whole set, with increased sampling probability towards inner strata. Similar to how two IBFs can be subtracted, strata estimators are subtracted by pairwise subtraction of the IBFs they consist of.

The set difference is estimated by having both peers encode their set in a strata estimator. One of the strata estimators is then sent over to the other peer, which subtracts the strata estimators from each other. With every IBF of the strata estimator that results from the subtraction, a decoding attempt is made. The number of successfully decoded elements in each stratum allows an estimate to be made on the set difference, which is then used to determine the size of the IBF for the actual set reconciliation.

3 Our approach

We now describe how to combine the previous approaches into a protocol for Byzantine fault-tolerant set consensus. The goal of the adversary is to sabotage timely consensus among correct peers, e.g. by increasing message complexity or forcing timeouts.

A major difficulty with agreeing on a set of elements as a whole is that malicious peers can initially withhold elements from the correct peers and later send them only to a subset of the correct peers. This could possibly happen at a time when it is too late to reconcile the remaining difference caused by distributing these elements. We assume that the number of these elements that are initially known to the adversary but not to all correct peers is bounded by k, where k exists but is not necessarily known to the correct participants.

3.1 Definition

We now give a definition of set-union consensus that is motivated by practical applications to secure multiparty computation protocols such as electronic voting, which are discussed in more detail in Section 7.

Consider a set of n peers $\mathcal{P} = \{P_1, \ldots, P_n\}$. Fix some (possibly infinite) universe M of elements that can be represented by a bit string. Each peer P_i has an initial set $S_i^{(0)} \subseteq M$.

Let $R : 2^M \to 2^M$ be an idempotent function that canonicalizes subsets of M by replacing multiple conflicting elements with the lexically smallest element in the conflict set and removes invalid elements. What is considered conflicting or invalid is application-specific. During the execution of the set-union consensus protocol, after finite time each peer P_i irrevocably commits to a set S_i such that:

1. For any pair of correct peers P_i, P_j it holds that $S_i = S_j$.
2. If P_i is correct and $e \in S_i^0$ then $e \in S_i$.
3. The set S_i is canonical, that is $S_i = R(S_i)$.

The canonicalization function allows us to set an upper bound on the number of elements that can simultaneously be in a set. For example in electronic voting, canonicalization would remove malformed ballots and combine multiple different (encrypted) ballots submitted by the same voter into a single "invalid" ballot for that voter.

3.2 Byzantine set-union consensus (BSC) protocol

Recall that every peer P_i, $0 < i \leq n$ starts with a set $S_i^{(0)}$. The BSC protocol incorporates two subprotocols, bounded set reconciliation and lower bound agreement, and uses those to realize an efficient Byzantine fault-tolerant variant of ByzConsensus. An existing generalization of IBFs to multi-party set reconciliation [34] based on network coding is not applicable to this problem, as it requires trusted intermediaries.

The basic problem solved by the two subprotocols is bounding the cost of Eppstein's set reconciliation. Given a set size difference between two peers of k, the expected cost of Eppstein's set reconcilation is $O(k)$ if both participants are honest. However, we need to ensure that malcicious peers cannot generally raise the complexity to $O(m)$ where m is the size of the union.

For this, we use a bounded variant of Eppstein's set reconciliation protocol, which is given a lower bound $\mathcal{L}$ on the size of the set of elements shared by all honest participants. Given such a lower bound, the bounded set reconciliation protocol must detect faulty participants in $O(k + (m - \mathcal{L}))$. We note that for $\mathcal{L} = 0$, the bounded set reconciliation is still allowed to cost $O(m)$.

3.2.1 Bounded set reconciliation

In *bounded* set reconciliation we are thus concerned with modifications that ensure that a set reconciliation step between an honest and a faulty peer either succeeds after $O(k)$ traffic, or aborts notifying the honest peer that the other peer is faulty. While we use probabilities to detect faulty behavior, we note that suitable parameters can be used to ensure that false-positives are rare, say $1 : 2^{128}$, and thus as unlikely as successful brute-force attacks against canonical cryptographic primitives, which BSC also assumes to be safe.

To begin with, to bound the complexity of Eppstein set reconciliation one needs to bound the number of iterations the protocol performs. Assuming honest peers, the initial strata estimation ensures that the IBFs will decode with high probability, resulting in Eppstein's claim of single-round complexity. Given aggressive choices of the parameters to improve the balance between round-trips and bandwidth consumption, decoding failures can happen with non-negligible probability in practice. In this case, the process can simply be restarted using a different set of hash functions and an IBF doubled in size. This addresses issues caused by conservative choices for IBF sizes that optimize for the average case. What is critical is that the probability of such failures remains small enough that after if the number of rounds exceeds some constant, we can assert faulty behavior and overall remain within the $O(k)$ bound assuming individual rounds are bounded by $O(k)$.

Another problem with Eppstein's original protocol related to aggressive parameter choices is that iterative decoding does not always have to end with an empty or an undecodable IBF. Specifically, the decoding step can sometimes decode a key that was never added to the IBF, simply because the two purity checks are also probabilistic. This is usually not an issue, as when a decoder requests the transmission of the element corresponding to improperly decoded key, the presumed element's owner can indicate a decoding failure at that time. Here, another round of the protocol is unlikely to produce the same error and would again fix the problem. However, given reasonably short strings for the `hashKeySum`, it is actually even possible to obtain a looping IBF that spawns an infinite series of "successfully" decoded keys. Here, the implementor has to be careful to ensure that the iterated decoding algorithm terminates. Instead of mandating an excessively long `hashKeySum` to prevent this, it is in practice more efficient to handle this case by stopping the iteration and reporting the IBF as undecodable when the number of decoded keys exceeds a threshold proportional to the size of the IBF.

We also need to consider the bandwidth consumption of an individual round. To cause more than $O(k)$ traffic, a malicious peer could produce strata that result in a huge initial symmetric difference. In this case, the initial size of the IBF may exceed $O(k)$. We address this problem by not permitting the use of Eppstein's method if the symmetric difference definitively exceeds $\frac{n-\mathcal{L}}{2}$, where n is the smaller of the two set sizes.[4] Instead, once the estimate of the symmetric difference substantially exceeds this threshold, the reconciliation algorithm falls back to sending the complete set. As this creates $O(m)$ traffic, it must only be allowed under certain conditions.

First, we consider the case where the honest peer has the larger set. Here, the honest peer P_i will only send its full set if the set difference is no larger than $|S_i| - \mathcal{L}$, and otherwise report a fault. This ensures that a malicious peer cannot arbitrarily request the full set from honest peers.

Second, we consider the case where the honest peer P_i is facing a faulty peer that claims to have a huge set. This is can happen either directly from the strata estimator, or after P_i observes a constant number of successive IBF decoding failures.[5] At this point, instead of passively accepting the transmission of elements, the receiver P_i checks that a sufficient number of the elements received are not in S_i. Let R be the stream of elements e received at any point in time. We assume that the sender is required to transmit the elements in randomized order. Thus, if $|R \cap S_i| - |R \setminus S_i| \geq 128$, P_i can determine that the sender is faulty with probability $2^{128} : 1$, as the the $\frac{n}{2}$-threshold for converting to complete set transmission ensures that for an honest sender less than half of the elements would be in S_i.

Finally, we note that the individual *insert*, *delete*, *decode* and *subtract* operations on the IBF are all constant time and that IBFs are also constant size. Thus, given a constant number of rounds and a bound on the bandwidth per round, we have implicitly assured that memory and CPU consumption of the bounded set reconciliation is also $O(k + (m - \mathcal{L}))$.

