Communication-Efficient BFT Protocols Using Small Trusted Hardware to Tolerate Minority Corruption

Sravya Yandamuri[1], Ittai Abraham[2], Kartik Nayak[1], and Michael K. Reiter[1]

[1] Duke University - {sravya.yandamuri,kartik.nayak,michael.reiter}@duke.edu
[2] VMware Research - iabraham@vmware.com

Abstract. Agreement protocols for partially synchronous or asynchronous networks tolerate fewer than one-third Byzantine faults. If parties are equipped with trusted hardware that prevents equivocation, then fault tolerance can be improved to fewer than one-half Byzantine faults, but typically at the cost of increased communication complexity. In this work, we present results that use small trusted hardware without worsening communication complexity assuming the adversary controls a fraction of the network that is less than one-half. Our results include a version of HotStuff that retains linear communication complexity in each view and a version of the VABA protocol with quadratic communication, both leveraging trusted hardware to tolerate a minority of corruptions. Our results use expander graphs to achieve efficient communication in a manner that may be of independent interest.

1 Introduction

Byzantine fault tolerant (BFT) consensus is an important problem in distributed computing. It has received revived interest as the foundation of decentralized ledgers or blockchains. The goal of BFT consensus is for a set of parties to agree on a value (or a sequence of values) even if a fraction of the parties are Byzantine (malicious). To rule out trivial solutions, these protocols additionally need to satisfy a validity constraint which, depending on the setting, is a function of the input of a designated party or all parties or external clients.

The number of faults tolerated by a BFT protocol depends on the network assumptions between parties, the use of cryptography, and other assumptions. In particular, it is known that to maintain safety when the system is asynchronous, without additional assumptions, one cannot tolerate one-third or more Byzantine faults [19]. However, tolerating less than one-third Byzantine faults may not be enough for some applications. There are two known approaches to increase this fault threshold. The first approach is to give up safety in asynchrony. One can tolerate less than one-half Byzantine faults by assuming synchrony (any message sent by an honest party reaches its destination within a bounded network delay) and some method to limit the ability of the adversary to simulate honest parties (for example assuming a PKI or proof-of-work) [20, 26, 4, 12, 18, 36, 29]. Protocols using synchrony increase the fault threshold by detecting equivocations (assuming signatures) and making deductions based on absence of messages from other parties (e.g., [41, 4]). The second approach lets the adversary delay messages but limits its ability to corrupt by assuming the existence of a *trusted hardware*. The adversary cannot tamper with this hardware even if it fully controls the node. At a high-level, the hardware provides non-equivocation guarantees, essentially transforming Byzantine failures to omission failures and hence improving the fault tolerance threshold to one-half (e.g., [19, 15, 25, 51]) in partial synchrony and asynchrony.

In this work, we focus on the use of *small trusted hardware* primitives to tolerate a minority Byzantine corruption and stay safe in asynchrony. Realizations of such primitives include TPMs [1] and YubiKey [47], and at a high level, they provide us with the abstraction of an "append-only log". Compared to trusted hardware instantiations such as Intel SGX [17], which are capable of computing arbitrary functions in a trusted manner, small trusted hardware primitives only allow operations such as appending to a (logical) log and *attesting* to the log contents, and they typically only have $O(1)$ registers as storage. The use of small trusted hardware has several advantages: (1) a smaller trusted computing base is likely to have fewer security vulnerabilities [39, 50, 53]; (2) much more widespread availability; (3) the ability for many different manufacturers to produce such devices. A noticeable drawback of small trusted hardware is that it cannot, for example, sign a threshold-signature or even a multi-signature [30, 15].

In this work, we investigate the *asymptotic communication complexity of protocols tolerating a minority corruption assuming a small trusted hardware in asynchronous or partially synchronous networks*. We measure communication complexity as the (expected) number of words that all honest parties send and receive. Given some $\epsilon > 0$ we define a *minority corruption adversary* as having control of at most $t \leq (\frac{1}{2} - \epsilon)n$ parties.

The study of trusted hardware to boost fault tolerance has been studied in the past by works such as A2M [15] and TrInc [30]. Those works improve the fault tolerance of PBFT [10] using small trusted hardware primitives. However, this comes at the expense of an $O(n^3)$ communication complexity per view for consensus among n parties. On the other hand, in the standard setting, recently, we have seen considerable progress in improving communication complexity of consensus protocols. In particular, HotStuff [54] achieves linear communication complexity per view under partial synchrony and VABA [5] achieves the optimal $O(n^2)$ communication complexity under asynchrony. A natural question is whether similar results hold under a minority corruption assuming trusted hardware. In this work, we answer these questions affirmatively although for a corruption threshold $t \leq (\frac{1}{2} - \epsilon)n$ for an arbitrarily small ϵ. Our work leaves open addressing the communication complexity with trusted hardware and optimal resilience. In the following, we describe our results and the key techniques used to achieve these results.

1.1 HotStuff-M: HotStuff with Minority Corruption

Our first result improves HotStuff to tolerate a $t \leq (\frac{1}{2} - \epsilon)n$ corruption while still retaining its linear communication complexity per view. In particular, we show the following result:

Theorem 1 (HotStuff-M, Informal). *For any $\epsilon > 0$, there exists a primary-backup based BFT consensus protocol with $O(n)$ communication complexity per view, consisting of n parties each of which has access to a small trusted hardware and $t \leq (\frac{1}{2} - \epsilon)n$ parties are Byzantine.*

HotStuff is a primary-backup protocol that progresses in a sequence of views, each having a designated leader (primary) and consisting of a sequence of phases. HotStuff routes all messages (votes) through the leader independent of whether the communication is within the view or across views, while keeping the message size $O(1)$ for a total of $O(n)$ communication per view. To achieve this, HotStuff crucially relies on threshold signatures to aggregate votes of individual parties into an $O(1)$-sized message; these signatures act as a proof for parties in subsequent phases/views to determine whether they should vote in that phase. Within a view, HotStuff maintains safety due to the fact that if a leader has a threshold signature for a given proposal, a majority of the honest parties voted for that proposal. By quorum intersection, and the fact that an honest party votes only once in a given phase, a conflicting proposal cannot have a valid threshold signature.

We improve the resilience of HotStuff from one-third to $\frac{1}{2} - \epsilon$ while keeping a total of $O(n)$ communication per view using small trusted hardware in a partially synchronous network. In the minority corruption model, the presence of a threshold signature on a proposal no longer implies that a majority of the honest parties voted for that proposal, and is therefore insufficient for safety against a Byzantine adversary. The key property that we need from the hardware is its ability to maintain an append-only log that can be used to provide a non-equivocation property, i.e., if the hardware produces a *signed attestation* at a given position in the log, then the party cannot produce a valid signed attestation for a different value at the same position. Thus, intuitively, if $\frac{n}{2} + 1$ parties attest to a value at a position, then no other value can have $\frac{n}{2} + 1$ attestations. However, while a party's attestation from its trusted hardware is sufficient for safety, receiving such proofs from $O(n)$ parties produces an $O(n)$-sized proof sent to the leader of the view. Since the leader uses this proof in a subsequent round, and these proofs cannot be compressed as they are not threshold signatures, this will grow the communication complexity to $O(n^2)$ per view.

Instead of sending the attestations directly to the leader, our solution relies on diffusing the attestations to a constant number of parties, called its *neighbors*. A party *votes* if it receives attestations from a threshold of its neighbors. This vote can be a threshold signature share, which can eventually be combined by the leader to an $O(1)$-sized voting proof. Why does this work? We connect parties to each other using a constant-degree expander graph. Informally, to send a (non-attested) vote, a party just needs to verify that a constant fraction of its neighbors have attested. The specific construction of our expander guarantees that if a small ϵn-sized fraction of honest parties have voted for a proposal, then a majority of the parties have attested to that proposal. Thus, if a leader receives votes from $t + \epsilon n = \frac{n}{2}$ parties, at least ϵn are honest, and they can vet attestations from a majority of parties; this guarantees safety within a view. To ensure liveness, the expander graph is also parameterized such that if all honest parties attest, more than $\frac{n}{2}$ parties vote. We note that expander graphs have been used in consensus protocols before [28, **?**,**?**], although only in the context of synchronous protocols and exploiting a different set of expander properties.

The above arguments do not suffice for safety across views. The key mechanism that ensures safety across views in HotStuff is that if an honest party commits a value v in a given view, they received a threshold signature on v from the previous phase of this view, indicating that a majority of the honest parties "locked" on v in this view. These locked parties will not vote for a different value in later views, and a $v' \neq v$ will never gain a threshold signature in a subsequent view (and will therefore not be committed by an honest party). In the minority corruption model, while our trusted hardware disallows appending different values at the same position (equivocation), we cannot enforce the conditions under which a Byzantine party appends a value to their log. We also cannot enforce that a Byzantine party presents the latest state of its log as necessary. This can potentially result in a safety or liveness violation; e.g., even if a party "locked" on v in a given view by attesting to v, it can present a state that does not involve this attestation. In the original HotStuff protocol, an honest leader waits to hear the value from the highest view in which parties have stored a value during the second phase of that view for liveness. Since, in the minority corruption model, a leader cannot wait to hear from a majority of the honest parties, it must rely on Byzantine parties to present the correct state of their logs. Of course, this could be fixed by requiring a party to always present the entire contents of the log in its trusted hardware, but the communication complexity would grow (unbounded) with the number of views. Instead, we use a combination of techniques including: multiple logs, one for each phase of the protocol ($O(1)$ total); tying log positions to view numbers; and using one attestation to present the end state of all logs. We elaborate on these techniques in Section 4 when we describe the protocol.

1.2 VABA-M: Validated Asynchronous Byzantine Agreement with Minority Corruption

Our second result improves the VABA protocol of Abraham et al. [5] to tolerate minority corruption while retaining its $O(n^2)$ communication complexity. We show the following result:

Theorem 2 (VABA-M, informal). *For any $\epsilon > 0$, there exists a validated asynchronous Byzantine Agreement protocol with $O(n^2)$ communication complexity consisting of n parties each of which has access to a small trusted hardware such that $t \leq (1/2 - \epsilon)n$ parties are Byzantine.*

The VABA protocol adapts HotStuff to an asynchronous network and incurs $O(n^2)$ communication complexity. At a high level, the protocol progresses in a sequence of views. In each view, in parallel, each party attempts to drive progress by acting as a leader in a "proposal promotion". After $n - t$ proposal promotions have completed, the parties elect one leader uniformly at random, and adopt the progress from the leader's proposal promotion instance during the view-change step. Depending on whether the leader completed its proposal promotion, parties may decide at the end of the view or repeat the process in another view.

At first sight, it appears that the ideas used for HotStuff-M should directly follow. However, there are two key challenges in augmenting VABA to tolerate a minority corruption, both of which are related to the amount of storage in the small trusted hardware.

First, in HotStuff-M, only $O(1)$ logs were used. For each log, the latest state can be maintained in the hardware, and attestations of previous positions can be stored externally by a party. With an asynchronous protocol, since there are n proposal promotion instances, a straightforward translation requires $O(n)$ number of logs. When reduced to $O(1)$ logs, each log needs to maintain $O(n)$ amount information on the hardware. The challenge though is that (i) in an asynchronous protocol, messages arrive in an arbitrary order across proposal promotion instances, and (ii) the hardware can only append to a log. This makes it hard to use any mapping between a position on the log and a proposal promotion instance. Our solution crucially relies on the fact that all parties have the same neighbors across proposal promotion instances, and thus, even if values from proposal promotion instances are appended arbitrarily, they can perform the necessary validation across all instances in a non-blackbox manner.

Second, the view-change step requires every party to share the "progress" from the elected leader's proposal promotion instance to all parties. However, due to the concern described earlier, only a parties' neighbors can validate whether it used the trusted hardware correctly. To make matters worse, a party or its neighbors can be Byzantine. Fortunately, since parties are connected using an expander graph, we can bound the number of honest and Byzantine parties with a majority of Byzantine neighbors. By a careful analysis, we can ensure the delivery of the latest state of leader's proposal promotion instance to all parties. We describe our solution in detail in Section 5.

2 Model and Preliminaries

We consider n parties $p_i, \ldots, p_n$ connected by a reliable, authenticated all-to-all network, where up to $t \leq (1/2 - \epsilon)n$ parties may be corrupted by an adversary for $\epsilon > 0$. The corrupted parties are Byzantine and may behave arbitrarily. All the correct (honest) parties follow the protocol specification. Depending on our construction, we consider either an asynchronous network, where a message sent by one correct party to another arrives eventually but with arbitrary delay, or a partially synchronous network, where after an unknown period of time called Global Stabilization Time (GST), every message will arrive within a known bounded delay. We solve the validated Byzantine Agreement problem:

Definition 1 (Validated Byzantine Agreement). *A protocol solves validated Byzantine agreement among n parties tolerating a maximum of t faults if it satisfies the following properties:*

(Agreement/Safety) *If any two honest parties output values v and v', then $v = v'$.*

(Validity) *If an honest party outputs v, then v is an externally valid value, i.e., ext-valid(v) = true.*

(Termination/Liveness under asynchrony) *If all honest parties start with an externally valid value and all messages sent by the honest parties eventually arrive, then all honest parties will output a value.*

(Termination/Liveness under partial synchrony) *If all honest parties start with an externally valid value, then after GST, all honest parties will output a value within a bounded time.*

Following Cachin et al. [9], the definition has an external validity property. Such a property can be useful in the context of state machine replication (SMR) where ext-valid(v) captures validity of a command sent by a client. We assume that each party has access to a small trusted hardware (described in Section 2.2). In addition, for communication efficiency, some of the messages are sent by the parties through an expander graph. We describe the properties needed from the expander graph in Section 2.3. We measure communication complexity as the number of *words* that all honest parties send and receive; each word is $O(\kappa)$ bits long where κ is a security parameter. We also assume all the messages sent by parties are signed using a threshold signature scheme described in Section 2.4.

2.1 Small Trusted Hardware

In this section, we introduce the abstraction of a *small trusted hardware* with $O(1)$ storage. As described in the introduction, the motivation behind the minimality is to allow for the existence of multiple hardware units by different vendors. However, at the same time, the hardware provides a *non-equivocation* capability. Such hardware units have been considered in prior works such as A2M [15], TrInc [30], etc. While the exact interface for the trusted hardware in these works differ, their capabilities are similar and supported by existing hardware modules such as Trusted Platform Modules (TPMs) and YubiKey [47, 1]. Without loss of generality, we assume the existence of a functionality similar to that of A2M [15].

Hardware state and interfaces. The trusted hardware provides a party with a set of append-only logs (denoted *log*) that can only be modified by the party's trusted hardware component. The functionality is shown in Figure 1. Each log within a single party's trusted component has its own identifier (denoted id) and includes a counter (denoted c_{id}) that starts from 0 and is incremented for each entry that is appended to the log. The trusted hardware guarantees that the party cannot modify the information stored in any position of the log. We use the notation $\langle\cdot\rangle_{K_{priv}}$ to denote that an attestation is signed using the private key of the trusted hardware component. To differentiate between a signature from a hardware device and a signature from the party holding it, we always refer to the former signature as an attestation.

2.2 Small Trusted Hardware Abstraction

Algorithm 1 Trusted Hardware Functionality.

```
1: $(K_{pub}, K_{priv})$: public-private key pair associated with the hardware device
2: $C$: monotonic counter representing the number of logs maintained by the hardware
3: $log$: list of logs indexed by $id$; each log is an array indexed by sequence number
4: $c_{id}$: monotonic counter representing the length of log indexed by $id$
5:
6: function CreateLog()
7:     Increment $C$, initialize empty log with $id := C$, $c_{id} := 0$; return $id$
8:
9: function Append($id, c_{new}, x$)
10:     if $id \leq C$:
11:         if $c_{new} = \perp$: Increment $c_{id}$, $log[id][c_{id}] := x$;
12:         if $c_{new} > c_{id}$: set $c_{id} := c_{new}$, $log[id][c_{id}] := x$;
13:         return Lookup($id, c_{id}$)
14:
15: function Lookup($id, s$)
16:     if $id \leq C$ and $s \leq c_{id}$: return $\langle \text{LOOKUP}, id, s, log[id][s]\rangle_{K_{priv}}$
17:
18: function End($id, z$)
19:     if $id \leq C$: return $\langle \text{END}, id, c_{id}, log[id][c_{id}], z\rangle_{K_{priv}}$
20:
21: function Counters(z)
22:     return $\langle \text{HEAD}, \bigcup_{id<C} \{(id, c_{id})\}, z\rangle_{K_{priv}}$
```

The hardware provides us with the following four functions. The Append(id, c_{new}, x) interface is used to append the value x to the log identified by id. If $c_{new} = \perp$, the functionality increments the counter of the log and inserts x into the position of the log. Otherwise, it appends to position c_{new} if c_{new} is strictly higher than the current log position. The Lookup(id, s) and End(id, z) interfaces are used to obtain an attestation of log with identifier id for the value stored at position s and the last position respectively. Finally, the trusted hardware provides us with a Counters(z) interface which returns the counter value of each log at the point when it is called. The nonce z is used to ensure freshness of an attestation. In our use of the trusted hardware interface later, we simply omit mentions of the nonce when it's not used. To simplify the description, we imagine that the hardware stores the entire set of $O(1)$ logs. In reality, the hardware only needs to maintain the end state of a log; a party can always store the attestations at different positions separately.

If party p_i calls Append($q, \perp, x$), then it should receive $\langle \text{LOOKUP}, q, s, x\rangle_{K_{priv}}$ in response, for some log position s. p_i can forward this response—or another copy, obtained by invoking LOOKUP(q, s)—to another party p_j to prove that p_i added value x to its log q at position s. Since the hardware interface only allows appending to the log, p_j can be assured that no other value can be attested at position s of log q. The use of the End(q, z) function is similar, with the addition that p_j passes a random nonce z to p_i that will be included in the attestation to prove that it is fresh.

2.3 Expander Graphs

Expander graphs are sparse graphs with a high degree of connectivity between groups of nodes. We call a node connected to a node p_i as its neighbor and denote the set of neighbors of p_i by $\rho(i)$. We describe the expander graph properties we need in this section and prove them in Appendix A.

Definition 2. *An (n, α, β)-expander graph, denoted $G_{n,\alpha,\beta}$, where $0 < \beta < 1$ and $\alpha < \beta$, is a graph with n vertices such that every set of αn vertices has at least βn unique neighbors.*

Lemma 1. *There exists a d-regular graph $G_{n,\epsilon,(1-\frac{\epsilon}{c})}$ for sufficiently large n and positive constants $0 < \epsilon < \frac{1}{2}$ and $c > 2$ such that:*

1. *For any set S of $(\frac{1}{2} + \epsilon)n$ nodes, there exists a set Q of more than $\frac{n}{2}$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S.*
2. *For any partition of its nodes into blocks T and Q where $|T| = (\frac{1}{2} - 2\epsilon)n$ and $|Q| = (\frac{1}{2} + 2\epsilon)n$, there exists a set $T' \subseteq T$, $|T'| > (\frac{1}{2} - 3\epsilon)n$, such that each node in T' has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in Q*
3. *For any set S of $(\frac{1}{2} + 2\epsilon)n$ nodes, there exists a set Q of more than $(\frac{1}{2} + \epsilon)n$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S*
4. *For any set S of ϵn nodes, and any sets $\{S_i\}_{i \in S}$ where $S_i \subset \rho(i)$ and $|S_i| = (\frac{1}{2} + \frac{\epsilon}{2})d$, the set $U = \bigcup_{i \in S} S_i$ satisfies $|U| > \frac{n}{2}$.*

2.4 Threshold sigantures

In addition to (and separate from) the signatures generated by hardware modules, we make use of a k out of l threshold signature scheme [45] for $k = \frac{n}{2} + 1$ and $l = n$, i.e., $\frac{n}{2} + 1$ parties must participate in order to create a valid threshold signature. We use the following interface:
- threshold-sign$_i(m)$: produces signature share produced by p_i on message m.
- share-validate(m, s_j, pk_j): validate signature share s_j produced by p_j on m.
- threshold-combine(m, S): combine a set S of signature shares from distinct parties for message m to an $O(1)$-sized signature where $|S| \geq k$ and share-validate$(m, s_j, pk_j) = \text{true}, \forall s_j \in S$.
- threshold-verify(m, σ): returns true if σ was a result of computing threshold-combine(m, S) where $|S| \geq k$ and share-validate$(m, s_j, pk_j) = \text{true}, \forall s_j \in S$.

3 $(\frac{n}{2} + 1)$-Provable-Broadcast

In this section, we present a core broadcast primitive that will enable protocols to tolerate up to $(\frac{1}{2} - \epsilon)n$ Byzantine faults for any $0 < \epsilon < \frac{1}{2}$ when every party is equipped with a trusted hardware component as described in Section 2.2. In subsequent sections, we will show how $(\frac{n}{2} + 1)$-Provable-Broadcast along with trusted hardware can be used to increase the fault tolerance of protocols to minority faults without worsening their communication complexity.

$(\frac{n}{2} + 1)$-Provable-Broadcast. This primitive is a generalization of $(t + 1)$-provable broadcast introduced by Abraham et al. [5]. Informally, in this broadcast, a designated sender sends a message $m = (v, \sigma_{\text{in}})$ consisting of a value v and a proof σ_{in} to all parties. If the message satisfies a certain predicate denoted by the validation function validate(), parties deliver the message. Finally, the sender delivers a proof σ_{out} indicating that $\frac{n}{2} + 1$ parties have delivered the broadcasted message. The primitive provides the following guarantees:
- **Integrity.** An honest party (acting as a participant) delivers at most one message m for a given broadcast instance id.
- **Validity.** If an honest party delivers a message m for instance id, then validate$(id, (m, \sigma_m)) = \text{true}$, where σ_m is the proof of validity for m.

– **Provability.** If a sender can produce two valid proofs σ_{out} and σ'_{out} such that they are valid proofs for the delivery of m and m' respectively in instance id, then $m = m'$, and there exist $n/2 + 1$ parties who cannot deliver a value m' such that $m' \neq m$ in instance id.

– **Termination.** If the sender is honest, no honest party invokes abandon(id), all messages among honest parties arrive, and validate($id, (m, \sigma_{\text{in}})$) = true for all honest parties, then (i) eventually all honest parties deliver m, and (ii) the sender delivers m with a valid proof σ_{out}.

The requirements described above have minor differences from Abraham et al. [5]. In particular, we modify the threshold from $t + 1$ to $\frac{n}{2} + 1$ and require the provability property to have $\frac{n}{2} + 1$ parties to not be able to deliver a different message.

Our goal is to tolerate $(\frac{1}{2} - \epsilon)n$ Byzantine parties with linear communication complexity and ensure the size of σ_{out} to be $O(1)$. The $O(1)$-sized proof allows us to use the primitive in a cascading manner while still maintaining linear communication complexity. To achieve these guarantees, we make use of two components: trusted hardware modules and expander graphs. Each party has access to a trusted hardware module as described in Section 2.2. Parties are connected in a d-regular expander graph $G_{n,\epsilon,(1-\frac{\epsilon}{c})n}$ for a constant d, $0 < \epsilon < \frac{1}{2}$, and $c > 2$ that satisfies the properties in Lemma 1; the expander graph is used to communicate messages with constant communication complexity per party with its neighbors. We denote the neighbors of party p_i in the expander graph by $\rho(i)$.

Intuition. In the presence of a trusted hardware, any party receiving a valid message from the sender can attest to this message using the Append() call to their trusted hardware in a specified log and sequence number. Sending this attestation back to the sender guarantees both provability as well as termination against a corruption threshold of $< 1/2$. For provability, if delivery for every honest party requires attesting at a specific position in a log as a proof, then receiving $\frac{n}{2}+1$ attestations from a set of parties P is a sufficient proof to state that parties in P cannot deliver a different message. For termination, if the sender is honest and eventually the sender's messages arrives at honest parties, then all honest parties will attest to this message in the correct log and sequence number and deliver m; the attestations sent back to the sender will allow it to deliver m with a proof σ_{out} consisting of all the attestations it received. However, the proof σ_{out} is not $O(1)$ words. Observe that the attestations from the small trusted hardware from a linear number of parties provide us with $O(n)$ signatures. Thus, the challenge is to ensure that the proof σ_{out} remains $O(1)$ without relying on the hardware to generate threshold signatures.

Our key idea is to verify the existence of $\frac{n}{2} + 1$ attestations by spreading this work evenly among the parties. We aim for two seemly opposing goals: On the one hand, each party needs to check just a constant number of attestations to be locally satisfied. On the other hand, if a majority of parties say they are locally satisfied then, even if t of them are lying then it is still the case that there were $\frac{n}{2}+1$ attestations. We obtain this through the magic of expander graphs. Every party communicates their attestation with their neighbors in the network. The expansion properties of the graph ensure that receiving correct information from a small fraction of honest parties, specifically ϵn, suffices to learn the state about a majority of the parties in the network. In particular, on receiving some *vote* messages (containing a threshold signature share) from $\frac{n}{2}+1$ parties (see lines 4–7 in Algorithm 2), out of which at least ϵn parties are honest, the sender can learn that a majority of parties (not necessarily honest) have attested to a message m in the log and sequence number corresponding to the instance, and thus cannot deliver a different message with a valid attestation. To ensure that the proof σ_{out} is $O(1)$ words, the sender can simply combine the threshold signature shares sent in the *vote* messages of each of the $\frac{n}{2} + 1$ parties that vote.

Algorithm 2 $(\frac{n}{2}+1)$-PB-Initiate instance id (sender s)

```
1: procedure $(\frac{n}{2}+1)$-PB-Initiate $(id, (v, \sigma_{\text{in}}))$
2:     S := {}
3:     send "$id, send, (v, \sigma_{\text{in}})$" to all parties
4:     while $|S| \leq \frac{n}{2}$
5:         upon receiving "$id, vote, \xi_j$" from $p_j$ for the first time do
6:             if share-validate$(v, \xi_j, pk_j)$ = true
7:                 $S := S \cup \{\xi_j\}$
8:     $qc :=$ threshold-combine$(S)$
9:     $\sigma_{\text{out}}.id := id,\ \sigma_{\text{out}}.val := v,\ \sigma_{\text{out}}.qc_{\sigma_{\text{in}}} := \sigma_{\text{in}}.qc,\ \sigma_{\text{out}}.qc := qc$
10:    deliver $\sigma_{\text{out}}$
```

Algorithms 2 and 3 present the pseudocode for $(\frac{n}{2}+1)$-Provable-Broadcast. We assume a setup phase during which each party creates the necessary logs for the protocol using the CreateLog interface. Further, we assume, for an instance of $(\frac{n}{2}+1)$-Provable-Broadcast, that every party appends to, and expects attestations from, the log in the trusted hardware module of each node with the same $logId$ and in the same position, $seqNo$, within the log.

Protocol. Each instance of this protocol is identified by an id and a designated sender s. The sender receives two inputs (v, σ_{in}); v is the value to be sent and σ_{in} is a proof to be validated by other parties using the validate() function. The sender sends the message "$id, send, (v, \sigma_{\text{in}})$" to all parties (Algorithm 2 line 3).

Algorithm 3 $(\frac{n}{2}+1)$-PB-Respond instance id (party p_i)

```
1: procedure $(\frac{n}{2}+1)$-PB-Respond ($id$, validate, validateNeighbor)
2:     $stop$ := false
3:     upon receiving "$id, send, (v, \sigma_{\text{in}})$" from $s$ do
4:         $(\sigma_i, \text{valid}) :=$ validate$(id, (v, \sigma_{\text{in}}))$
5:         if valid:
6:             $(\text{att}_{logId}, \text{att}_{head}) :=$ createAttestations$(id, (v, \sigma_{\text{in}}))$
7:             send "$id, send, ((\text{att}_{logId}, \text{att}_{head}), \sigma_i)$" to all parties in $\rho(i)$
8:             wait for $id, send, ((\text{att}_{logId,j}, \text{att}_{head,j}), \sigma_j)$
               from $(\frac{1}{2} + \frac{\epsilon}{2})d$ parties $p_j$ in $\rho(i)$ s.t.
               validateNeighbor$(id, p_j, (v, \sigma_{in}), (\text{att}_{logId,j}, \text{att}_{head,j}), \sigma_j)$ = true
9:             $\xi_i :=$ threshold-sign$_i(id, (v, \sigma_{\text{in}}.qc))$
10:            send "$id,\ vote,\ \xi_i$" to $s$
11:            $stop$ := true
12:
13:    upon abandon($id$) do
14:        $stop$ := true
15:
16: procedure createAttestations$(id, (v, \sigma_{\text{in}}))$
17:    $logId := \log(id),\ seqNo := \text{seq}(id)$                    ▷ parse $id$
18:    $\text{att}_{logId} :=$ Append$(logId, seqNo, (v, \sigma_{\text{in}}))$
19:    deliver $((v, \sigma_{\text{in}}), (\text{att}_{logId}))$
20:    return $(\text{att}_{logId}$, Counters())
```

On receiving the message from the sender, p_i invokes validate$(id, (v, \sigma_{\text{in}}))$ (Algorithm 3 line 4). The validate() function is used to check that the sender's proposal is valid and that it satisfies any predicates on p_i's state as necessary for the higher level protocol. Thus, the interface allows each party to optionally provide some additional data/state that can be used for validation. If validate() is successful (valid = true), it returns a proof, σ_i, as proof that p_i can provide to other parties to prove that its call to validate($id, (v, \sigma_{\text{in}})$) returned true. Upon successful validation, p_i then delivers the sender's proposal by appending it to the the log in its trusted hardware component using the createAttestations() method (Algorithm 3 lines 6, 16-20). In this method, p_i determines the log $logId$ and the sequence $seqNo$ in the log to be used using the log(id) and seq(id) functions. p_i then appends (v, σ_{in}) to log $logId$ at sequence number $seqNo$ in its trusted hardware component. It sends the attestation (along with a COUNTERS attestation, for reasons explained in Section 4) to all its neighbors $\rho(i)$ in the expander graph, as proof that it has delivered (v, σ_{in}). On receiving messages from a majority of its neighbors (specifically $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors) that satisfy validateNeighbor() (Algorithm 3 line 8), a party sends a *vote* message with threshold-sign$_i((v, \sigma_{in}))$ to the sender (line 10). The validateNeighbor() function allows an invoking party to perform validation on the messages sent by their neighbors as proof of delivery; in the above instance, we can assume that it only validates that the attestation $\text{att}_{logId,j}$ is correct, i.e. that it is from the log $logId$, signed by the sender's trusted hardware component, and that the value was appended in the correct sequence number. In Section 4, we show how the proof output from validate() as well as the att_{head} attestation are used as well.