3.2.2 Lower bound agreement

To provide a lower bound on the permissable set size for set reconciliation, BSC first executes a protocol for *lower bound agreement* (LBA). In this first step, every correct peer P_i learns a superset $S_i^{(1)}$ of the union of all correct peers' initial sets, as well as a lower bound ℓ_i for the minimum number of elements shared by all correct peers where $n - \ell_i \leq k$. Note that neither $S_i^{(1)} = S_j^{(1)}$ nor $\ell_i = \ell_j$ necessarily hold even for correct peers P_i and P_j. Our LBA protocol proceeds in three steps:

(i) All peers reconcile their initial set with each other, using pairwise bounded set reconciliation using a lower bound of $\mathcal{L} = 0$.
(ii) All peers send their current set size to each other, and each peer P_i sets sets ℓ_i to the $(t + 1)$-smallest set size that P_i received.
(iii) All peers again reconcile their sets with each other, using pairwise bounded set reconciliation.

[4] The optimal formula here depends on the size ratio of IBF element to the transmission size of an individual element and the estimated size of the set overlap. However, to simplify the exposition, we will assume a simple 50% threshold henceforth.

[5] Each failure causes the IBF size to double and thus corresponds to a doubling of the set difference estimate. Thus, the number of decoding failures could remain the threshold that causes an abort, while the set difference estimate substantially exceeds $2(|S_i| - \mathcal{L})$.

The third step is necessary to ensure that every correct P_i has at least ℓ_i elements, since malicious peers could use the k elements initially withheld to force an honest peer's set size below the $(t+1)$-smallest set size. Thanks to the repetition even if ℓ_i is different for each peer, it is guaranteed that P_i has at least ℓ_i elements in common with every other good peer.

In subsequent set reconciliations, ℓ_i can be used to bound the traffic that malicious peers are able to cause by falsely claiming to have a large number of elements missing. LBA itself has complexity $O(nmf)$: initially all malicious peers can *once* claim to have empty sets with all other peers. LBA ensures that for the remainder of the protocol, a correct peer with m_i elements can stop sending elements to malicious peer P_M after P_M requested $m_i - \ell_i \leq k$ elements by reducing the complexity of bounded set reconciliation with peer m_i to $O(k)$ using $\mathcal{L} = \ell_i$.

3.2.3 Exact set agreement

After LBA, an *exact set agreement* is executed, where all peers reach Byzantine agreement for a super-set of the set reached in LBA. The exact set agreement is implemented by executing a variant of ByzConsensus which instead of sending values reconciles sets.

The Gradecast is adapted as follows:

(i) LEAD: If $i = L$, reconcile the input set V_L with $\mathcal{P}$.
(ii) ECHO: Reconcile the set received in LEAD with $\mathcal{P}$.
(iii) CONFIRM: Let $\mathcal{U}_E$ be the union of all sets received in the ECHO round, and $N_E(e)$ the number of times a single set element e was received. If $\bigvee_{e \in \mathcal{U}_E} t < N_E(e) < n - t$, send $\bot$ (where $\bot \neq \emptyset$). Otherwise send $\mathcal{U}_E - \{e \mid N_E(e) \leq t\}$ to $\mathcal{P}$.

The grading rules are also adapted to sets. Let $\mathcal{U}_C$ be the union of sets received in CONFIRM, $N_C^+(e)$ the number of times a single element $e \in \mathcal{U}_C$ was received, and $N_C^-(e)$ the number of sets (not $\bot$) received in CONFIRM that excluded e.

– If $\bigwedge_{e \in \mathcal{U}} N_C^+(e) \geq n - t \vee N_C^-(e) \geq n - t$,
  output $\langle \{e \mid N_C^+(e) \geq n - t\}, 2 \rangle$.
– Otherwise if $\bigwedge_{e \in \mathcal{U}_C} N_C^+(e) > t \wedge N_C^+(e) \geq N_C^-(e)$
  or $\bigwedge_{e \in \mathcal{U}_C} N_C^-(e) > t \wedge N_C^-(e) > N_C^+(e)$,
  output $\langle \{e \mid N_C^+(e) > t \wedge N_C^+(e) \geq N_C^-(e)\}, 1 \rangle$.
– Otherwise, output $\langle \bot, 0 \rangle$.

Similar to ByzConsensus, the BSC consists of at most $f+1$ super-rounds, where $f \leq t$. Each peer P_i starts with $S_i^{(1)}$ as its current set. In sequential super-rounds, all peers lead a Gradecast for their candidate set. Like in ByzConsensus, if P_i receives a Gradecast with a confidence value that is not 2, then P_i puts the leader of the Gradecast on its blacklist, and correct peers stop all communictation with peers on their blacklist.

At the end of each super-round, peers update their candidate set as follows. Let n' be the number of leaders that gradecasted a set with a non-zero confidence. The new candidate set contains all set elements that were included in at least $\lceil n'/2 \rceil$ sets that were gradecasted with a non-zero confidence value. If all elements occur with a $(n-t)$-majority, then the next round is the last round. The output of the consensus protocol is the candidate set after the last round—or failure if $f > t$.

We give a correctness proof that generalizes Feldman's proof for Gradecast of single values [19, Section 4.1].

Lemma 1 *If two correct peers send sets $A \neq \bot$ and $B \neq \bot$ respectively in CONFIRM, then $A = B$.*

Proof Proof by contradiction and counting argument. Assume w.l.o.g. that $e \in A$ and $e \notin B$. At least $n-t$ peers must have echoed a set that includes e to the first peer. Suppose f of these peers were faulty, then at least $n-t-f > t$ good peers included e in the ECHO transmission to the second peer. If $e \notin B$, then $t < N_E(e) < n-t$. In this case, an honest second peer must output $B = \bot$. Contradiction.

Theorem 1 *The generalization of Gradecast to sets satisfies the three Gradecast properties.*

Proof We show that each property holds:

- Property 1 (If $c_i, c_j \geq 1$ then $\hat{V}_i = \hat{V}_j$ for correct P_i and P_j): Assume w.l.o.g. that $e \in \hat{V}_i \setminus \hat{V}_j$.
  For $e \in \hat{V}_j$, P_i must have received e at least $N_C^+(e) > t$ times in CONFIRM. Given $f \leq t$ failures, at least one honest peer must thus have included e in CONFIRM. According to Lemma 1, then all $n-f$ honest peers must either include e in CONFIRM or send $\bot$.
  Because $\bot$ is not a set, this leaves at most all $f \leq t$ faulty peers that can send a set without e. But for $e \notin \hat{V}_j$ we need $N_C^-(e) \geq t+1$. Contradiction.
- Property 2 (If P_L is correct, then $c_i = 2$ and $\hat{V}_i = \hat{V}_L$ for correct P_i): All $n-f \geq n-t$ good peers ECHO and CONFIRM the same set. By the grading rules, they must output a confidence of 2.
- Property 3 ($|c_i - c_j| \leq 1$ for correct P_i and P_j): Proof by contradiction. Assume w.l.o.g. $c_i = 2$ and $c_j = 0$. $c_i = 2$ implies that for each $x \in \hat{V}_i$ at least $n-t$ peers (and thus $(n-t)-f \geq t+1$ correct peers) must have sent a set in CONFIRM that includes x. For any $y \notin \hat{V}_i$, $n-t$ peers (and thus $(n-t)-f \geq t+1$ correct peers) must have sent a non-$\bot$ set in CONFIRM that excludes y.
  Given $c_j = 0$, there must have been an element e such that, $N_C^+(e) \leq t$ and $N_C^-(e) \leq t$ for P_j. However, we just derived that for all elements either $N_C^+(e) > t$ or $N_C^-(e) > t$. Contradiction. □

Given the Gradecast properties for sets, the correctness argument given by Ben-Or [7] for the Byzantine consensus applies to BSC's generalization to sets.

As described, the protocol has complexity $O(mnf + fkn^3)$. However, the n parallel set reconciliation rounds in each super-round can be combined by tagging the set elements that are being reconciled in the LEAD, ECHO and CONFIRM rounds with the respective leader L. Because LBA (via $n - \ell_i \leq k$) and bounded set reconciliation limit mischief for the combined super-round, each malicious peer can, as leader, *once* cause bounded set reconciliation during the ECHO round to all-to-all transmit at most k extra elements, resulting in a total of $O(fkn^2)$ extra traffic over all $f+1$ rounds. Before exposing themselves this way, non-leading malicious peers can only cause $O(f^2kn)$ additional traffic during all ECHO rounds. Finally, malicious peers can also cause at most $O(fkn^2)$ traffic in the CONFIRM round. Thus, BSC has overall message complexity of $O(mnf + fkn^2)$.