On collecting threshold-sign$_i((v, \sigma_{in}))$ from a majority of replicas, the sender combines the signature shares to generate σ_{out} and delivers σ_{out} (Algorithm 2 lines 8-10).

Here are the key ideas that ensure provability and termination.

1. **Any set of ϵn nodes, each of which receives an attestation from at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ of its neighbors, collectively receives attestations from at least $\frac{n}{2} + 1$ unique parties.** This property has been shown in Lemma 1 and it guarantees provability since if a sender receives *vote* messages from a majority of parties, at least ϵn of them are honest and they will ensure that at least $\frac{n}{2} + 1$ parties have attested to (v, σ_{in}) in the correct log and sequence number. Thus, they cannot deliver a message other than v with a valid proof of attestation. Also, by the same argument, another value $v' \neq v$ cannot receive a sufficient number of attestations, causing another set of ϵn honest parties to send a *vote* message for v'; this is because at least $\frac{n}{2} + 1$ parties need to attest to $(v', *)$, and two majority sets will intersect in at least one node.
2. **For any set of $(1/2 + \epsilon)n$ nodes S, at least $(\frac{n}{2} + 1)$ nodes in S each have at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S.** This property has been shown in Lemma 1 and it guarantees termination since if an honest sender sends a valid (v, σ_{in}) to all $(1/2 + \epsilon)n$ honest parties, then at least $\frac{n}{2} + 1$ honest parties will send *vote* messages, sufficient to generate the proof σ_{out}.

3.1 Security Proofs

Lemma 2 (Provability). *In $(\frac{n}{2} + 1)$-PB-Initiate, if the sender delivers two valid proofs σ_{out} and σ'_{out} corresponding to values (v, σ_{in}) and (v', σ'_{in}) respectively, then (i) $v = v'$, and (ii) at least $\frac{n}{2} + 1$ parties satisfy the criteria in the validate() function, and the parties have created attestations in createAttestations() such that they satisfy validateNeighbor().*

Proof. Since σ_{out} contains a threshold signature for (v, σ_{in}) signed by at least $\frac{n}{2} + 1$ parties, at least $\frac{n}{2} + 1 - t > \epsilon n$ honest parties p_i must have sent messages "*id*, *vote*, ξ_i" to the sender. Thus, each such p_i must have received at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ messages "*id*, *send*, $((\text{att}_{logId}, \text{att}_{head,j}), \sigma_j)$" from parties p_j such that the attestations $(\text{att}_{logId}, \text{att}_{head,j})$ along with σ_j satisfy the criteria in the validateNeighbor() function. Thus, $(\text{att}_{logId}, \text{att}_{head,j})$ were created by running createAttestations(id, (v, σ_{in})) and σ_j proves that (v, σ_{in}) is valid for p_j's state based on the conditions in validate(id, (v, σ_{in})). By Lemma 1, any ϵn set of parties each receiving attestations from $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors, should collectively receive attestations from at least $\frac{n}{2} + 1$ parties such that they satisfy validateNeighbor(). This completes part (ii) of the proof. For part (i), observe that any two quorums of size $\frac{n}{2} + 1$ will always intersect at one party, and due to the use of trusted hardware, this party cannot attest to two different values v and v' such that $v \neq v'$ for the same log and sequence number. This completes part (i) of the proof.

Lemma 3 (Termination). *If the sender is honest, no honest party invokes abandon(id), all messages among honest parties arrive, and validate(id, (v, σ_{in})) = true for all honest parties, then (i) eventually all honest parties deliver v, and (ii) the sender delivers v with a valid proof σ_{out}.*

Proof. Observe that the sender's message will eventually arrive at all $(1/2+\epsilon)n$ honest parties if no party invokes abandon(). Since the message sent by the sender is valid for all honest parties, all honest parties will invoke createAttestations() and deliver the sender's message along with a valid attestation as the proof. They then send their attestations to all their neighbors. By Lemma 20, at least $\frac{n}{2}+1$ of the honest parties H will each receive attestations from at least $(\frac{1}{2}+\frac{\epsilon}{2})d$ of their neighbors that satisfy validateNeighbor() without any participation from any Byzantine parties. Each party in H will send a *vote* message with a threshold signature share to the sender, who can combine them into (v, σ_{out}).

Theorem 3. *The* $(\frac{n}{2}+1)$*-Provable Broadcast algorithm in Algorithms 2 and 3 satisfies Integrity, Validity, Provability, and Termination. Moreover, the protocol has linear communication complexity with an* $O(1)$*-sized proof.*

Proof. Integrity is satisfied deterministically by the algorithm. All the messages from the sender to the parties and vice-versa involve messages with $O(1)$ words. All the communication between parties through the expander graph consists of $O(1)$ sized messages to a constant d number of neighbors. Thus the communication complexity is linear. Also, the proof delivered by the sender is a threshold signature of $O(1)$ size.

An honest party only sends a signature share to s for a value v if v is externally valid as per the validate() function. Therefore, as long as the threshold for the threshold signature is greater than the number of Byzantine parties in the network, only an externally valid message can obtain a valid threshold signature, satisfying validity.

Provability and Termination property have been shown in Lemmas 2 and 3.

4 HotStuff-M: HotStuff with Minority Corruption

In this section, we present HotStuff-M, a version of the HotStuff [54] protocol that tolerates minority Byzantine corruption under partial synchrony assuming a minimal trusted hardware at each party. Similar to HotStuff, the protocol has a linear communication complexity per view. For simplicity, we show the construction of a single-shot version of HotStuff, though the ideas directly extend to the state machine replication setting.

4.1 Overview of Basic HotStuff

We start with an overview of the Basic HotStuff protocol [54] tolerating $n=3t+1$. The protocol proceeds in a sequence of consecutive views where each view has a unique leader. Each view of HotStuff progresses as follows:

- **Promote.** The leader proposes a PROMOTE message containing a proposal v along with the $\sigma_{highKey}$ from the highest view known to it and sends it to all parties. On receiving a PROMOTE message containing a value v in a view e and a $\sigma_{highKey}$, a party sends a vote for v if it is *safe* to vote based on a locking mechanism (explained later). It sends this vote to the leader.
- **Key.** The leader collects $2t+1$ votes to form a threshold signature σ_{key} in view e. The leader sends the σ_{key} for view e to all parties. On receiving a σ_{key} in view e containing message v, a party updates its highest σ_{key} to (v,e) and sends LOCK to the leader.
- **Lock.** The leader collects $2t+1$ such votes to form a threshold signature σ_{lock}, and sends it to all parties. On receiving σ_{lock} in view e containing message v from the leader, a party locks on (v,e) and sends COMMIT message to the leader.
- **Commit.** The leader collects $2t+1$ such votes to form a threshold signature σ_{commit} and sends it to all parties. On receiving σ_{commit} from the leader, parties output the value v.

Once a party locks on a given value v, it only votes for the value v in subsequent views. The only scenario in which it votes for a value $v' \neq v$ is when it observes a $\sigma_{highKey}$ from a higher view in a PROMOTE message. At the end of a view, every party sends its highest σ_{key} to the leader of the next view. The next view leader collects $2t+1$ such values and picks the highest σ_{key} as $\sigma_{highKey}$. The safety and liveness of HotStuff follows from the following:

Uniqueness within a view. Since parties only vote once in each phase, a σ_{commit} can be formed for only one value.

Safety and liveness across views. Safety across views is ensured using locks and the voting rule for a PROMOTE message. Whenever a party outputs a value, at least $2t+1$ other replicas are locked on the value in the view. A party only votes for the value it is locked on. The only scenario in which it votes for a conflicting value v' is if the leader includes a σ_{key} for v' from a higher view in a PROMOTE message. This indicates that at least $2t+1$ replicas are not locked on v in a higher view, and hence it should be safe to vote for it. The latter constraint of voting for v' is not necessary for safety, but only for liveness of the protocol.

Phase	validate$(id, (v, \sigma_{\text{in}}))$	validateNeighbor$(id, (v, \sigma_{\text{in}}), (\text{att}_{logId,j}, \text{att}_{heads,j}), \sigma_j)$
PROMOTE	**cond:** ext-valid(v) = true, $\sigma_{\text{in}}.val = \sigma_{lock}.val$ or view(σ_{in}) > view(σ_{lock}) **proof:** $\sigma_i := \text{att}_{lock}$	($\sigma_{\text{in}}.val = \sigma_{lock}.val$ or view(σ_{in}) > view(σ_{lock})), and LOCK log in $\text{att}_{heads,j}$ is at view(σ_{lock}), and PROGRESS log in $\text{att}_{heads,j}$ is at $e-1$, and $\text{att}_{logId,j}$ is a valid attestation from HW_j for value v in the PROMOTE log and sequence number e
KEY	**cond:** view$(\sigma_{\text{in}}) = e$ and phase(σ_{in}) = PROMOTE **proof:** $\sigma_i := \perp$	PROGRESS log in $\text{att}_{heads,j}$ is at $e-1$, and KEY log in $\text{att}_{heads,j}$ is at e, and $\text{att}_{logId,j}$ is a valid attestation from HW_j for value v in the KEY log and sequence number e
LOCK	**cond:** view$(\sigma_{\text{in}}) = e$ and phase(σ_{in}) = KEY **proof:** $\sigma_i := \perp$	PROGRESS log in $\text{att}_{heads,j}$ is at $e-1$, and LOCK log in $\text{att}_{heads,j}$ is at e, and $\text{att}_{logId,j}$ is a valid attestation from HW_j for value v in the LOCK log and sequence number e

Table 1. Validation functions passed to provable broadcast in different phases of view e. We assume end attestations att_{lock} (containing σ_{lock}) is invoked during validate() call as needed. Also note that $(\text{att}_{logId,i}, \text{att}_{heads,i})$ are sent by every party to their neighbor as a part of provable broadcast and generated in the invocation of createAttestations$(id, (v, \sigma_{\text{in}}))$.

4.2 HotStuff-M: Towards Minority Corruption

The arguments for safety and liveness of HotStuff crucially rely on having fewer than one-third Byzantine faults. Otherwise, Byzantine parties could create multiple $\sigma_{key}, \sigma_{lock}, \sigma_{commit}$ by partitioning the honest parties. Similarly, across views, Byzantine parties could send an incorrect (stale) σ_{key} to the leader, as well as vote for a message in the PROMOTE phase without respecting the locking condition, leading to both safety and liveness concerns.

Our goal is to increase the corruption threshold from one-third to a minority while still retaining the linear communication complexity. The trusted hardware provides a non-equivocation guarantee, i.e., it ensures that once a value v has been appended to a position in a specified log, no other value can be appended at that position in that log. Moreover, the hardware provides an attestation, i.e., verifiable proof of the existence of value v at that position of the specified log. However, a party can still send a stale attestation to another party. For instance, during a view-change, a party can send an attestation to a key from an old view, possibly leading to a liveness violation. It can also potentially participate in a previous view even after quitting the current view. Similarly, a party can potentially append conflicting information at two different positions of the log and provide attestations to these different positions to different parties.

A potential way to fix the above concerns is to always send an attestation of all positions in the log whenever sending a message. The receiving party can validate that the log has been correctly constructed, e.g., non-existence of conflicting information on the log, and absence of a quit-view message on the log. However, this solution makes the communication complexity proportional to the number of views for each message.

Our approach and protocol. Our approach uses multiple logs in the trusted hardware, one for each phase that consists of an instance of $(\frac{n}{2}+1)$-Provable-Broadcast, as well as a log to keep track of the

Algorithm 4 HotStuff-M: HotStuff with Minority Corruption (for party p_i).

```
1: for $e := 1, 2, 3, \ldots$ do
2:     as a leader                                                    $\triangleright$ NEW-VIEW phase
3:         wait for a set $M$ of $\geq \frac{n}{2} + 1$ NEW-VIEW messages s.t. the attestations on PROGRESS and KEY
4:         logs are valid, sequence numbers in $\text{att}_{progress}$ and $\text{att}_{highQC}$ match those in the COUNTERS attestation
   for the
5:         respective logs, and counter value of PROGRESS log in the COUNTERS attestation is $e - 1$
6:         For each $m \in M$, let $\sigma^m_{highKey}$ denote the highest key QC from party $p_m$
7:         $\sigma_{highKey} := (\arg\max_{m \in M}\{\text{view}(\sigma^m_{highKey})\})$    $\triangleright$ $\text{view}(\sigma^m_{highKey})$ is the view in which $\sigma^m_{highKey}$ was formed
8:         if $\sigma_{highKey} = \bot$ then proposal := client's command else proposal := $\sigma_{highKey}.val$
9:     as a party                                                     $\triangleright$ NEW-VIEW phase
10:        go to this line if no progress happens during the "wait" step in any phase
11:        $\text{att}_{progress} := \langle \text{LOOKUP}, \text{PROGRESS}, e, e \rangle := \text{APPEND}(\text{PROGRESS}, e, e)$
12:        $\text{att}_{highQC} := \langle \text{END}, \text{KEY}, seqNoHighQC, highKeyQC \rangle := \text{END}(\text{KEY})$
13:        send "$((e, \text{NEW-VIEW}), send, (\text{att}_{progress}, \text{att}_{highQC}, \text{COUNTERS}()))$" to leader of view $e + 1$
14:    as a leader
15:        $\sigma_{key} := (\frac{n}{2} + 1)$-PB-Initiate($(e, \text{PROMOTE}), (\text{proposal}, \sigma_{highKey})$)    $\triangleright$ PROMOTE phase
16:        $\sigma_{lock} := (\frac{n}{2} + 1)$-PB-Initiate($(e, \text{KEY}), (\text{proposal}, \sigma_{key})$)    $\triangleright$ KEY phase
17:        $\sigma_{commit} := (\frac{n}{2} + 1)$-PB-Initiate($(e, \text{LOCK}), (\text{proposal}, \sigma_{lock})$)    $\triangleright$ LOCK phase
18:        send "$((e, \text{COMMIT}), send, \sigma_{commit})$"    $\triangleright$ COMMIT phase
19:    as a party, invoke the following in parallel
20:        $(\frac{n}{2} + 1)$-PB-Respond($(e, \text{PROMOTE})$, validate(), validateNeighbor())    $\triangleright$ PROMOTE phase
21:        $(\frac{n}{2} + 1)$-PB-Respond($(e, \text{KEY})$, validate(), validateNeighbor())    $\triangleright$ KEY phase
22:        $(\frac{n}{2} + 1)$-PB-Respond($(e, \text{LOCK})$, validate(), validateNeighbor())    $\triangleright$ LOCK phase
23:        wait for "$((e, \text{COMMIT}), send, \sigma_{commit})$" from the leader of view $e$    $\triangleright$ COMMIT phase
24:        if $\sigma_{commit}$ is a valid signature from view $e$ and from phase COMMIT then commit $\sigma_{commit}.val$
```

view a party is in. For each log, the data appended to position j corresponds to the message sent by the party in view j. Thus, if a party votes for a value v in view e in the KEY phase of the protocol, it calls APPEND(KEY, e, $(v, *)$) to the KEY log at position e ($*$ denotes some additional information). However, a disadvantage of using multiple logs is the absence of relative ordering between them. This allows a Byzantine adversary to participate in a previous view by showing a stale state of a log or send a stale σ_{lock} while voting in the PROMOTE phase. We leverage the COUNTERS() call on the hardware to address this concern; it provides the end state of all of the logs at once, thus allowing the receiving party to validate the freshness of the state. Although the functionality provided by the COUNTERS attestation can be achieved using a nonce with the same communication complexity, we use COUNTERS for the simplicity of the description.