4 Implementation

We implemented the BSC protocol in the SET and CONSENSUS services of GNUnet [35].

4.1 The GNUnet framework

GNUnet is composed of various components that run in separate operating system processes and communicate via message passing. Components that expose an interface to other components are called *services* in GNUnet. The main service used by our implementation is the CADET service, which offers pairwise authenticated end-to-end encryption between all participants. CADET uses a variation of the Axolotl public key ratcheting scheme and double-encrypts using both TwoFish and AES. [36] The resulting encryption is relatively expensive compared to the other operations, and thus dominates in terms of CPU consumption for the experiments.

4.2 Set reconciliation

Bounded set reconciliation is implemented in the SET service. The SET service provides a generic interface for set operations between two peers; the operations currently implemented are the IBF-based set reconciliation and set intersection [37].

In addition to the operation-specific protocols, the following aspects are handled generically (i.e. independent of the specific remote set operation) in the SET service:

Local set operations
: Applications need to create sets and perform actions (iteration, insertion, deletion) on them locally.

Concurrent modifications
: While a local set is in use in a network operation, the application may still continue to mutate that set. To allow this without interfering with concurrent the network operations, changes are versioned. A network operation only sees the state of a set at the time the operation was started.

Lazy copying
: Some applications building on the SET service—especially the CONSENSUS service described in the next section—manage many local sets that are large but only differ in a few elements. We optimize for this case by providing a lazy copy operation which returns a logical copy of the set without duplicating the sets in memory.

Negotiating remote operations
: In a network operation, the involved peers have one of two roles: The acceptor, which waits for remote operation requests and accepts or rejects them, as well as the initiator, which sends the request.

Our implementation estimates the initial difference between sets only using *strata estimators* as described by Eppstein [10]. However, we compress the strata estimator—which is 60KB uncompressed—using `gzip`. The compression is highly effective at reducing bandwidth consumption due to the high probability of long runs of zeros or ones in the most sparse or most dense strata respectively.

We also use a *salt* when deriving the bucket indices from the element keys. When the decoding of an IBF fails, the IBF size is doubled and the salt is changed. This prevents decoding failures in scenarios where keys map to the same bucket indices even modulo a power of two, where doubling the size of the IBF does not remove the collision.

4.3 Set-Union consensus

To keep the description of the set-union consensus protocol in the previous section succinct, we merely stated that peers efficiently transmit sets using the reconciliation protocol. However, given that the receiving peer has usually many sets to reconcile against, an implementation needs to be careful to ensure that it scales to large sets as intended.

The key goal is to avoid duplicating full sets and to instead focus on the differences. New sets usually differ in only a few elements, thus our implementation avoids copying entire sets. Instead, in the leader round we just store the set of differences with a reference to the original set. In the ECHO and CONFIRM round, we also reconcile with respect to the set we received from the leader, and not a peer's current set. In the ECHO round, we only store one set and annotate each element to indicate which peer included or excluded that element. This also allows for a rather efficient computation of the set to determine the $\perp$-result in the CONFIRM round.

4.4 Evaluating malicious behavior

For the evaluation, our CONSENSUS service can be configured to exhibit the following types of adversarial behavior:

– *SpamAlways:* A malicious peer adds a constant number of additional elements in every reconciliation.
– *SpamLeader:* A malicious peer adds a constant number of additional elements in reconciliations where the peer is the leader.
– *SpamEcho:* A malicious peer adds a constant number of additional elements in echo rounds.
– *Idle:* Malicious peers do not participate actively in the protocol, which amounts to a crash fault from the start of the protocol. This type of behavior is not interesting for the evaluation, but used to test the implementation with regards to timeouts and majority counting.

For the *Spam-** behaviors, two different variations are implemented. One of them ("*-replace") always generates new elements for every reconciliation. This is not typical for real applications where the number of stuffable elements ought to be limited by set canonicalization. However, this shows the performance impact in the worst case. The other variation ("*-noreplace") reuses the same set of additional elements for all reconciliations, which is more realistic for most cases. We did not implement adversarial behaviour where elements are elided, since the resulting traffic is the same as for additional elements, and memory usage would only be reduced.

5 Experimental results

All of the experiments were run on a single machine with a 24-core 2.30GHz Intel Xeon E5-2630 CPU, and GNUnet SVN revision 36765. We used the `gnunet-consensus-profiler` tool, which is based on GNUnet's TESTBED service [38], to configure and launch multiple peers on the target system. We configured the profiler to emulate a network of peers connected in a clique topology (via loopback, without artificial latency). Elements for the set operations are randomly generated and always 64 bytes large.

Bandwidth consumption was measured using the statistics that GNUnet's CADET service [36] provides. Processor time was measured using GNUnet's resource reporting functionality, which uses the `wait3` system call for that purpose.

5.1 Bounded set reconciliation

We now summarize the experimental results for the bounded set reconciliation protocol between two peers. We first measured the behavior of the set reconciliation if identical sets were given to both peers (Figure 1 and 2). Figure 1 shows that total CPU utilization generally grows slowly as the set size

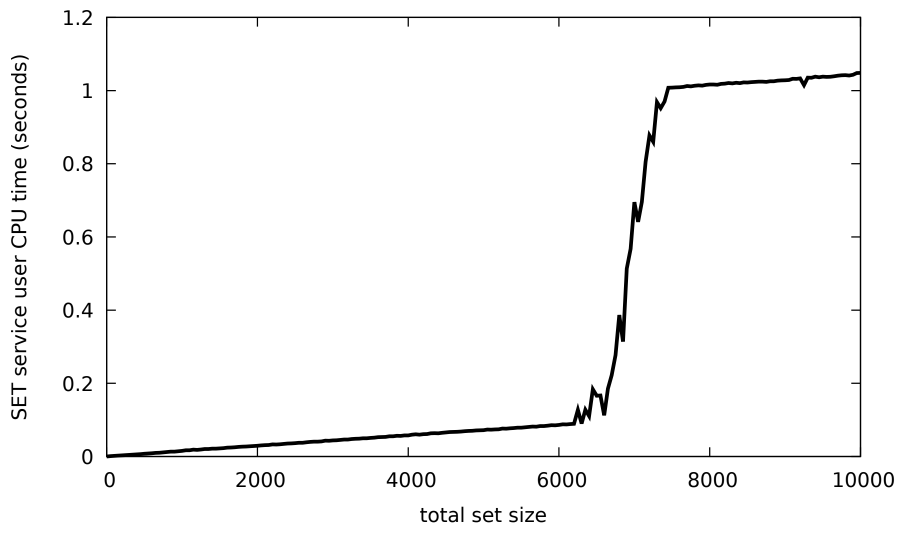

Fig. 1: CPU system time for the SET service in relation to total set size. Average over five executions.

increases. The sudden jump in processing time that is visible at around 7,000 elements can most likely be explained by cache effects. The effect could not be observed when we ran the experiment under profiling tools.

Figure 2 shows that bandwidth consumption does not grow linearly with the total set size, as long as the set size difference between the two peers is small. The logarithmic increase of the traffic with larger sets can be explained by the compression of strata estimators: The k-th strata samples the set with probability 2^{-k}, and for small input sets the strata tends to contain long runs of zeros that are more easily compressed.

We also measured the behavior of the set reconciliation implementation if the sets differed. Figure 3 and 4 show that—as expected—CPU time and bandwidth do grow linearly with the symmetric difference between the two sets.

Finally, we analyzed what happens at the point where the algorithm switches from transmitting set differences to full sets. Figure 5 shows the bandwidth in relation to the symmetric set difference, for different total numbers of elements in the shared set. Up to the threshold point where the algorithm switches from IBFs to full set transmission, we expect the transmission size to grow steeply, and then afterwards continue linearly at a lower rate again. If the handover threshold is chosen well, the two lines should meet. This is the case in the dashed curve in Figure 5. The small bump at a set difference of ≈ 800 is due to an unlucky size estimate by the strata estimator causing the algorithm to initially attempt set reconciliation, before switching to full set transmission. If the threshold between IBF and full set transmission is picked a bit too high

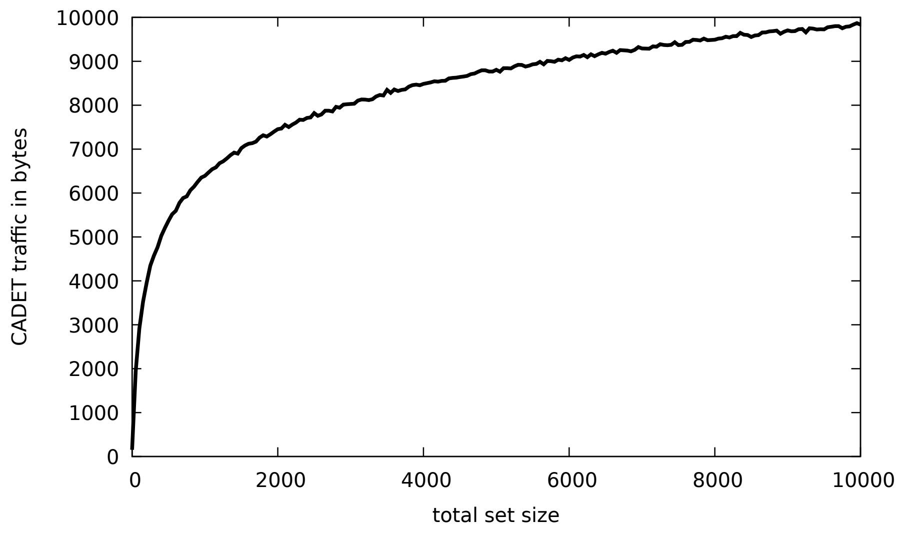

Fig. 2: CADET traffic for the SET service in relation to total set size. Average over five executions.

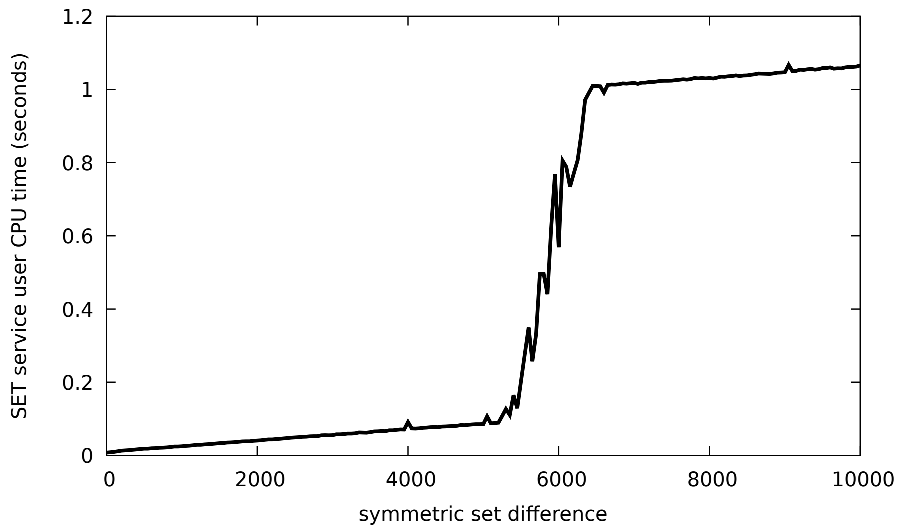

Fig. 3: CPU system time for the SET service in relation to symmetric set difference. Average over five executions.

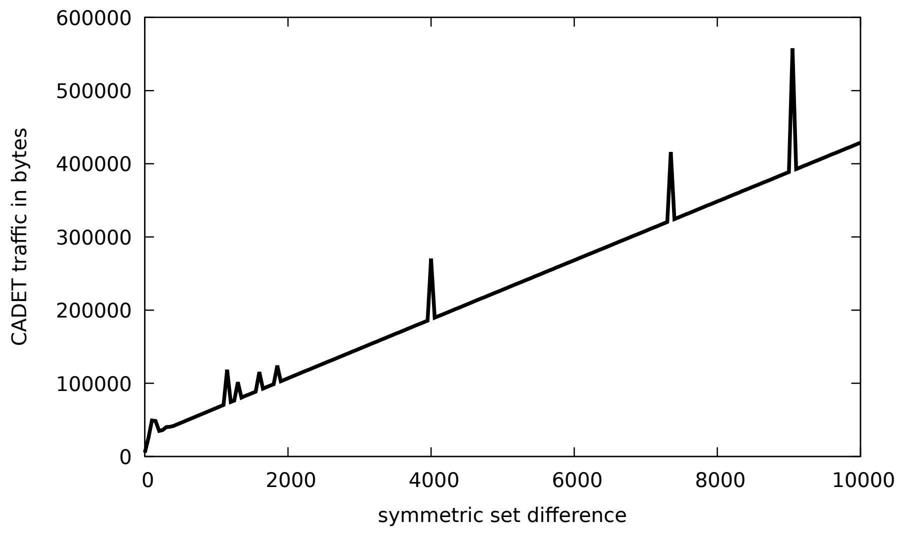

Fig. 4: CADET traffic for the SET service in relation to symmetric difference. Average over five executions.

and IBFs are sent slightly beyond the point where they are beneficial, the curve from the IBF transmission will peak above the one that represents the full set transmission. This is the case in the solid curve in Figure 5. Finally, the dotted curve shows the case where the threshold is picked too low, causing expensive full set transmission to occur when IBFs would have been more useful. Here, we also see a lucky case of underestimating the size of the difference. We note that given the size of an IBF entry, the average size of a set element and an estimate of the size overlap, near-perfect thresholds (instead of the 50%-heuristic we described earlier) can be trivially computed.

5.2 Byzantine set consensus

For our experiments with the BSC implementation, all ordinary peers start with the same set of elements; different sets would only affect the all-to-all union phase of the protocol which does pairwise set reconciliation, resulting in increased bandwidth and CPU consumption proportional to the set difference as shown in the previous section.

As expected, traffic increases cubically with the number of peers when no malicious peers are present (Figure 6). Most of the CPU time (Figure 7) is taken up by CADET, which uses expensive cryptographic operations [36]. Since we ran the experiments on a multicore machine, the total runtime follows the same pattern as the traffic (Figure 8).

We now consider the performance implications from the presence of malicious peers. Figures 10 and Figure 11 show that bandwidth and runtime

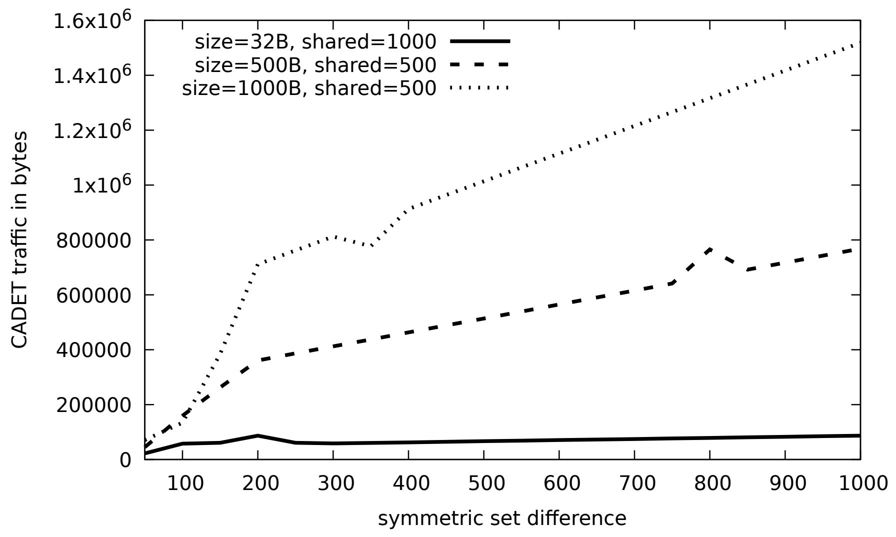

Fig. 5: CADET traffic for the SET service in relation to symmetric difference at the boundary between IBF and full set transmission. Note that we did cherry-pick runs for this graph. Our goal is to illustrate how the curves evolve with regard to different thresholds between IBF and full set transmission. We also wanted to show how significant deviations in set difference estimates generated by the strata estimator can have a minor impact on performance.