Our protocol is presented in Algorithm 4 and Table 1. The parties proceed in a sequence of views. We assume that the parties know the leader in a given view. Let e denote the current view of the protocol. At the end of the previous view, each party invokes an APPEND(PROGRESS, $e-1$, $e-1$) (line 11) to obtain attestation $\text{att}_{progress}$. In addition, it obtains an end attestation att_{highQC} for its *keyQC* (line 12). The party sends this information together with the COUNTERS() attestation to the leader in a NEW-VIEW message (line 13).

The leader of view e waits for a valid NEW-VIEW message from a majority of parties. Here, the message is considered valid if (i) the attestations are valid (i.e. signed by the trusted hardware component of the sending party), (ii) the sending party has quit view $e-1$, i.e., the counter value of the PROGRESS log is equal to $e-1$, and (iii) the sequence number on att_{highQC} and $\text{att}_{progress}$ matches the ones in the COUNTERS attestation (lines 3-5). This ensures that even if the sending party is Byzantine, the att_{highQC} is fresh and the party can no longer act in the previous view (due to the current counter value of its PROGRESS log and the conditions in validateNeighbor()). The leader picks the *keyQC* from the highest view as $\sigma_{highKey}$ and proposes the value in the certificate. Otherwise, it proposes any client command.

Our modular construction allows us to present the next three phases PROMOTE, KEY, and LOCK as invocations of $(\frac{n}{2} + 1)$-Provable-Broadcast (lines 15-17 and 20-22). As described in the previous section, if the leader successfully receives a σ_{key} (respectively σ_{lock} and σ_{commit}), it guarantees that $\geq \frac{n}{2}+1$ parties have attested to the proposed value in their PROMOTE log (respectively KEY log and LOCK log) in the position corresponding to view e. However, in each provable broadcast phase, a party should vote for the leader's proposal only if it is safe to do so depending on the party's state. We use the validate() and validateNeighbor() interface to specify these constraints (described in Table 1). Recall that the former is used to validate the leader's proposal and provide a proof that the leader's proposal satisfies validate() for this party, while the latter is used by a neighbor in the expander graph to verify correct behavior.

In the PROMOTE phase, a party votes for a leader's message only if it is locked on the same value as the proposal or if $\sigma_{highKey}$ in the leader's proposal is from a higher view than the party's lock, σ_{lock}. The party sends an attestation to its lock as proof for the neighbor to verify. The expander graph neighbor verifies the correctness of the computation in addition to ensuring that the attestations received are valid and fresh (using the counter values in $\text{att}_{heads,j}$ and comparing them to the sequence numbers in the other attestations). In the KEY, LOCK, and COMMIT phases, the parties check if σ_{key}, σ_{lock}, and σ_{commit}, respectively, are from the same view and the proofs were formed in the correct phases. The expander graph neighbors verify validity of attestations and freshness (to ensure they have not quit the view). Finally, the leader sends a COMMIT message along with σ_{commit} as proof of commit. Each of the parties can then commit $\sigma_{commit}.val$.

We present a view synchronization protocol in Appendix B; the protocol is a generalization of the expected linear communication complexity protocol of [38] to withstand minority corruption.

Communication complexity. From Theorem 3, an instance of provable broadcast in each of the three phases (PROMOTE, KEY, and LOCK) incurs linear communication complexity. To change views, each party sends a constant number of attestations to the leader in a single message. In both the NEW-VIEW phase and the COMMIT phase, the leader sends a single, constant-sized message to all the parties. Therefore, the HotStuff with Minority Corruption protocol incurs $O(n)$ communication complexity per view.

4.3 Security Proofs

Lemma 4. *At the end of a view e, (i) if a party receives a σ_{commit} on value v, then $\geq \frac{n}{2}+1$ parties appended value v at position e in their LOCK logs prior to appending a value at position e in their PROGRESS logs, and (ii) if a party receives a σ_{lock} on value v, then $\geq \frac{n}{2}+1$ parties appended value v at position e in their KEY logs prior to appending a value at position e in their PROGRESS logs.*

Proof. This lemma follows from Lemma 2 and the criteria for validateNeighbor() in Table 1.

Lemma 5. *Suppose the earliest view in which a value v is committed by an honest party is e. For all views $> e$, a valid σ_{key} for a value $v' \neq v$ does not exist.*

Proof. Suppose for contradiction that v has been committed by an honest party in view e and a σ_{key} for $v' \neq v$ exists in view $e' > e$. Let e^* be the earliest view in which a σ_{key} for a value v^* is formed such that $v^* \neq v$ and $e^* > e$. It follows that $e^* \leq e'$. Since there is a σ_{key} for v^* in e^*, by Lemma 2, a set Q of at least $\frac{n}{2}+1$ parties have sent messages with attestations $(\text{att}_j, \text{att}_{heads,j}), \sigma_j$ that satisfy validateNeighbor(). Since v was committed in view e, there exists a set P of at least $\frac{n}{2}+1$ parties who have inserted v into their LOCK log. The two sets P and Q should intersect at least one party p.

We now show a contradiction w.r.t. party p's log and its attestation satisfying validateNeighbor(). Since view e^* is the first view where a higher σ_{key} was formed for a different value, the end state of LOCK in view e^* must be for value v. Thus, the predicate $\text{view}(\sigma_{\text{in}}) > \text{view}(\sigma_{lock})$ in the PROMOTE phase is not satisfied. Moreover, in view e^*, the proposed value $v^* \neq v$ for our setup. Thus, the condition $\sigma_{\text{in}}.val = \sigma_{lock}.val$ in PROMOTE phase is not satisfied either. Additionally, since a party presents the state of their PROGRESS log through the $\text{att}_{heads,j}$ attestation, they always present the state of their logs after the end of view $e^* - 1 \geq e$. Thus, for party p, validateNeighbor() cannot be satisfied. Consequently, σ_{key} in view e^* cannot be formed for value v', a contradiction.

Theorem 4 (Safety). *Two honest parties p_i and p_j cannot commit to values v and v' such that $v' \neq v$.*

Proof. Let e be the view in which v is committed and e' be the view in which v' is committed s.t. $v \neq v'$. By Lemma 2, $e \neq e'$. Suppose, without loss of generality, $e > e'$. By Lemma 5, no σ_{key} can be formed in view $> e$ for value $\neq v'$. Also, honest parties only vote for a σ_{lock} in the LOCK phase in view e' if it was generated in view e'. Thus, a σ_{commit} cannot be formed in view e'.

Theorem 5 (Liveness). *After GST, there exists a bounded time such that when an honest leader is elected in view e and all honest parties remain in view e for that time, then a value will be committed by all honest parties.*

Proof. View e is a view after GST where the leader is honest. Suppose σ^*_{lock} is the σ_{lock} stored in a party's LOCK log from the highest view e^* for a value v. By Lemma 4, at least $\frac{n}{2}+1$ parties must have the σ^*_{key} for value v stored in their LOCK logs at position e^* prior to creating the COUNTERS() attestation to report their highest σ_{lock} in e. Since the leader waits for $\frac{n}{2}+1$ valid σ_{key} messages during a view change, it will obtain σ_{key} from a view $\geq e^*$. Let σ'_{key} be the highest valid σ_{key} received by the leader. The leader will propose $(\sigma'_{key}.val, \sigma'_{key})$ in the PROMOTE phase. Since e^* is the highest view for which a party has a lock, it must be the case that for each honest party, either view$(\sigma'_{key}) >$ than the locked view of the party or that v is the value that the party is locked on (Lemma 2). Since e is after GST, all messages will arrive within the bounded delay Δ, and thus for each of the three phases, the termination property for provable broadcast from Lemma 3 should be satisfied. Thus, all honest parties will commit.

5 VABA-M: Validated Asynchronous Byzantine Agreement with Minority Corruption

In this section, we describe VABA-M, an *asynchronous* protocol that tolerates minority Byzantine corruption assuming a small trusted hardware at each party. Our goal is to achieve the optimal quadratic communication complexity for Byzantine agreement. We improve the resilience of the VABA protocol from PODC'19 [5], which originally tolerated $< n/3$ Byzantine faults using $O(n^2)$ communication to tolerate $\leq (1/2 - 2\epsilon)n$ corruption while retaining the $O(n^2)$ communication complexity.

5.1 Overview of the VABA Protocol

At a high level, the VABA protocol [5] consists of three stages:
- **Leader nomination.** In this stage, each of the n parties run n parallel HotStuff-like instances, called proposal promotions, where party i acts as the leader within instance i.
- **Leader election.** After the completion of the leader nomination phase as a result of the completion of $n-t$ parties' proposal promotions, parties run a leader election protocol using a threshold coin primitive [9] to elect the leader of this view uniformly at random. At the end of the view, parties adopt the "progress" from the leader's proposal promotion instance, and discard information from other instances.
- **View-change.** Parties broadcast their updated state from the leader's proposal promotion instance and/or output values and update their cross-view variables as appropriate.

Since proposal promotion is similar to a HotStuff view, the guarantees provided by the leader nomination stage are the same as that of a HotStuff view. The leader election phase elects a unique leader at random – this stage guarantees (i) with $\geq 2/3$ probability, a leader whose proposal promotion completed is elected, and (ii) an adaptive adversary cannot stall progress (since a leader is elected in hindsight). Finally, in the view-change phase, every party broadcasts the "quorum certificates" from the elected leader's proposal promotion instance to all other parties. Since the protocol uses $> 2n/3$-sized quorums, if a party is locked (resp. committed) on a given value in a view, there are $> 2n/3$ parties who hold a key (resp. lock), out of which $> n/3$ are guaranteed to be honest parties. If a party waits for $> 2n/3$ view-change messages, due to a simple quorum intersection argument, it will receive a key (resp. lock) from at least one of the parties. This ensures no conflicting value can be proposed in subsequent views, thus ensuring safety.

On the other hand, liveness is guaranteed when a party who has completed its proposal promotion instance is chosen as the leader. Since a leader is chosen after the leader nomination stage, a party who has completed proposal promotion is elected with probability $> 2/3$.

5.2 VABA-M: Asynchronous Byzantine Agreement with Minority Corruption: Intuition

At first sight, converting the VABA protocol to tolerate minority corruption using trusted hardware seems straightforward. Since proposal promotion instances are similar to the computations within a HotStuff view, the techniques we used in the previous section should carry over to the leader nomination phase in a straightforward manner. The threshold coin primitive in the VABA protocol can still be used to elect a leader under a minority corruption. Finally, the view-change stage is similar to that of HotStuff; the only difference is that parties are now sending their updated state (keys, locks, and commits) to all other parties instead of just the leader. However, this can still be performed with quadratic communication complexity. Thus, techniques similar to that in a HotStuff-M view-change can directly be used here to improve the fault tolerance.

On a closer look, the above arguments are flawed. The key concern is related to the amount of storage required in the hardware unit. Running n parallel proposal promotion instances, as is, requires the hardware to store a factor n more information. For a small trusted hardware, this is unreasonable. This limitation affects both the proposal promotion stage as well as the view-change stage. Thus, we resort to different techniques to address these concerns.

Using $O(1)$ trusted storage during proposal promotion. HotStuff-M used multiple logs, one for each phase of $(\frac{n}{2}+1)$-Provable-Broadcast, as well as one to maintain the progress of the party across views. In each log, position j corresponded to information in view j. In contrast, since our proposed asynchronous protocol has n parallel proposal promotion instances, if we still use only one log for each phase, we need n different positions to store information about a single view. Moreover, we cannot use fixed log positions for each instance within a view, e.g., storing information of instance k for view e at position $ne + k$. This is because the logs can only be appended to, and thus, before inserting a value for instance k, we need to append values for $k' < k$. Due to asynchrony and since some parties may be faulty, a party may never be able to distinguish between a delayed message for instance k' and a faulty leader of instance k' not sending a message. In summary, we cannot rely on such a mapping to store information about view j for some proposal instance. In fact, due to the asynchrony in the network, no such deterministic mapping may be possible.

We solve this concern by allowing every party to store information about a view e in n consecutive positions, say ne to $(n+1)e-1$. However, among these positions, the information can be stored in an arbitrary order. We crucially rely on the observation that all parties communicate using the same expander graph for all proposal promotion instances. Thus, independent of the order in which a party p_i appends to its log, it always sends all of this $O(n)$-sized information to the same set of neighbors $\rho(i)$ across all instances. Thus, if a party uses the hardware correctly and uses the positions consecutively, its neighbor will eventually receive attestations corresponding to all previous positions. Thus, a neighbor can always validate that no two log positions are used for the same proposal promotion instance. Further, once a party appends a value to a trusted hardware log, the interface immediately returns an attestation that the party can store locally. Therefore, the hardware does not need to maintain more than a small, constant amount of storage.

Using $O(1)$ trusted storage during view-change. The parties engage in a leader election stage after $n-t$ proposal promotion instances are completed in the leader nomination stage. If a leader p_k is elected, then subsequently, in a view-change phase parties only need to learn information about proposal promotion instance k. In particular, parties need to learn the status of the highest keys, locks, and commits from this view. For simplicity of description, we will assume that parties only share their locks, but the same arguments work for keys and commits too. In HotStuff-M, to learn the status of the highest lock that they hold, parties share the *end* status of their LOCK logs with all other parties. HotStuff-M used the COUNTERS() attestation in conjunction with a PROGRESS log to guarantee a fresh state of attestations after a party has quit the view.

Recall that during the proposal promotion stage we use the logs in an arbitrary order among different instances. Since not all parties have access to every parties' state of logs, the solution used in HotStuff-M does not work unless each party shares the last n attestations for each of the logs (thus worsening communication complexity). If all n attestations are not shared, a Byzantine party can use logs incorrectly and share conflicting information with other parties, e.g., stating that it does not hold a lock from this view while it did.

We observe that for every party p_i, their neighbors already do have information about exactly how the logs were used by p_i. If the neighbors could confirm the correctness of information sent by a party, perhaps we can detect incorrect information shared by Byzantine parties. However, again, one or more of its neighbors could be Byzantine. Fortunately, since parties are connected through an expander graph, we can use the expander properties to bound the amount of incorrect information shared during the view change stage.

In particular, from Lemma 1, and assuming we tolerate $t \leq (1/2 - 2\epsilon)n$ we can show the following properties hold in the graph.

1. The number of honest parties in the network with fewer than $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors is less than ϵn.
2. The number of Byzantine parties in the network with fewer than $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors is less than ϵn.
3. The number of honest parties in the network with at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors is greater than $\frac{n}{2} + \epsilon n$.

Thus, if we consider a valid view-change message as consisting of an att_{lock} (containing σ_{lock}) and signatures from $\geq (\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors attesting to the correctness of the attestation, then at most ϵn Byzantine parties can send stale information (they cannot send arbitrarily incorrect information since the attestation contains σ_{lock}) that is considered as valid. There can also be at most ϵn honest parties' messages that can be blocked or incorrectly flagged as incorrect by their majority Byzantine neighbors. However, every party will either not receive this message or can detect that the messages are invalid. Moreover, we know that if some party holds a σ_{commit} of a view, based on the arguments we made in Section 3, at least $\frac{n}{2} + 1$ parties must have created attestations in their LOCK logs. Thus, there can be at most $\frac{n}{2} - 1$ parties who do not hold a lock from this view.

The above discussion suggests that there are at most $\frac{n}{2} + \epsilon n - 1$ valid messages (honest parties who did not lock and Byzantine parties with majority Byzantine neighbors) from parties that does not consist of the highest lock in this view. Thus, if we wait for $> \frac{n}{2} + \epsilon n - 1$ valid messages, every honest party should receive the highest lock. Since there are $\geq \frac{n}{2} + 2\epsilon n$ honest parties and fewer than ϵn have $< (\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors, every honest party should receive sufficiently many valid messages required to complete the view-change process while still receiving the highest lock.