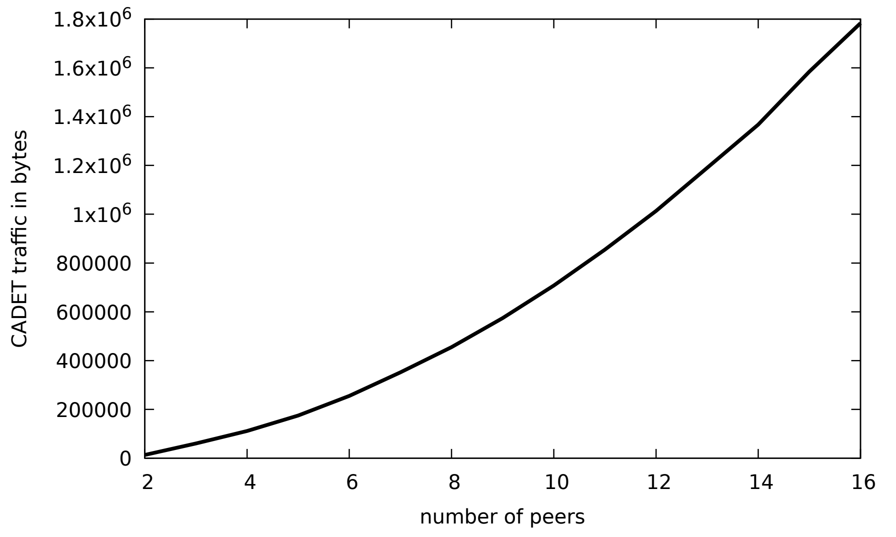

Fig. 6: CADET traffic for BSC per peer for 100 elements and only correct peers. Average over five executions.

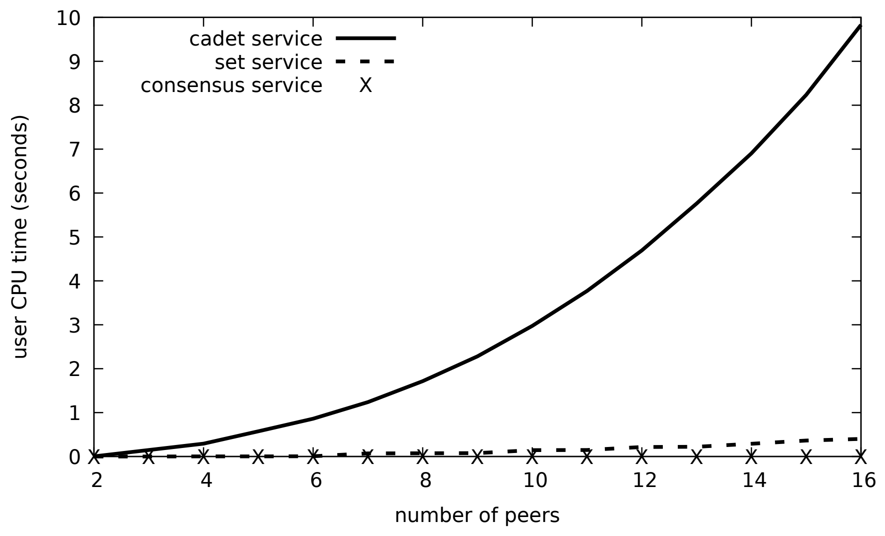

Fig. 7: CPU of BSC for 100 elements of 64 bytes and only correct peers. Average over five executions.

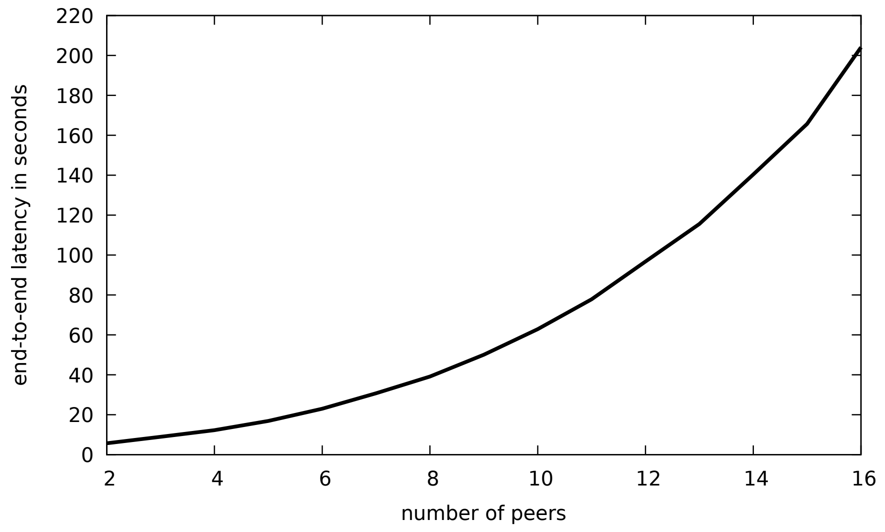

Fig. 8: Runtime of BSC for 100 elements of 64 bytes and only correct peers. Average over five executions.

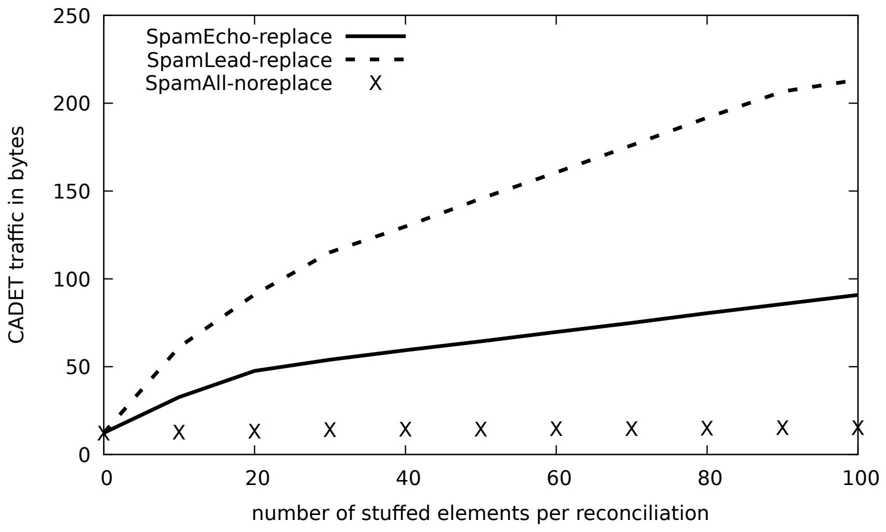

Fig. 9: CADET traffic for BSC on 100 elements of 64 bytes and one malicious peer with the indicated mode. Average over five executions.