Phase	validate$(((j, e), phase), (v, \sigma_{\text{in}}))$	validateNeighbor$(id, (v, \sigma_{\text{in}}), (\text{att}_j, \perp), \perp)$
PROMOTE	**cond:** ext-valid(v) = true, threshold-verify$((((L(view(\sigma_{\text{in}})), \text{view}(\sigma_{\text{in}})), \text{PROMOTE}), (v, \sigma_{\text{in}}.qc_{\sigma_{\text{in}}})), \sigma_{\text{in}}.qc)$ = true, and view(σ_{in}) $\geq LOCK$	att_j is a valid attestation from HW_j for value v in the PROMOTE log and in a position between en and $(e(n+1) - 1)$, and all attestations for positions corresponding to view e in p_j's PROMOTE log up to the position of att_j have been seen and are not for the same proposal promotion instance
KEY, LOCK, COMMIT	**cond:** threshold-verify$((((j, view), phase'), (v, \sigma_{\text{in}}.qc_{\sigma_{\text{in}}})), \sigma_{\text{in}}.qc)$ = true, where $phase'$ is the phase before $phase$	att_j is a valid attestation from HW_j for value v in the log corresponding to this phase and in a position between en and $(e(n+1) - 1)$, and all attestations for positions corresponding to view e in p_j's log for this phase up to the position of att_j have been seen and are not for the same proposal promotion instance

Table 2. Validation functions passed to provable broadcast in different phases of view e in VABA-M.

5.3 Protocol Description

The pseudocode for the top-level protocol has been described in Algorithm 5 and Algorithms 6-7 are used as sub-protocols. Algorithm 8 overrides the createAttestations method presented in Algorithm 3. The protocol progresses in a sequence of views. Algorithm 5 describes the three stages of the protocol within a view: leader

Algorithm 5 VABA-M: VABA with Minority Corruption (for party p_i).

```
1: $LOCK := 0$, $KEY := (v, \sigma_{\text{in}}) := (v_i, \bot)$, $L := []$, $D_{key} := D_{commit} := D_{lock} := []$, $vcCount := 0, doneCount := 0$
2: for view $e := 1, 2, 3, \ldots$ do
3:     for $k = 1, \ldots, n$ do                                        ▷ Leader nomination phase
4:         Proposal-Promotion($(k, e)$, validate(), validateNeighbor())
5:     $\sigma_{\text{out}} :=$ Proposal-Promotion($(i, e), KEY$) as $s$         ▷ Start a proposal promotion instance as $s$
6:     wait for Proposal-Promotion($(i, e), KEY$) instance as $s$ to return or $skip = \text{true}$
7:     if $skip = \text{false}$ then
8:         send "$e$, finished-proposal-promotion, $(v, \sigma_{\text{out}})$" to all
9:     wait until $skip = \text{true}$
10:    for $k := 1, \ldots, n$ do
11:        abandon($(k, e)$)
12:    $L[e] \leftarrow \text{elect}(e)$                                   ▷ Leader election phase
13:    initiate sendAndGatherVCProofs($e, L[e], D_{key}, D_{lock}, D_{commit}$)      ▷ View-change phase
14:    wait until $vcCount = n - t - \epsilon n + 1$
15:    $skip :=$ false, $skipShares := \{\}$, $D_{key} := D_{commit} := D_{lock} := []$, $vcCount := 0, doneCount := 0$
16:
17: upon receiving "$e$, finished-proposal-promotion, $((v_{e,j}, \sigma_{commit,e,j}), \sigma_{\text{out},e,j})$" from $p_j$ for the first time do
18:    if threshold-verify($(((j, e), \text{COMMIT}), (v_{e,j}, \sigma_{commit,e,j})), \sigma_{\text{out},e,j}$) = true then
19:        $doneCount := doneCount + 1$
20:        if $doneCount \geq n - t$ then
21:            for $logId \in \{\text{PROMOTE}, \text{KEY}, \text{LOCK}, \text{COMMIT}\}$ do
22:                let $seqNo$ be the current counter value of the log corresponding to $logId$
23:                $seqNo := seqNo + 1$
24:                while $seqNo < (e * (n + 1))$ do
25:                    $\text{att}_{endLog} := \text{APPEND}(logId, seqNo, \bot)$
26:                    send "$e, fill\text{-}log, \text{att}_{endLog}$" to $p_j \in \rho(i)$
27:            $\xi := \text{threshold-sign}_i((skip, e))$
28:            send "$e$, skip-share, $\xi$" to all
29: upon receiving "$e$, skip-share, $\xi_j$" from $p_j$ do
30:    if share-validate($(skip, e), \xi_j, pk_j$) = true then
31:        $skipShares := skipShares \cup \xi_j$
32:        if $|skipShares| \geq \frac{n}{2} + 1$ then
33:            $\sigma_{skip} :=$ threshold-combine($(skip, e), skipShares$)
34:            send "$e, skip, \sigma_{skip}$" to all
35: upon receiving "$e, skip, \sigma_{skip}$" do
36:    if threshold-verify($(skip, e), \sigma_{skip)}$ = true then
37:        $skip :=$ true
38:    if "$skip$" message was not yet sent then
39:        send "$e, skip, \sigma_{skip}$" to all
40: upon receiving "$e$, view-change, $\text{att}_{key,j}, \text{att}_{lock,j}, \text{att}_{commit,j}, M_j$" from $p_j$ s.t. $|M_j| = (\frac{1}{2} + \frac{\epsilon}{2})d$, all parties
    with messages in $M_j$ are in $\rho(j)$, attestations in $M_j$ are valid, $\text{att}_{key,j}, \text{att}_{lock,j}, \text{att}_{commit,j}$ are either $\bot$ or valid
    attestations, and consistent with attestations in $M_j$
41:    let $\sigma_{key}$, $\sigma_{lock}$, and $\sigma_{commit}$ be the values stored in $\text{att}_{key,j}$, $\text{att}_{lock,j}$, and $att_{commit,j}$ respectively (or $\bot$)
42:    if threshold-verify($(((L[e], e), phase'), (v, \sigma_{phase}.qc_{\sigma_{\text{in}}})), (\sigma_{phase}.qc)$) = true, where $phase'$ is the phase before
       $phase$ for each $\sigma_{phase}$ then
43:        if $\text{att}_{commit,j} \neq \bot$ then
44:            decide $\sigma_{commit}.val$, where $\sigma_{commit}$ is the value in $\text{att}_{commit,j}$
45:        if $\text{att}_{lock,j} \neq \bot$ then $LOCK := \text{view}(\sigma_{lock})$, where $\sigma_{lock}$ is the value in $\text{att}_{lock,j}$
46:        if $\text{att}_{key,j} \neq \bot$ then $KEY := (v, \sigma_{\text{in}}) := (\sigma_{key}.val, \sigma_{key})$, where $\sigma_{key}$ is the value in $\text{att}_{key,j}$
47:        $vcCount := vcCount + 1$
```

Algorithm 6 Proposal-Promotion

```
1: procedure Proposal-Promotion($id, (v, \sigma_{\text{in}})$) //(for sender $S$)
2:     $\sigma_{key} := (\frac{n}{2}+1)$-PB-Initiate($(id, \text{PROMOTE}), (v, \sigma_{\text{in}})$)    ▷ PROMOTE phase
3:     $\sigma_{lock} := (\frac{n}{2}+1)$-PB-Initiate($(id, \text{KEY}), (v, \sigma_{key})$)    ▷ KEY phase
4:     $\sigma_{commit} := (\frac{n}{2}+1)$-PB-Initiate($(id, \text{LOCK}), (v, \sigma_{lock})$)    ▷ LOCK phase
5:     $\sigma_{out} := (\frac{n}{2}+1)$-PB-Initiate($(id, \text{COMMIT}), (v, \sigma_{commit})$)    ▷ COMMIT phase
6:     return $\sigma_{out}$
7:
8: procedure Proposal-Promotion($id$, validate(), validateNeighbor()) //(for party $p_i$)
9:     Invoke the following in parallel
10:    $(\frac{n}{2}+1)$-PB-Respond($(id, \text{PROMOTE})$, validate(), validateNeighbor())    ▷ PROMOTE phase
11:    $(\frac{n}{2}+1)$-PB-Respond($(id, \text{KEY})$, validate(), validateNeighbor()))    ▷ KEY phase
12:    $(\frac{n}{2}+1)$-PB-Respond($(id, \text{LOCK})$, validate(), validateNeighbor())    ▷ LOCK phase
13:    $(\frac{n}{2}+1)$-PB-Respond($(id, \text{COMMIT})$, validate(), validateNeighbor())    ▷ COMMIT phase
14:
15:    upon abandon($id$) do
16:        for $phase \in \{\text{PROMOTE}, \text{KEY}, \text{LOCK}, \text{COMMIT}\}$ do
17:            abandon($(id, phase)$)
```

Algorithm 7 sendAndGatherVCProofs (for party p_i)

```
1: procedure sendAndGatherVCProofs($e, L, D_{key}, D_{commit}, D_{lock}$)
2:     send "$e$, request-vc-ack"
3:     upon receiving set $M$ of $(\frac{1}{2}+\frac{\epsilon}{2})d$ messages "$e$, ack-vc, $(att_{key}, att_{lock}, att_{commit})$" s.t. $att_{key} = D_{key}[L]$,
       $att_{lock} = D_{lock}[L]$, and $att_{commit} = D_{commit}[L]$
4:         send "$e$, view-change, $D_{key}[L], D_{commit}[L], D_{lock}[L], M$" to all
5:     upon receiving "$e$, request-vc-ack" from $p_j \in \rho(i)$
6:         upon receiving all attestions from $p_j$'s KEY, LOCK, and COMMIT logs for sequence numbers $en$
7:             to $e(n+1)-1$ in fill-log messages or send messages
8:             if all attestations for positions corresponding to this view in the KEY, LOCK, and COMMIT log
9:                 are valid or $\perp$ and each non-null attestation is for a distinct proposal promotion then
10:                send "$e$, ack-vc, $(\text{att}_{key}, \text{att}_{lock}, \text{att}_{commit})$" to $p_j$, where the attestations are the attestations
11:                received from $p_i$ for $L$ and the corresponding phases (or $\perp$ where none was received)
```

Algorithm 8 createAttestations (for party p_i)

```
1: procedure createAttestations($((j, e), phase), (v, \sigma_{\text{in}})$)
2:     $logId := phase$
3:     $\text{att}_{logId} :=$ APPEND($logId, \perp, (j, (v, \sigma_{\text{in}}))$)
4:     if $phase \in \{\text{KEY}, \text{LOCK}, \text{COMMIT}\}$ then
5:         $D_{phase}[j] := \text{att}_{logId}$
6:     return $(\text{att}_{logId}, \perp)$
```

nomination (3-11), leader election (12), and view-change (13-15). Since the leader election stage uses a simple threshold-coin primitive similar to that in the VABA protocol, we abstract it out and do not describe it in detail.

Leader nomination stage. The leader nomination stage (lines 3-11 in Algorithm 5) starts with Proposal-Promotion, which consists of four sequential stages of provable broadcast (described in Algorithm 6), similar to that in HotStuff-M. There are n instances that are run in parallel and each party acts as a leader in one of them. The inputs to the instances are values corresponding to the highest key held by the leader, or any externally valid value if the party acting as the leader does not hold a key. The parties use the validation functions described in Table 2. In particular, the validateNeighbor() function ensures that in a span of n consecutive positions between en and $e(n+1)-1$ for view e, a proposal promotion instance uses only one position and all positions before it are used by other instances. If a party completes its own instance, it sends a *finished-proposal-promotion* message containing the value and a proof of commit to all parties. Otherwise, it waits until $n-t$ proposal promotion instances have completed. From a party's perspective, if $n-t$ instances have completed, i.e., it has received as many *finished-proposal-promotion* messages for this view), it appends $\perp$ to all its logs at all remaining positions for this view. Recall that in each log, every view has n slots dedicated to it, and they can be used in an arbitrary order during the provable broadcast calls in different proposal promotion instances. This step, thus, fills the remaining positions with $\perp$s and shares it with all neighboring parties $\rho(i)$ (Algorithm 5 lines 24-26).

At the end, each party sends a threshold signature with threshold $\frac{n}{2}+1$ to indicate the end of the leader nomination stage (lines 27-28). Once a party receives $\frac{n}{2}+1$ signature shares, it can combine and send the combined signature to all parties (lines 29-39). Such a proof allows parties to abandon all provable broadcast instances (lines 10-11 in Algorithm 5).

View-change stage. Once a leader $k := L[e]$ has been elected (line 12), parties initiate the view-change process. Since each parties' usage of logs is only known to its neighbors, it first collects confirmations from its neighbors on its state. Specifically, it sends a request to its neighbors to acknowledge the attestations sent for the proposal promotion of $L[e]$ (Algorithm 7, line 2). A neighbor, on receiving these requests, waits for all attestations corresponding to the view (Algorithm 2, lines 5-6). If all of the attestations in all of the logs are valid, i.e., each attestation is individually correct and is used for a distinct proposal promotion instance in each log, then the party sends a signed *ack-vc* message containing attestations from all logs corresponding to proposal promotion instance k (lines 9-11). On receiving $(\frac{1}{2}+\frac{\epsilon}{2})d$ signed messages from their neighbors, parties send their attestations along with the signed proofs to all parties (Algorithm 7, lines 3-4).

Each party then waits for $n-t-\epsilon n+1$ valid view-change messages (Algorithm 5, line 14). A view-change message sent by a party p_j is valid if it includes at least $(\frac{1}{2}+\frac{\epsilon}{2})d$ messages from parties in $\rho(j)$ containing the same set of attestations, att_{key}, att_{lock}, and att_{commit} are included in p_j's message (line40). In addition, for each σ_{phase} for $phase \in \{\textsc{key}, \textsc{lock}, \textsc{commit}\}$ that a party receives in *view-change* messages, it checks that $\sigma_{phase}.qc$ is a valid threshold signature resulting from the previous phase of the proposal promotion of $L[e]$ in the current view. If all the checks go through, a party delivers the value proposed by the $L[e]$ and, updates their KEY and $LOCK$ accordingly (lines 42-46).

Communication complexity. The leader nomination phase consists of n concurrent proposal promotion instances, each of which incurs $O(n)$ communication complexity due to the linear communication complexity of $(\frac{n}{2}+1)$-Provable-Broadcast (Theorem 3). Then, prior to the leader election phase, each party sends a constant number of constant-sized messages to each of the other parties. The elect function incurs $O(n^2)$ communication complexity. In the VIEW-CHANGE phase, each party sends a constant number of constant-sized messages to each of the other parties. Lastly, the protocol requires constant rounds in expectation (Lemma 7).

5.4 Security Proofs

Lemma 6. *For phase phase of proposal promotion in view e, if p_i obtains σ_{out} such that threshold-verify($(((i,e),phase),(v,\sigma_{in}.qc)),\sigma_{out}.qc$) = true and σ'_{out} such that threshold-verify($(((i,e),phase),(v',\sigma'_{in}.qc)),\sigma_{out}.qc$) = true, then $v = v'$.*

Proof. This follows from the provability property of $(\frac{n}{2}+1)$-Provable-Broadcast described in Lemma 2.

Lemma 7 (Uniqueness for proposal promotion within a view). *For a given view e, if p_i can deliver σ_{out}^{phase} from the $(\frac{n}{2}+1)$-Provable-Broadcast of phase phase $\in \{\text{PROMOTE}, \text{KEY}, \text{LOCK}, \text{COMMIT}\}$ in their proposal promotion, and another proof $\sigma_{out}^{phase'}$ from phase phase′ $\in \{\text{PROMOTE}, \text{KEY}, \text{LOCK}, \text{COMMIT}\}$ such that threshold-verify($(((i,e),phase),(v,\sigma_{in}^{phase}.qc)),\sigma_{out}^{phase}.qc$) = true and threshold-verify($(((i,e),phase'),(v',\sigma_{in}^{phase'}.qc)),\sigma_{out}^{phase'}.qc$) = true, then $v = v'$.*

Proof. If $phase = phase'$, the lemma follows from Lemma 6. If $phase'$ is the phase immediately after $phase$, this follows from Lemma 6 and the fact that in the validate() function, an honest party only votes for a proposal $(v,\sigma_{\text{in}}^{phase'})$ if threshold-verify$((((i,e)),phase),(v,\sigma_{\text{in}}^{phase'}.qc_{\sigma_{\text{in}}})),\sigma_{\text{in}}^{phase'}.qc)$ = true (Table 2), which can only be the case if $\sigma_{\text{in}}^{phase'}$ is the output of the $phase$ phase of view e, and therefore by Lemma 6, $v' = v$. Finally, if $phase'$ is two or more phases after $phase$ (e.g. $phase$ is PROMOTE and $phase'$ is COMMIT), then the same argument holds transitively.

Lemma 8. *For phase $\in \{\text{KEY}, \text{LOCK}, \text{COMMIT}\}$, if p_j has a valid proof σ_{out}^{phase} output from the phase phase of their proposal promotion in view e, then at least $\frac{n}{2}+1$ parties have inserted $(j,(v,\sigma'))$ into their logs corresponding to phase, where σ' is the output of the previous phase of this proposal promotion, in a position $ne \leq seqNo < n(e+1)-1$, and they have not placed $(j,(v',\sigma''))$ into a position $seqNo'$ in their log such that $ne \leq seqNo' < seqNo$.*

Proof. We will prove the lemma for $phase$ = COMMIT but the same argument holds for the remaining cases. If p_j has a valid proof $\sigma_{\text{out}}^{\text{COMMIT}}$ in view e, then $\frac{n}{2}+1-t \geq \epsilon n$ honest parties must have voted in the COMMIT phase. By Lemma 2, the ϵn honest parties have received attestations from at least $\frac{n}{2}+1$ parties, p_k. Each such attesation att_k received from their neighbor satisfies validateNeighbor$(((j,e),\text{COMMIT}),(v,\sigma_{commit}),(\text{att}_k,\perp),\perp)$ = true, where (v,σ_{commit}) is the proposal sent by p_j at the beginning of the COMMIT phase of the proposal promotion and the output of the LOCK phase of p_j's proposal promotion. The validateNeighbor() method Table 2 checks that none of the values appended to p_k's COMMIT log in positions corresponding to view e prior to the sequence number in att_k are for the same party's proposal promotion.

Lemma 9. *If a party sends a view-change message containing att_{phase} for phase $\in$ KEY, LOCK, COMMIT such that threshold-verify$((((L[e],e),phase^-),(v,\sigma_{phase}.qc_{\sigma_{in}})),\sigma_{phase}.qc)$ = true, and another party sends a view-change message containing att'_{phase} such that, threshold-verify$((((L,e),phase^-),(v',\sigma_{phase}.qc_{\sigma_{in}})),\sigma'_{phase}.qc)$ = true, where $phase^-$ is the phase before phase, $v' = v$.*

Proof. This proof follows from Lemma 6.

Lemma 10. *If an honest party has committed v during the* VIEW-CHANGE *phase of view e, then all honest parties set $LOCK = e$ by the end of view e.*

Proof. If an honest party has committed v, then it received a valid proof, σ_{commit}, for the completion of the LOCK phase of the $L[e]$'s proposal promotion such that threshold-verify$((((L[e],e),\text{LOCK}),(v,\sigma_{commit}.qc_{\sigma_{\text{in}}})),\sigma_{commit}.qc)$ = true. This implies that there is a valid σ_{lock} output from the KEY phase of $L[e]$'s proposal promotion. It follows from Lemma 8 that a set S of at least $\frac{n}{2}+1$ parties have inserted $(L[e],(v,\sigma_{lock}))$ into their LOCK logs in a valid position corresponding to view $view$, such that it does not conflict with any of the values appended before it for this view, and produced an attestation, att, proving it such that validateNeighbor$((((L[e],e),\text{LOCK}),(v,\sigma_{lock}),(\text{att},\perp),\perp)$

= true. At most $\frac{n}{2} - 1$ parties in the network do not have such an attestation in their logs, and we refer to this set as Y. In order to prove the lemma, we can assume that all the parties in Y are honest. Within S, there are at most four different categories of parties. The first category is Byzantine parties without $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors, of which there are fewer than ϵn (Lemma 1). By Lemma 9, these parties can only lie by producing a proof showing that they did not append anything for $L[e]$ in their LOCK logs for this view. We refer to the parties in this set as T_1. A second set of parties in S is the set T_2 of Byzantine parties with at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors. These parties may never send a view change message, and there are $t - |T_1|$ of these parties in S. The third set of parties in S is the set G_1 of fewer than ϵn honest parties (Lemma 1) without $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors, and they may never get a sufficient view change proof from their neighbors in order to send a view change message that will be accepted by honest parties. The final set of parties in S is the set G_2 of more than $\epsilon n + 1$ honest parties who each have at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ honest neighbors, and are therefore able to obtain a view change proof by calling sendAndGatherVCProofs() and send out valid view change messages to all the parties. During view change, each honest party waits for $n - t - \epsilon n + 1$ valid view change messages. The party can receive $\frac{n}{2} - 1$ messages from the set of honest parties in Y that aren't aware of the σ_{lock} from $L[e]$'s proposal promotion and fewer than ϵn messages from Byzantine parties in T_1 who successfully lie. The remaining view change messages, of which there are at least 2, that a party receives must be from a party that has σ_{lock} and cannot lie about it. This can be from a party in G_2. Even if no Byzantine party sends a view change message, there are more than $\frac{n}{2} + \epsilon n$ honest parties in the network in the sets Y and G_2 that are able to get valid view change proofs to send view change messages, and all parties hear about σ_{lock} through parties in G_2. Therefore, all honest parties find out about σ_{lock} and update $LOCK$.

Lemma 11. *If an honest party has set $LOCK = e$ at the end of the view e, then all honest parties have set $KEY := (v, \sigma_{key})$ at the end of view e where σ_{key} is the output of the* PROMOTE *phase of the proposal promotion of $L[e]$.*

Proof. This proof proceeds in the same manner as that of Lemma 10.

Theorem 6 (Safety). *If an honest party commits v in view e, and another honest party commits v' in view $e' \geq e$, then $v' = v$.*

Proof. By Lemma 7, only a single value v can receive σ_{commit} within a view. Hence, if $e' = e$, $v' = v$.

Suppose that an honest party has committed v in view e and another honest party commits v' in e', where $e' > e$ and $v' \neq v$. Since v was committed in view e, all parties must have $LOCK = e$ and $KEY = (v, \sigma_{key})$ for v after view e. If an honest party has committed v' in view $e' > e$, a valid σ'_{key} must have been formed on a value other than v in a view $> e$. Let e^* be the first view in which a σ_{key} is formed on a value $v^* \neq v$. By the conditions in the validate() function, an honest party only votes for $(v^*, \sigma^*_{\text{in}})$ if view$(\sigma^*_{\text{in}}) \geq LOCK$. Every honest party has $LOCK$ at least equal to e, so view(σ^*_{in}) should be greater than e. So, suppose σ^*_{in} was created in view e^{**} such that $e < e^{**} < e^*$. However, this contradicts the fact that e^* was the first view where a σ_{key} was created for a value other than v. Therefore, the lemma holds.

Lemma 12. *If all honest parties start view e with externally valid values, for all views $e' > e$, all values proposed by honest parties are externally valid.*

Proof. If $e = 1$, the lemma is trivially true. Consider a view $e' > e$. Each honest party proposes the value v in their KEY along with the proof of its validity σ_{in}. If an honest party set their $KEY = (v, \sigma_{\text{in}})$ upon receiving a *view-change* message containing it in view e'', it must be the case that threshold-verify$(((L[e''], e''), \text{PROMOTE}), (v, \sigma_{\text{KEY}}.qc_{\sigma_{\text{in}}}), (\sigma_{key}.qc))$ = true. In order for this to be the case, an honest party must have voted for $(v, \sigma_{\text{PROMOTE}})$. An honest party will only do so if ext-valid(v) = true. Therefore, the lemma holds.

Lemma 13. *If in view e, p_i, the leader of Proposal-Promotion instance (i, e), is honest, no honest party invokes abandon((i, e)) , and all messages sent by honest parties arrive, then the Proposal-Promotion completes and p_i receives a proof, σ_{out}, of its completion.*

Proof. Assuming that all honest parties start with externally valid values, by Lemma 12, the values proposed by all honest parties in view e are externally valid. Further, by Lemma 11, all honest parties have KEY equal to (v, σ_{key}) from a view at least as high as that of the highest $LOCK$ held by an honest party. Since p_i is honest, it must be the case that threshold-verify$((((L[\text{view}(\sigma_{\text{in}})], \text{view}(\sigma_{\text{in}})), \text{PROMOTE}), (v, \sigma_{\text{in}}.qc_{\sigma_{\text{in}}})), \sigma_{\text{in}}.qc)$ $=$ true (otherwise they wouldn't have adopted it to their KEY). In the PROMOTE phase, p_i proposes (v, σ_{in}). Since ext-valid(v) $=$ true, threshold-verify$((((L[\text{view}(\sigma_{\text{in}})], \text{view}(\sigma_{\text{in}})), \text{PROMOTE}), (v, \sigma_{\text{in}}.qc_{\sigma_{\text{in}}})), \sigma_{\text{in}}.qc)$ $=$ true, and view$(\sigma_{\text{in}}) \geq LOCK$ for all honest parties, every honest party will append $(i, (v, \sigma_{\text{in}}))$ to their PROMOTE logs and send the attestation to their neighbors. Since all messages sent by honest parties arrive, and by the termination property of $(\frac{n}{2}+1)$-Provable-Broadcast(Lemma 2), at least $\frac{n}{2}+1$ honest parties receive a sufficient number of attestations, att, from their neighbors such that validateNeighbor$(((i, e), \text{PROMOTE}), (v, \sigma_{\text{in}}), (\text{att}, \perp), \perp)$ $=$ true and send *vote* messages with threshold signature shares to p_i. p_i combines the threshold signature shares into a proof, σ_{key}. The arguments for the rest of the phases of p_i's proposal promotion follow from Lemma 3, the conditions laid out in Table 2, and the fact that no honest party abandons the proposal promotion.

Lemma 14. *If in view e, all messages sent by honest parties arrive and no honest party invokes abandon() on the proposal promotion of an honest party, then eventually skip = true for all honest parties.*

Proof. By Lemma 13 the proposal promotions of all honest parties complete, and therefore all honest parties p_j send messages "e, *finished-proposal-promotion*, (v, σ_{out})" such that threshold-verify$((((j, e), \text{COMMIT}), (v, \sigma_{\text{out}}.qc_{\sigma_{\text{in}}})), \sigma_{\text{out}}.qc)$ $=$ true. Eventually all honest parties get $doneCount = n - t$ and send *request-skip* messages with threshold signature shares. Therefore, some honest party gets at least $\frac{n}{2}+1$ $skipShares$ and sends out a proof, σ_{skip}, to all parties. Then every honest party gets σ_{skip} such that threshold-verify$((skip, e), \sigma_{skip})$ $=$ true, and sets $skip =$ true.

Lemma 15. *If one honest party sets skip = true, every honest party sets skip = true.*

Proof. An honest party sets $skip =$ true upon receiving a valid threshold signature σ_{skip}. Upon setting $skip =$ true, they forward σ_{skip} to the rest of the network. Therefore, all honest parties eventually set $skip =$ true.

Lemma 16. *If all honest parties start view e, and all messages sent by honest parties arrive, then view e completes.*

Proof. If an honest party has set $skip =$ true in view e, then by Lemma 14, all honest parties set $skip =$ true. If no honest party has set $skip =$ true, then no honest party ever invokes abandon(), and by Lemma 14, all honest parties eventually set $skip =$ true. Then, all honest parties participate in the leader election phase and elect $L[e]$. Every honest party then initiates sendAndGatherVCProofs(). By Lemma 1, at least $\frac{n}{2}+\epsilon(n)+1$ honest parties have at least $(\frac{1}{2}+\frac{\epsilon}{2})d$ honest neighbors, and therefore at least $\frac{n}{2}+\epsilon(n)+1$ valid *view-change* messages are sent to all parties. All honest parties get $vcCount = \frac{n}{2} - t - \epsilon(n) + 1$ and move on to the next view.

Lemma 17. *If the proposal promotion of the elected leader of view e has completed in view e, then all honest parties decide by the end of view e.*

Proof. Let $p_l = L[e]$ and v be the value proposed in p_l's proposal promotion. If p_l's proposal promotion completed in view e, they have a proof, σ_{out}, output from the COMMIT phase of their proposal promotion. By Lemma 8, this means that at least $\frac{n}{2}+1$ parties have inserted $(l, (v, \sigma_{commit}))$ into a position in their COMMIT logs corresponding to view e, such that it does not conflict with any of the values appended before it, where σ_{commit} is the output of the LOCK phase of p_l's proposal promotion in this view. Using the same argument as that in Lemma 10, we can show that all honest parties receive σ_{commit}. The rest of the lemma follows from this and the fact that honest parties commit after receiving the valid σ_{commit} from the leader's proposal promotion.

Lemma 18. *If an honest party invokes elect(e), then $\frac{n}{2}+2\epsilon n$ Proposal-Promotion instances have completed.*

Proof. An honest party invokes elect(e) after setting $skip = \text{true}$. They only set $skip = \text{true}$ upon receiving σ_{skip}, proof that $\frac{n}{2} + 1$ parties sent *skip-share* messages with threshold signatures. This means that at least one honest party must have sent a *skip-share* message. An honest party only does this after $doneCount \geq n - t$, where $doneCount$ is the number of *finished-proposal-promotion* messages they have received from parties with valid proofs of completion.

Theorem 7 (Liveness). *If all honest parties start with externally valid values, with probability* $> \frac{1}{2}$, *all honest parties decide in a given view.*

Proof. By Lemma 14, all honest parties eventually set $skip = \text{true}$. Upon setting $skip = \text{true}$, every honest party invokes elect(e). By Lemma 18, $\frac{n}{2} + 2\epsilon n$ proposal promotions must have completed. The probability that the party that is elected as the leader is one of the ones whose proposal promotions completed is $\frac{n-t}{n} > \frac{1}{2}$. By Lemma 17, if the elected leader's proposal promotion completed, all honest parties decide in view e.