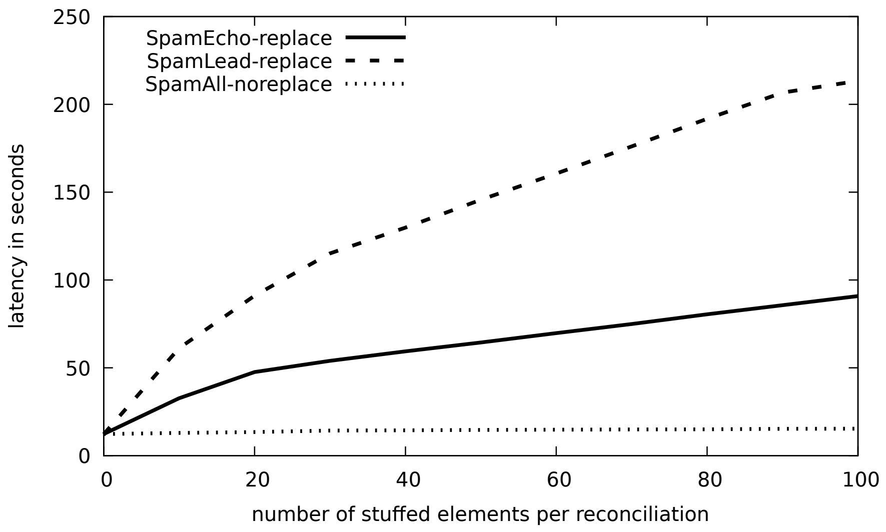

Fig. 10: Latency for BSC with 4 peers on 100 elements of 64 bytes and one malicious peer with the indicated mode. Average over five executions.

increase linearly with the additional elements malicious peers can exclusivly supply, in contrast to the sub-linear growth for the non-Byzantine case (Figure 2).

Figure 11 highlights how the different attack strategies impact the number of additional elements that were received during set reconciliations: The number of stuffed elements for the “SpamEcho” behavior is significantly larger than for “SpamLead”, since multiple ECHO rounds are executed for one LEAD round, and the number of stuffed elements is fixed per reconciliation. When malicious peers add extra elements during the LEAD round, the effect of that is amplified, since all correct receivers have to re-distribute the additional elements in the ECHO/CONFIRM round. Even though adding elements in the LEAD round requires the least bandwidth from the leader the effect on traffic and latency is the largest (see Figures 9 and 10).

As expected, when the number of stuffed elements is limited to a fixed set, the effect on the performance is limited (“SpamAll-noreplace” in Figures 9, 10, 11).

6 Opportunities for further improving BSC

We now discuss some of the key limitations of the current implementation and, how it could be optimised further.

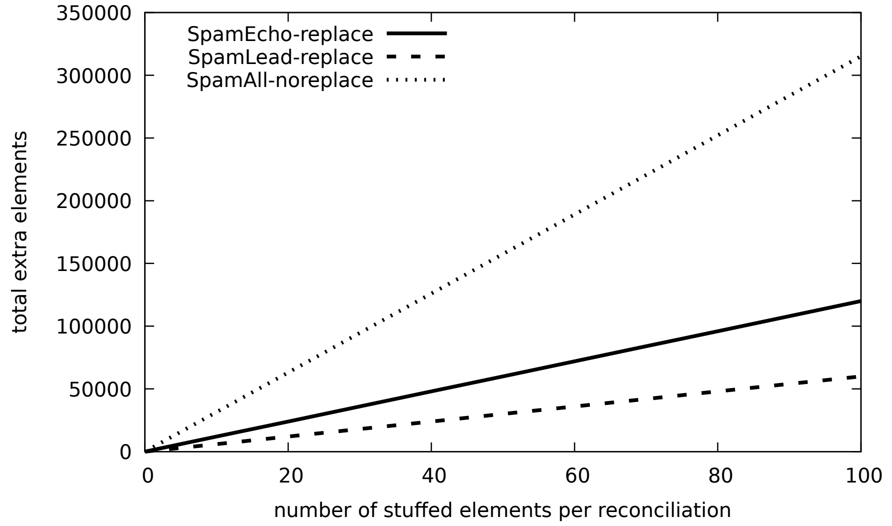

Fig. 11: Total number of extra elements received by each peer for BSC on 100 elements of 64 bytes and one malicious peer with the indicated mode. Average over five executions.

6.1 Extension to partial synchrony

The prototype used in the evaluation only works in the synchronous model. It would be trivial to extend it to the partially synchronous model with synchronous clocks by using the same construction as BPFT [2], namely retrying the protocol with larger round timeouts (usually doubled on each retry) when it did not succeed.

It might be worthwhile to further investigate the Byzantine round synchronization protocols discovered independently by Attya and Dolev [39] as well as Dwork, Lynch and Stockmeyer [8]. Running a Byzantine clock synchronization protocol interleaved with consensus protocol might lead to a protocol with lower latency, since the timeouts are dynamically adjusted instead of being increased for each failed iteration.

6.2 Persistent data structures

Both the SET and CONSENSUS service have to store many variations of the same set when faulty peers elide or add elements. While the SET service API already supports lazy copying, the underlying implementation is inefficient and based on a log of changes per element with an associated version number. It might be possible to reduce memory usage and increase performance of the element storage by using data structures that are more well suited, such as the persistent data structures described by Okasaki [40].

6.3 Fast dissemination

Recall that in order to be included in the final set, an element must be sent to at least $t+1$ peers, so that at least one correct peer will receive the element. In applications of set-union consensus such as electronic voting, the effort to the client should be minimized, and thus in practice elements might be sent only to $t+1$ peers, which would lead to large initial symmetric differences between peers.

A possible optimization would be to add another dissemination round that only requires $n \log_2 n$ reconciliations to achieve perfect element distribution when only correct peers are present. The n^2 reconciliations that follow will consequently be more efficient, since no difference has to be reconciled when all peers are correct. In the presence of faulty peers, the optimization adds more overhead due to the additional dissemination round.

More concretely, in the additional dissemination round the peers reconcile with their 2^ℓ-th neighbour (for some arbitrary, fixed order on the peers) in the ℓ-th subround of the dissemination round. After $\lceil \log_2 \rceil$ of these subrounds, the elements are perfectly distributed as long as every peer passed along their current set correctly.

7 Application to SMC

Secure multiparty computation (SMC) is an area of cryptography that is concerned with protocols that allow a group of peers $\mathcal{P} = P_1, \ldots, P_n$ to jointly compute a function $y = f(x_1, \ldots, x_n)$ over private input values $x_1, \ldots, x_n$ without using a trusted third party [41]. Each peer P_i contributes its own input value x_i, and during the course of the SMC protocol, P_i ideally only learns the output y, but no additional information about the other peers' input values. Applications of SMC include electronic voting, secure auctions and privacy-preserving data mining.

SMC protocols often assume a threshold $t < n$ on the amount of peers controlled by an adversary, which is typically either *honest-but-curious* (i.e. tries to learn as much information as possible but follows the protocol) or *actively malicious*. The actively malicious case mandates the availability of Byzantine consensus as a building block [42].[6]

In practical applications, the inputs typically consist of sets of values that were given to the peers $\mathcal{P}$ by external clients: In electronic voting protocols the peers need to agree on the set of votes; with secure auctions, the peers need to agree on bids, and so on.

In this section, we focus on one practical problem, namely electronic voting. We show how BSC is useful at multiple stages of the protocol, and discuss how our approach differs from existing solutions found in the literature.

7.1 Bulletin board for electronic voting

The *bulletin board* is communication abstraction commonly used for electronic voting [43, 44]. It is a stateful, append-only channel that participants of the election can post messages to. Participants of the election identify themselves with a public signing key and must sign all messages that they post to the bulletin board. The posted messages are publicly available to facilitate independent auditing of elections.

Existing work on electronic voting either does not provide a Byzantine fault-tolerant bulletin board in the first place [45] and instead relies on trusted third parties, or suggests the use of state machine replication [5].

Some of the bulletin board protocols surveyed by Peters [44] use threshold signatures to certify to the voter that the vote was accepted by a sufficiently large fraction of the peers that jointly provide the bulletin board service. With a naive approach, the message that certifies acceptance by t peers is the concatenation of the peers' individual signatures and thus $O(t)$ bits large. Threshold signature schemes allow smaller signatures, but at the expense of a more

[6] An attempt has been made to relax the definition of SMC to alleviate this requirement, resulting in a weaker definition that includes non-unanimous *aborts* as a possible result [41]. This definition is mainly useful in scenarios without an non-faulty 2/3 majority, where Byzantine consensus is not possible in the asynchronous model [8].

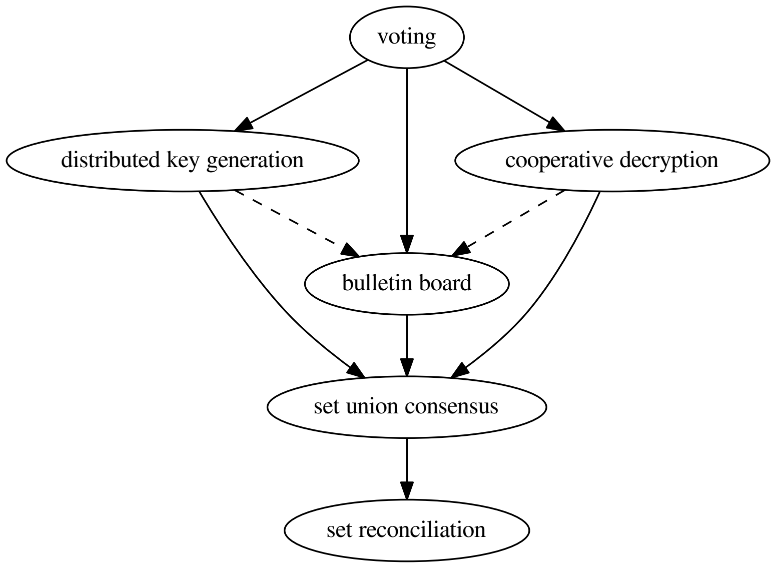

Fig. 12: Relation of different SMC protocols and communication primitives in GNUnet. Dashed arrows indicate optional dependencies.

complex protocol. Since the number of peers is typically not very large, a linear growth in t is acceptable, which makes the simple scheme sufficient for practical implementations.

It is easy to implement a variant of the bulletin board with set-union consensus. In contrast to traditional bulletin boards, this variant has *phases*, where posted messages are only visible after the group of peers have agreed that a phase is concluded. The concept of phases maps well to the requirements of existing voting protocols. Every phase is implemented with one set-union consensus execution. To guarantee that a message is posted to the bulletin board, it must be sent to at least one correct peer from the group of peers that jointly implements the bulletin board.

7.2 Distributed threshold key generation and cooperative decryption

Voting schemes as well as other secure multiparty computation protocols often rely on threshold cryptography [46]. The basic intuition behind threshold cryptography is that some operations—such as signing a message or decrypting a ciphertext—should only succeed if a large enough fraction of some group of peers cooperate. Typically the public key of the threshold cryptosystem is publicly known, while the private key is not known by any entity but reconstructible from the shares that are distributed among the participants, for example with Shamir's secret sharing scheme [47].

Generating this shared secret key either requires a trusted third party, or a protocol for distributed key generation [48, 49]. The former is undesirable for most practical applications since it creates a single point of failure.

In a distributed key generation protocol, each peer contributes a number of *pre-shares*. The peers agree on the set of pre-shares and each peer re-combines them in a different way, yielding the shares of the private threshold key.

In the key generation protocol used for the Cramer et al. voting scheme, the number of pre-shares that need to be agreed upon is quadratic in the number of peers. Every peer needs to know every pre-share, even if it is not required by the individual peer for reconstructing the share, since the pre-shares are accompanied by non-interactive proofs of correctness. Thus the number of values that need to be agreed upon is quadratic in the number of peers, which makes the use of set-union consensus attractive compared to individual agreement.

Even though the pre-shares can be checked for correctness, Byzantine consensus on the set of shares is still necessary for the case when a malicious peer submits a incorrect share to only some peers. Without Byzantine consensus, different correct recipients might exclude different peers, resulting in inconsistent shares.

Similarly, when a message that was encrypted with the threshold public key shall be decryped, every peer contributes a *partial decryption* with a proof of correctness. While the set of partial decryptions is typically linear in the number of peers, set-union consensus is still a reasonable choice here, this way the whole system only needs one agreement primitive.

7.3 Electronic voting with homomorphic encryption

Various conceptually different voting schemes use homomorphic encryption; we look as the scheme by Cramer et al. [5] as a modern and practical representative. A fundamental mechanism of the voting scheme is that a set of voting authorities $A_1, \ldots, A_n$ establish a threshold key pair that allows any entity that knows the public part of the key to encrypt a message that can only be decrypted when a threshold of the voting authorities cooperate. The homomorphism in the cryptosystem enables the computation of an encrypted tally with only the ciphertext of the submitted ballots. Ballots represent a choice of one candidate from a list of candidate options. The validity of encrypted ballot is ensured by equipping them with a non-interactive zero-knowledge proof of their validity.

It is assumed that the adversary is not able to corrupt more than $1/3$ of the authorities. The voting process itself is then facilitated by all voters encrypting their vote and submitting it to the authorities. The encrypted tally is computed by every authority and then cooperatively decrypted by the authorities and published. Since correct authorities will only agree to decrypt the final tally and not individual ballots, the anonymity of the voter is preserved. For the voting scheme to work correctly, all correct peers must agree on exactly the same set of ballots before the cooperative decryption process starts, otherwise the decryption of the tally will fail.

Using BSC for this final step to agree on a set of ballots again makes sense, as the number of ballots is typically much larger than the number of authorities. Figure 12 summarizes the various ways how BSC and is used in our implementation [50] of Cramer-style [5] electronic voting.

7.4 Other applications of BSC

Bitcoin [51] has gained immense popularity over the past few years. Bitcoin solves a slight variation of Byzantine consensus without strong validity [52, 53]. Given that a block in Bitcoin is basically just a set of (valid) transactions, BSC could be used to efficiently achieve agreement between participants about the next transaction group. Here, the most natural application would be to use BSC to improve the efficiency of proof-of-stake incentivized peers running BFT consensus in Cosmos [54].

8 Conclusion, Ongoing and Future Work

Given m ballots, n authorities, f Byzantine faults and k ballots exclusively available to the adversary, voting with BSC achieves a complexity of $O(mn + (f + k)n^3)$, which in practice is better than the $O(mn^2)$ complexity of using SMR as m is usually significantly larger than n. Equivalent arguments hold for other applications requiring consensus over large sets. Furthermore, BSC remains advantageous in the absence of Byzantine failures, and the bounded set reconciliation makes it particularly efficient at handling various non-Byzantine faults.

To ensure these performance bounds, BSC requires a bounded variant of Eppstein's set reconciliation protocol that ensures that individual steps in the protocol cannot consume unbounded amounts of bandwidth. We are currently applying bounded set reconciliation in related domains, as any set reconciation can be made more robust if the complexity of the operation is bounded. For example, the GNU Name System [55] can use bounded set reconcilation when gossiping sets of key revocation sets. Here, the use of bounded set reconciliation protects the key revocation protocol against denial-of-service attacks where an attacker might have previously sent excessively large IBFs or retransmitted known revocation messages already known to the recipient. The result is an efficient and resilient method for disseminating key revocation data.

In future work, it would be interesting to apply bounded set reconciliation to Byzantine consensus protocols that are more efficient than the simple gradecast consensus. It would also be interesting to experimentally compare bulletin boards using BSC with those using traditional replicated state machines.

9 Declarations

9.1 Ethics approval and consent to particpate

Not applicable.

9.2 Consent for publication

Not applicable.

9.3 Availability of data and material

The software used for the experiments is available in the public Git repository [56]. Raw experimental data can be made available together with the publication after acceptance (but is not suitable for the open data repositories listed in the guidelines). For example, it could simply be embedded with the PDF (it is not that big).