6 Related Work

Trusted hardware and consensus. Trusted hardware can be classified into two categories depending on the computations it can provide. The more powerful class is capable of running arbitrary specified code in a trusted execution environment (TEE). The protected execution state is encrypted by the trusted module and written to a specified memory range that only the trusted module can access while the code is running (e.g. Intel SGX [17], Flicker [34], Aegis [46], XOM [31], and Bastion [11]). They have been used to provide confidentiality [17, 46, 31, 11, **?**] as well as to improve resilience and performance in the context of consensus protocols [7, 23, 44, 8, 2, 32]. However, the trusted computing bases of such platforms tend to grow as they increase in their generality, up to an including extensive libraries and OS components (e.g., [48]). Consequently, these platforms can present a large attack surface, giving way to attacks from outside the TEE (e.g., [13]).

Our work focuses on using small trusted hardware with a fixed, limited functionality (e.g., YubiKeys [47]). There have been several works using such a hardware to improve the performance of BFT protocols [25, 51, 52, 15, 30, 40, 3]. Notable works include A2M [15] and TrInc [30]. Chun et al. [15] show how, by introducing append-only (A2M) logs in the trusted hardware component of all processors in a network, the fault tolerance of BFT protocols can be increased to minority faults. They show an implementation of PBFT that withstands minority faults using A2M logs. However, simply applying their approach to a BFT protocol can increase the communication complexity by at least a factor of n due to the communication pattern of the protocol. In TrInc [30], Levin et al. show how A2M logs can be implemented with a small trusted monotonic counter, a key, and a small amount of trusted storage.

Expander graphs and consensus. Expander graphs have been used in the context of consensus protocols in works in the past [28, 27, 35]. Chlebus et al. [14] present an algorithm that solves consensus in the crash fault setting such that the per-process communication complexity is polylogarithmic in the number of processors. King and Saia's protocol for leader election withstands a one-third Byzantine adversary in synchrony using expanders to achieve polylogarithmic per-process communication complexity. They extend this work to obtain $o(n^2)$ total bits of communication against an adaptive adversary. Recently, Momose and Ren [35] used expanders to solve Byzantine agreement against a minority corruption. Their work uses expanders to detect equivocation under synchrony. One could also consider the use of random sampling to be a randomized method to obtain the properties of expanders [21, 42, 22].

Non-equivocation. Over the past two decades, there has been various work in the space of non-equivocation [15, 16, 43, 6, 33]. Clement et al. [16] defined non-equivocation as when a process sends different messages to different processes during a single round of the protocol when, by the protocol, it should have sent the same message to the other processes. They then proceed to study the power of non-equivocation, showing that non-equivocation alone is not sufficient to increase the threshold of Byzantine processes in a network when trying to reach agreement. It is necessary to also have digital signatures, or another form of transferable authentication, to increase the resiliency of the network. Ruffing et al. [43] present non-equivocation contracts, which reveal the Bitcoin credentials of an equivocating party in order to penalize equivocation. Backes et al. [6] show how to use non-equivocation to improve the resilience of asynchronous MPC to match that of synchronous MPC, which tolerates minority corruption. Our work expands on these works by showing how non-equivocation implemented by the use of trusted hardware can be combined with expander graph techniques to increase the fault tolerance of BFT protocols without increasing the communication complexity.

References

1. Trusted computing group.
2. Hyperledger sawtooth, 2019.
3. Ittai Abraham, Marcos K Aguilera, and Dahlia Malkhi. Fast asynchronous consensus with optimal resilience. In *International Symposium on Distributed Computing*, pages 4–19. Springer, 2010.
4. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.
5. Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.
6. Michael Backes, Fabian Bendun, Ashish Choudhury, and Aniket Kate. Asynchronous mpc with a strict honest majority using non-equivocation. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 10–19, 2014.
7. Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237, 2017.
8. Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint arXiv:1805.08541*, 2018.
9. Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.
10. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
11. David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.
12. Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.
13. Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2013.
14. Bogdan S Chlebus, Dariusz R Kowalski, and Michal Strojnowski. Fast scalable deterministic consensus for crash failures. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 111–120, 2009.
15. Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.
16. Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 301–308, 2012.
17. Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.
18. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
19. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.
20. Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.
21. Seth Gilbert and Dariusz R Kowalski. Distributed agreement with optimal communication complexity. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 965–977. SIAM, 2010.

22. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi, and Yann Vonlanthen. Scalable byzantine reliable broadcast (extended version). *arXiv preprint arXiv:1908.01738*, 2019.
23. Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016, 2016.
24. Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.
25. Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308, 2012.
26. Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.
27. Valerie King and Jared Saia. Breaking the $o(n^2)$ bit barrier: Scalable Byzantine agreement with an adaptive adversary. *Journal of the ACM*, 58(4):1–24, 2011.
28. Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *17^{th} ACM-SIAM Symposium on Discrete Algorithms*, pages 990–999, 2006.
29. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.
30. Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.
31. David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices*, 35(11):168–177, 2000.
32. Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1):139–151, 2018.
33. Mads Frederik Madsen and Søren Debois. On the subject of non-equivocation: Defining non-equivocation in synchronous agreement systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 159–168, 2020.
34. Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.
35. Atsuki Momose and Ling Ren. Optimal communication complexity of byzantine consensus under honest majority. *arXiv preprint arXiv:2007.13175*, 2020.
36. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.
37. Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.
38. Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint arXiv:2002.07539*, 2020.
39. Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.
40. Vincent Rahli, Francisco Rocha, Marcus Völp, and Paulo Esteves-Verissimo. Deconstructing minbft for security and verifiability. *Grand Region Security and Reliability Day.*
41. Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. Practical synchronous byzantine consensus. *arXiv preprint arXiv:1704.02397*, 2017.
42. Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. *Available [online].[Accessed: 4-12-2018]*, 2018.
43. Tim Ruffing, Aniket Kate, and Dominique Schröder. Liar, liar, coins on fire! penalizing equivocation by loss of bitcoins. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 219–230, 2015.
44. Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. Ccf: A framework for building confidential verifiable replicated services. *Technical Report MSR-TR-201916*, 2019.
45. Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.
46. G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.
47. Suresh Thiru, Shamalee Deshpande, and Stina Ehrensvard. Yubikey strong two factor authentication, Jan 2021.

48. Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, July 2017.
49. Salil P. Vadhan. *Pseudorandomness*, volume 7 of *Foundations and Trends in Theoretical Computer Science*. Now Publishers, Inc., 2012.
50. Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [53].
51. Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 10–19. IEEE, 2010.
52. Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.
53. Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [50].
54. Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

A Proofs for Expander Graph Lemmas

Lemma 19. *For every constant* $0 < \alpha < \beta < 1$ *and sufficiently large* n*, there exists a* d*-regular graph that is an* (n, α, β)*-expander.*

Proof. For this proof, we will show a randomized way to construct a d-regular graph $G_{n,\alpha,\beta}$. Then, we will show that with high probability, it satisfies the lemma.

Let $\Gamma(V, G)$ refer to the set of neighbors of the vertices V in a graph G. Consider a random degree-d graph G constructed by taking the union of d random perfect matchings (assume that n is even; if n is odd, we can add a dummy node or assign a vertex to two neighbors). In order to ensure that each vertex has degree exactly d, one can construct each of the perfect matchings in the following way. Create the first perfect matching by taking one vertex at a time and matching it to a random unmatched vertex in the graph, repeating until all vertices have been matched. Repeat this process for d perfect matchings without replacement, so that for the kth perfect matching, each vertex is matched to a random vertex from the set of vertices that it hasn't been matched to in a previous perfect matching.

Consider a perfect matching P, a set of αn nodes, S, and a set of βn nodes, T. Now, consider the matching of the first set of αn nodes, S, in the first perfect matching (the perfect matching that results in a graph with degree 1). The probability that the first vertex that is matched is matched to a vertex in T is $\frac{\beta n}{n}$. Since $\alpha < \beta$, and we match without replacement, the probability of choosing a match in T for the ith node in S after all of the previous matchings of nodes in S have been to nodes in T can only be less than this quantity. Since it is possible that $S \subset T$, and nodes in S are matched to each other, we only multiply this quantity $\frac{\alpha n}{2}$ times. We are therefore able to obtain the following upper bound for the probability that in each perfect matching P, for any set of αn nodes S, and any set of βn nodes T, $\Gamma(S, P) \subseteq T$:

$$Pr[\Gamma(S,P) \subseteq T] \leq (\frac{\beta n}{n})^{\frac{\alpha n}{2}} = \beta^{\frac{\alpha n}{2}} \tag{1}$$

Using this, the probability that any set of αn vertices does not expand to more than βn other vertices, i.e. $|\Gamma(S, G)| \leq \beta n$ for any set S, is bounded above by:

$$\binom{n}{\alpha n}\binom{n}{\beta n}\beta^{\frac{\alpha n d}{2}} \tag{2}$$

$$\leq (\frac{e}{\alpha})^{\alpha n}(\frac{e}{\beta})^{\beta n}\beta^{\frac{\alpha n d}{2}} \tag{3}$$

$$\leq [e^{\alpha+\beta}((\frac{1}{\alpha})^{\alpha}(\frac{1}{\beta})^{\beta})\beta^{\frac{\alpha d}{2}}]^n \tag{4}$$

For a sufficiently large constant d, the above probability is exponentially small, which means that with high probability, a graph randomly chosen with the above procedure is an (n, α, β)-expander. Thus, $G_{n,\alpha,\beta}$ exists.

Lemma 20. *There exists an expander graph $G_{n,\epsilon,\beta}$ with degree d, $0 < \epsilon < \frac{1}{2}$, $\epsilon < \beta < 1$, such that for any set S of $(\frac{1}{2} + \epsilon)n$ nodes, there exists a set Q of more than $\frac{n}{2}$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S.*

Proof. From Lemma 19, we know that a $G_{n,\epsilon,\beta}$-expander exists. For the rest of the proof, we show that with high probability an expander graph with sufficiently high degree constructed using the randomized procedure outlined in the proof for Lemma 19 satisfies the lemma.

Consider a set T of $(\frac{1}{2} - \epsilon)n$ nodes in our expander graph G. Let S be the set of nodes in G that are not in T. If we show that with high probability, it is not the case that ϵn nodes in S have more than $(\frac{1}{2} - \frac{\epsilon}{2})d$ neighbors in T, then there must be a set of nodes Q of size greater than $\frac{n}{2}$ in S that satisfies the lemma. Therefore, using the same technique as that in the proof for Lemma 19, we first bound the probability that for a given set of nodes R of size ϵn in a perfect matching P, all nodes in R have neighbors in a set T of size $(\frac{1}{2} - \epsilon)n$, where R and T are pairwise disjoint.

$$Pr[\Gamma(R, P) \subseteq T] \leq (\frac{(\frac{1}{2} - \epsilon)n}{n})^{\epsilon n} = (\frac{1}{2} - \epsilon)^{\epsilon n} \tag{5}$$

Then the probability that there does not exist a set Q of more than $\frac{n}{2}$ nodes that satisfies the statement in the lemma is bounded by:

$$\binom{n}{\epsilon n}\binom{(1-\epsilon)n}{(\frac{1}{2}-\epsilon)n}(\frac{1}{2} - \epsilon)^{\epsilon n d(\frac{1}{2}-\frac{\epsilon}{2})} \tag{6}$$

$$\leq [(\frac{e}{\epsilon})^{\epsilon}(\frac{e(1-\epsilon)}{(\frac{1}{2}-\epsilon)})^{(\frac{1}{2}-\epsilon)}(\frac{1}{2} - \epsilon)^{\epsilon d(\frac{1}{2}-\frac{\epsilon}{2})}]^n \tag{7}$$

$$\leq [e(\frac{1}{\epsilon})^{\epsilon}(\frac{1-\epsilon}{\frac{1}{2}-\epsilon})^{(\frac{1}{2}-\epsilon)}(\frac{1}{2} - \epsilon)^{\epsilon d(\frac{1}{2}-\frac{\epsilon}{2})}]^n \tag{8}$$

Again, for sufficiently large d, the above probability is exponentially small. Thus, with high probability the lemma holds.

Lemma 21. *There exists an expander graph $G_{n,\epsilon,\beta}$ with degree d, $0 < \epsilon < \frac{1}{2}$, $\epsilon < \beta < 1$ such that for any partition of its nodes into blocks T and Q where $|T| = (\frac{1}{2} - 2\epsilon)n$ and $|Q| = (\frac{1}{2} + 2\epsilon)n$, there exists a set $T' \subseteq T$, $|T'| > (\frac{1}{2} - 3\epsilon)n$, such that each node in T' has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in Q.*

Proof. From Lemma 19, we know that a $G_{n,\epsilon,\beta}$-expander exists. For the rest of the proof, we show that with high probability an expander graph with sufficiently high degree constructed using the randomized procedure outlined in the proof for Lemma 19 satisfies the lemma.

Consider a set T of $(\frac{1}{2} - 2\epsilon)n$ nodes in our expander graph G. Let R be be a set of nodes in T of size ϵn. Using the same technique as that in the proof for Lemma 19, we first bound the probability that in a given perfect matching P, all nodes in R have neighbors in T.

$$Pr[\Gamma(R, P) \subseteq T] \leq (\frac{(\frac{1}{2} - 2\epsilon)n}{n})^{\frac{\epsilon n}{2}} = (\frac{1}{2} - 2\epsilon)^{\frac{\epsilon n}{2}} \tag{9}$$

Then the probability that ϵn nodes in T have more than $(\frac{1}{2} - \frac{\epsilon}{2})d$ neighbors in T is bounded by:

$$\binom{n}{(\frac{1}{2}-2\epsilon)n}\binom{(\frac{1}{2}-2\epsilon)n}{\epsilon n}(\frac{1}{2} - 2\epsilon)^{\frac{\epsilon n d}{2}(\frac{1}{2}-\frac{\epsilon}{2})} \tag{10}$$

$$\leq [(\frac{e}{\frac{1}{2}-2\epsilon})^{\frac{1}{2}-2\epsilon}(\frac{e(\frac{1}{2}-2\epsilon)}{\epsilon})^{\epsilon}(\frac{1}{2}-2\epsilon)^{\frac{\epsilon d}{2}(\frac{1}{2}-\frac{\epsilon}{2})}]^n \tag{11}$$

$$\leq [e(\frac{1}{\frac{1}{2}-2\epsilon})^{\frac{1}{2}-2\epsilon}(\frac{\frac{1}{2}-2\epsilon}{\epsilon})^{\epsilon}(\frac{1}{2}-2\epsilon)^{\frac{\epsilon d}{2}(\frac{1}{2}-\frac{\epsilon}{2})}]^n \tag{12}$$

Again, for sufficiently large d, the above probability is exponentially small. Thus, with high probability the lemma holds.

Lemma 22. *There exists an expander graph $G_{n,\epsilon,\beta}$ with degree d, $0<\epsilon<\frac{1}{2}$, $\epsilon<\beta<1$, such that for any set S of $(\frac{1}{2}+2\epsilon)n$ nodes, there exists a set Q of more than $(\frac{1}{2}+\epsilon)n$ nodes, $Q\subseteq S$, and every node in Q has at least $(\frac{1}{2}+\frac{\epsilon}{2})d$ neighbors in S.*

Proof. This lemma follows directly from Lemma 20, as any graph that violates this property also violates the property in Lemma 20.

The following lemma and theorem can be found in any resources on expander graphs, such as Hoory et al. [24].

Lemma 23 (Expander Mixing Lemma). *Let $G=(V,E)$ be a d-regular graph and let $S,T\subseteq V$. Then,*

$$||E(S,T)|-\frac{d|S||T|}{n}|\leq \lambda(G)\cdot d\sqrt{|S|(1-|S|/n)|T|(1-|T|/n)} \tag{13}$$

where $|E(S,T)|$ is the number of edges between the two sets (counting edges contained in the intersection of S and T twice) and $\lambda(G)$ is the second largest eigenvalue of the adjacency matrix of G.