9.4 Competing interests

None.

9.5 Funding

This work benefits from the financial support of the Brittany Region (ARED 9174) and the Renewable Freedom Foundation.

9.6 Author's contributions

Both authors contributed to the design, implementation and the writing of the article. Florian Dold ran the experiments.

9.7 Acknowledgements

We thank Jeffrey Burdges and the anonymous reviewers of ARES 2016 for comments on an earlier draft of this paper.

References

1. M. Fitzi, M. Hirt, in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing* (ACM, 2006), pp. 163–168
2. M. Castro, B. Liskov, et al., in *OSDI*, vol. 99 (1999), vol. 99, pp. 173–186
3. R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, in *ACM SIGOPS Operating Systems Review*, vol. 41 (ACM, 2007), vol. 41, pp. 45–58
4. A. Clement, E.L. Wong, L. Alvisi, M. Dahlin, M. Marchetti, in *NSDI*, vol. 9 (2009), vol. 9, pp. 153–168
5. R. Cramer, R. Gennaro, B. Schoenmakers, European transactions on Telecommunications **8**(5), 481 (1997)
6. P. Bogetoft, D.L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J.D. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, et al., in *Financial Cryptography and Data Security* (Springer, 2009), pp. 325–343
7. M. Ben-Or, D. Dolev, E.N. Hoch, arXiv preprint arXiv:1007.1049 (2010)
8. C. Dwork, N. Lynch, L. Stockmeyer, Journal of the ACM (JACM) **35**(2), 288 (1988)
9. M. Castro, B. Liskov, ACM Transactions on Computer Systems (TOCS) **20**(4), 398 (2002)
10. D. Eppstein, M.T. Goodrich, F. Uyeda, G. Varghese, in *ACM SIGCOMM Computer Communication Review*, vol. 41 (ACM, 2011), vol. 41, pp. 218–229
11. L. Lamport, R. Shostak, M. Pease, ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3), 382 (1982)
12. M.J. Fischer, N.A. Lynch, A lower bound for the time to assure interactive consistency. Tech. rep., DTIC Document (1981)
13. R. De Prisco, D. Malkhi, M. Reiter, Parallel and Distributed Systems, IEEE Transactions on **12**(1), 7 (2001)
14. N. Malpani, J.L. Welch, N. Vaidya, in *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications* (ACM, 2000), pp. 96–103
15. M.J. Fischer, N.A. Lynch, M. Merritt, Distributed Computing **1**(1), 26 (1986)
16. G. Neiger, Information Processing Letters **49**(4), 195 (1994)
17. M.J. Fischer, N.A. Lynch, M.S. Paterson, Journal of the ACM (JACM) **32**(2), 374 (1985)
18. M.K. Aguilera, in *Replication* (Springer, 2010), pp. 59–72
19. P.N. Feldman, Optimal algorithms for byzantine agreement. Ph.D. thesis, Massachusetts Institute of Technology (1988)
20. P. Feldman, S. Micali, in *Proceedings of the twentieth annual ACM symposium on Theory of computing* (ACM, 1988), pp. 148–161
21. A. Mostefaoui, H. Moumen, M. Raynal, in *Proceedings of the 2014 ACM symposium on Principles of distributed computing* (ACM, 2014), pp. 2–9
22. C. Cachin, K. Kursawe, V. Shoup, Journal of Cryptology **18**(3), 219 (2005)
23. J. Aspnes, Journal of the ACM (JACM) **45**(3), 415 (1998)
24. E. Syta, P. Jovanovic, E.K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M.J. Fischer, B. Ford. Scalable bias-resistant distributed randomness. Cryptology ePrint Archive, Report 2016/1067 (2016). `http://eprint.iacr.org/2016/1067`
25. R. Guerraoui, M. Hurfinn, A. Mostéfaoui, R. Oliveira, M. Raynal, A. Schiper, in *Advances in Distributed Systems* (Springer, 2000), pp. 33–47
26. M.K. Reiter, in *Theory and Practice in Distributed Systems* (Springer, 1995), pp. 99–110
27. K.P. Kihlstrom, L.E. Moser, P.M. Melliar-Smith, in *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, vol. 3 (IEEE, 1998), vol. 3, pp. 317–326
28. D. Dolev, C. Dwork, L. Stockmeyer, Journal of the ACM (JACM) **34**(1), 77 (1987)
29. A. Miller, Y. Xia, K. Croman, E. Shi, D. Song, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (ACM, New York, NY, USA, 2016), CCS '16, pp. 31–42. DOI 10.1145/2976749.2978399. URL `http://doi.acm.org/10.1145/2976749.2978399`

30. M. Abd-El-Malek, G.R. Ganger, G.R. Goodson, M.K. Reiter, J.J. Wylie, ACM SIGOPS Operating Systems Review **39**(5), 59 (2005)
31. P.L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, M. Vukolić, ACM Trans. Comput. Syst. **32**(4), 12:1 (2015). DOI 10.1145/2658994. URL `http://doi.acm.org/10.1145/2658994`
32. Y. Minsky, A. Trachtenberg, R. Zippel, Information Theory, IEEE Transactions on **49**(9), 2213 (2003)
33. B.H. Bloom, Communications of the ACM **13**(7), 422 (1970)
34. M. Mitzenmacher, R. Pagh, arXiv preprint arXiv:1311.2037 (2013)
35. The GNUnet Project. `https://gnunet.org/`
36. B. Polot, C. Grothoff, in *Ad Hoc Networking Workshop (MED-HOC-NET), 2014 13th Annual Mediterranean* (IEEE, 2014), pp. 71–78
37. S. Tarkoma, C.E. Rothenberg, E. Lagerspetz, Communications Surveys & Tutorials, IEEE **14**(1), 131 (2012)
38. S.H. Totakura, Large scale distributed evaluation of peer-to-peer protocols. Master's thesis, Technische Universität München, Garching bei München (2013)
39. C. Attiya, D. Dolev, J. Gil, in *Proceedings of the third annual ACM symposium on Principles of distributed computing* (ACM, 1984), pp. 119–133
40. C. Okasaki, *Purely functional data structures* (Cambridge University Press, 1999)
41. S. Goldwasser, Y. Lindell, Journal of Cryptology **18**(3), 247 (2005)
42. J. Saia, M. Zamani, in *SOFSEM 2015: Theory and Practice of Computer Science* (Springer, 2015), pp. 24–44
43. J.D.C. Benaloh, *Verifiable secret-ballot elections* (Yale University. Department of Computer Science, 1987)
44. R. Peters, A secure bulletin board. Master's thesis, Technische Universiteit Eindhoven (2005)
45. B. Adida, in *USENIX Security Symposium*, vol. 17 (2008), vol. 17, pp. 335–348
46. Y.G. Desmedt, European Transactions on Telecommunications **5**(4), 449 (1994)
47. A. Shamir, Communications of the ACM **22**(11), 612 (1979)
48. P.A. Fouque, J. Stern, in *Public Key Cryptography* (Springer, 2001), pp. 300–316
49. T.P. Pedersen, in *Advances in CryptologyEUROCRYPT91* (Springer, 1991), pp. 522–526
50. F. Dold, Cryptographically secure, distributed electronic voting. Bachelor's thesis, Technische Universität München (2014)
51. S. Nakamoto, Consulted **1**(2012), 28 (2008)
52. A. Miller, J.J. LaViola Jr, Retrieved from Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin (2014)
53. J. Garay, A. Kiayias, N. Leonardos, in *Advances in Cryptology-EUROCRYPT 2015* (Springer, 2015), pp. 281–310
54. J. Kwon, E. Buchman. Cosmos: A network of distributed ledgers. `https://cosmos.network/whitepaper` (2016)
55. M. Wachs, M. Schanzenbach, C. Grothoff, in *13th International Conference on Cryptology and Network Security (CANS 2014)* (2014), pp. 127–142
56. The GNUnet Project Git Repository. `git://gnunet.org/git/gnunet`