Theorem 8 (Theorem 4.12 of Vadhan [49], restated). *For any constant $d\in N$, a random d-regular n-vertex graph satisfies $\lambda(G)\leq 2\sqrt{d-1}/d+O(1)$ with probability $1-O(1)$ where $\lambda(G)$ is the second largest eigenvalue of the adjacency matrix of G and both $O(1)$ terms vanish as n approaches ∞ (and d is held constant).*

Lemma 24. *For all sufficiently large integers n and positive constants ϵ and β such that $0<\epsilon<\beta<1$ there exists an d-regular expander $G_{n,\epsilon,\beta}$ such that for any set S of ϵn nodes and any set T of $(\frac{1}{2}+\frac{\epsilon}{c})n$ nodes, where $c>2$, the number of edges with one vertex in S and one vertex in T is less than $(\frac{1}{2}+\frac{\epsilon}{2})\epsilon dn$.*

Proof. By Lemma 19, we know that a random d-regular expander $G_{n,\epsilon,\beta}$ exists for a sufficiently large constant d. By the Expander Mixing Lemma [24], we know that for any two sets of vertices S and T where $|S|=\epsilon n$ and $|T|=(\frac{1}{2}+\frac{\epsilon}{c})n$, the number of edges between the vertices in S and those in T, $E(S,T)$, in a d-regular expander graph G is upper bounded by:

$$E(S,T)\leq \lambda d\sqrt{\epsilon n(1-\epsilon)\left(\frac{1}{2}+\frac{\epsilon}{c}\right)n\left(\frac{1}{2}-\frac{\epsilon}{c}\right)}+\frac{\epsilon dn}{2}+\frac{\epsilon^2 dn}{c} \tag{14}$$

In order to satisfy the lemma, we need:

$$E(S,T)\leq \lambda\sqrt{(\epsilon n-\epsilon^2 n)\left(\frac{1}{2}+\frac{\epsilon}{c}\right)\left(\frac{n}{2}-\frac{\epsilon n}{c}\right)}<\frac{\epsilon^2 n}{2}-\frac{\epsilon^2 n}{c} \tag{15}$$

Since G is a random d-regular expander graph, we can upper bound $\lambda(G)$ using [49, Theorem 4.12]:

$$\lambda\leq\frac{2\sqrt{d-1}}{d}+O(1)<\frac{\frac{\epsilon^2}{2}-\frac{\epsilon^2}{c}}{\sqrt{\frac{\epsilon}{4}-\frac{\epsilon^2}{4}-\frac{\epsilon^3}{c^2}+\frac{\epsilon^4}{c^2}}} \tag{16}$$

With some simplification, and as the $O(1)$ term goes to 0 as n goes to infinity, we get:

$$d>\frac{\epsilon}{(\frac{\epsilon^2}{2}-\frac{\epsilon^2}{c})^2} \tag{17}$$

Which is satisfied by sufficiently large constant d.

Proof of Lemma 1

Proof. By Lemma 24, we know that there exists an expander $G_{n,\epsilon,(1-\frac{\epsilon}{c})}$ with sufficiently large constant degree d, such that every set T of $(\frac{1}{2}+\frac{\epsilon}{c})n$ nodes has fewer than $(\frac{1}{2}+\frac{\epsilon}{2})\epsilon dn$ edges to S. Further, by Lemmas 20, 21, 22, we know that there is a randomized construction for a d-regular expander for sufficiently high degree d that satisfies properties 1-3 with high probability. We deterministically choose a graph that satisfies these properties. For property 4, consider an arbitrary set S of ϵn nodes in the graph. Suppose that we create a set U' consisting of $(\frac{1}{2}+\frac{\epsilon}{2})d$ neighbors of each node in the set S such that $|U'| \leq \frac{n}{2}$. If we consider the multiset of $(\frac{1}{2}+\frac{\epsilon}{2})d$ neighbors of each node in S, the size of the multiset is $(\frac{1}{2}+\frac{\epsilon}{2})\epsilon dn$. By the construction of our expander, every set of ϵn nodes expands to more than $(1-\frac{\epsilon}{c})n$ nodes. Refer to the set of $(1-\frac{\epsilon}{c})n$ nodes that S expands to as Y. It follows that within Y, there must be a set T of $(\frac{1}{2}+\frac{\epsilon}{c})n$ nodes that contains U' such that there are more than $(\frac{1}{2}+\frac{\epsilon}{2})\epsilon dn$ edges with one vertex in S and one in T. We have arrived at a contradiction, as this violates that the graph satisfies the property in Lemma 24. Therefore the lemma holds.

B View synchronization.

In this section, we describe protocols for view synchronization to ensure that honest parties are in the same view for a sufficiently long time. Informally, the view synchronization property states that there should be an infinite number of views e with honest leaders such that every honest party is in view e for at least δ time, where δ is a parameter indicating the desired length of a view. There are a few different ways to achieve view synchronization, each with tradeoffs. One such implementation uses the view doubling method as described in PBFT [10]. One way to implement view doubling is such that there is a predefined ordering of the parties, and each view maps deterministically to one party as a leader. After every n views, the protocol cycles back through the list of parties in order. At the beginning of the protocol, each party starts with the same timeout interval δ. In each view, they time out only after δ time and then double the value of δ prior to beginning the next view. We refer the reader to [37] for a proof that this implementation results in a synchronous view after GST. Since no messages are exchanged in view doubling, its application for view synchronization does not increase the communication complexity of the underlying protocol. Thus, such a protocol should directly work to synchronize views in our protocol too. However, it does have the disadvantage that it does not allow the protocol to proceed at the speed of the network. Further, the view timeout can grow unbounded.

Another proposal is to define a set time interval δ for a view if we assume clocks to be incrementing at the same speed and that all parties start the protocol at the same time. Starting from the first view, a party always proceeds to the next view after δ time. This protocol does not result in unbounded growth of the length of each view. Since this protocol does not increase communication complexity, it can be used with our protocol too. However, like the view doubling proposal, this protocol is not responsive.

Expected linear communication complexity, constant latency view synchronization [38]. Naor et al. [38] present a view synchronization protocol that achieves expected linear communication complexity and expected constant latency against a Byzantine adversary that corrupts fewer than $\frac{1}{3}$ of the parties in the network. It also allows the protocol to proceed at the speed of the network under optimistic conditions. In this section, we present a brief overview of the solution presented in [38]. We then present our modified version of the view synchronization protocol with the same expected communication complexity and latency against a Byzantine adversary that controls a minority of the parties in the network.

The protocol presented in [38] assumes a network of n parties such that $t < \frac{n}{3}$ parties are corrupted by a static oblivious Byzantine adversary. The protocol uses PKI to verify the sender of a message and assumes a shared source of randomness. Since the adversary is oblivious, it does not know the randomness prior to corrupting parties. For each view, every party that queries the shared source of randomness receives the same random sequence of $t+1$ parties such that the k-th party in the sequence is the k-th relay for that view. The sequence of relays for a given view is selected uniformly at random from the set of parties without replacement. Finally, the protocol uses a threshold signature scheme to create constant-sized threshold signatures with two different thresholds: $t+1$ and $2t+1$.

The protocol uses two higher level functions: propose-view(e) and wish-to-advance(). When a party times out of a view of the higher level protocol, it invokes wish-to-advance(), which signals the beginning of the view synchronization protocol for the subsequent view. Upon receiving the propose-view(e) signal from the view synchronization protocol, a party begins execution of view e. There are three message types: **wish**, **move**, and **finalize**.

We now describe the execution of the view synchronization protocol from the perspective of a party. In the subsequent paragraph, we describe the execution from the perspective of a relay. When a party begins the execution of the view synchronization protocol for view e, it obtains the sequence R of $t+1$ relays for this view from the shared source of randomness. The party then iterates through the list of relays in order, attempting to obtain a $\sigma_{wish,e}$, a threshold signature indicating that at least $t+1$ parties invoked wish-to-advance() in view $e-1$. During this time, the party maintains a count variable, *attempted* $[e]$, which it uses to keep track of the relays it has tried to obtain a $\sigma_{wish,e}$ from for view e and that is initialized to 1. Upon entering the execution of the view synchronization protocol for view e after invoking wish-to-advance() in view $e-1$, a party sends a message "**wish**, e, 1" to the 1st relay in the sequence, $R[1]$. Note that this message actually contains a threshold signature share on the contents of the message, however for simplicity we write the message this way. Upon waiting 2Δ and not receiving a $\sigma_{wish,e}$ from $R[1]$, where Δ corresponds to the message delay of the network, a party increments its *attempted*$[e]$ counter and sends a message "**wish**, e, *attempted*$[e]$" to the second relay, $R[2]$, in the sequence. The party continues timing out and trying the next relay in the sequence until they receive a $\sigma_{wish,e}$ from a relay. Note that there is guaranteed to be at least one honest relay in R. Upon receiving a $\sigma_{wish,e}$ from relay r in the sequence, a party sends a message "**move**, e, i" to relay r (even if it did not yet invoke wish-to-advance() from view $e-1$), where i is the sequence number of relay r in R. Upon waiting 2Δ and not receiving a $\sigma_{move,e}$ from relay r, and if *attempted*$[e] < t+1$, the party picks up from where it left off sending **wish** by incrementing *attempted*$[e]$ and sending "**wish**, e, *attempted*$[e]$" to relay $R[$*attempted*$[e]]$. Upon receiving $\sigma_{move,e}$ from relay r' in the sequence, a party enters view e and sends **finalize** to relay r'. 2Δ after sending **finalize** to relay r' and not receiving $\sigma_{finalize,e}$, and if *attempted*$[e] < t+1$, the party picks up where it left off sending **wish**, increments *attempted*$[e]$ and sends "**wish**, e, *attempted* $[e]$" to relay $R[$*attempted*$[e]]$. Upon receiving **finalize**, a party stops this execution of the view synchronization protocol.

As the i-th relay of view e, a party p_r acts as follows. When it receives $t+1$ "**wish**, e, i" messages, relay p_r combines them into a threshold signature $\sigma_{wish,e}$ and sends it to all the parties in the network. Upon receiving $2t+1$ "**move**, e, i" (or "**finalize**, e, i") messages, the party combines them into a $\sigma_{move,e}$ (or $\sigma_{finalize,e}$) and sends the threshold signature to all parties in the network. Note that the threshold signatures are on the message type, view, and relay number. This means that each relay must collect their own threshold signature, so Byzantine relays cannot arbitrarily participate in the protocol.

We briefly discuss three properties guaranteed by the synchronizer for correctness and liveness: validity, stabilization, and progress. **Validity** states that an honest party should only move to a new view e if some honest party requested to move to this view. This property is trivially guaranteed by the fact that the σ_{wish} threshold is $t+1$. Stabilization and progress are two properties that ensure liveness. **Stabilization** ensures, for any time T, the stabilization of at least $t+1$ honest parties to the same maximum view e at, or after, T. In addition, if the first relay for view e is honest, and the time T when the first honest party to request to move to view e is $T \geq$ GST, then all the honest parties enter view e within a constant amount of time from T. This property ensures that there are always a sufficient number of honest parties in the highest view to request to move to the subsequent view. The protocol guarantees this property before GST, and after GST if the first relay for view e is not honest, because an honest party only stops sending **wish** messages to relays once it receives a $\sigma_{finalize,e}$ from a relay, ensuring that eventually an honest relay receives a sufficient number of **wish** messages to broadcast a $\sigma_{wish,e}$. Once the first honest party receives a $\sigma_{finalize,e}$, at least $t+1$ honest parties must have received a $\sigma_{move,e}$ and moved to the new view. After GST if the first relay for view e is honest, since an honest party moved to view e, it must have received a $\sigma_{move,e}$ at time T, and the first relay in the sequence must have received at least $t+1$ **wish** from honest parties and broadcasted a $\sigma_{wish,e}$ by time $T+\Delta$. By time $T+3\Delta$, the first relay receives a sufficient number of **move** messages to broadcast a $\sigma_{move,e}$, and all relays enter the view by time $T+4\Delta$. Finally, **progress** states that for a given time T, if a set H of $t+1$ honest parties call wish-to-advance() while in the maximum view of any honest party at T, e, by time T_0 (and no honest parties do so in a view greater than e), then there is at least one honest party in view $e+1$ from some time T_1 onward. Further, if $T_0 \geq$ GST, and the first relay in the sequence for view $e+1$ is correct, then all honest parties enter view $e+1$ a constant amount of time after T_0. Clearly, by validity, no honest party moves to view $e+2$ or higher, so the parties in H must be in view e or $e+1$. The proof of this property then follows from the fact that an honest party only stops sending **wish** messages to relays for view $e+1$ after it receives $\sigma_{finalize,e+1}$. After GST, and if the first relay for view $e+1$ is honest, the proof follows from the fact that this relay receives at least $t+1$ **wish** messages by time $T+\Delta$, and by time $T+4\Delta$, all correct processes receive a $\sigma_{move,e+1}$ from this relay and move to view $e+1$.

With the properties of validity, stabilization, and progress, [38] goes on to prove that by defining the leader of a view as the first relay for that view, since relays are chosen randomly, there are an infinite number of synchronized views with honest leaders after GST. The proof that this protocol achieves expected linear communication complexity and constant latency relies on the fact that there are an expected constant number of bad relays ($\leq \frac{3}{2}$) before reaching a good relay in a relay sequence. They show that there is a constant amount of time in expectation between each stabilized view (and until the first stabilized view) after GST. In addition, the random selection of relays ensures that each party communicates with a constant number of relays in expectation for each run of view synchronization after GST. We refer the reader to [38] for the detailed proofs.

Generalizing Naor-Keidar view synchronization to our setting. We now describe our modified version of the view synchronization protocol. Compared to [38], we allow $t \leq (\frac{1}{2}-\epsilon)n$ parties in the network to be corrupted. We can therefore no longer use a threshold signature size for σ_{move} and $\sigma_{finalize}$ that ensures $t+1$ honest parties have contributed to their creation. However, we can still gain the properties that we need through the use of the expander graph, and with the assumption of synchrony after GST. Observe that after GST, the expander graph that connects all the parties in the network gives us additional properties. Our expander graph has the property that the neighbor set of any set of ϵn honest parties contains more than $\frac{n}{2}$ honest parties (this follows from the fact that we use an $G_{n,\epsilon,(1-\frac{\epsilon}{c})}$ expander where $c > 2$, and at most $t \leq (\frac{1}{2}-\epsilon)n$ parties are Byzantine). This means that, after synchrony, if ϵn honest parties send a message to their neighbors at time T, more than $\frac{n}{2}$ honest parties receive the message by time $t+\Delta$. So, although we cannot use a threshold signature size that includes a majority of the honest parties, if our threshold signatures include at least ϵn honest parties, we can ensure that if ϵn honest parties move to a new view, they can send the σ_{move} they received to more than $\frac{n}{2}$ honest parties through the expander graph, so that a majority of the honest parties move to the new view within a constant amount of time. By setting δ accordingly, we achieve view synchronization with expected constant latency. Since our expander has constant degree, the communication complexity also remains linear in expectation. Note that this protocol does not require the use of trusted hardware at all, so we are able to create the threshold signatures outside of the trusted hardware. Thus, this is a strict generalization of [38]. We present the pseudocode in Algorithm 9. In the pseudocode, we use relay(e, k) to refer to a call to the shared source of randomness to return the process that is the relay for view e, sequence number k. For the sake of simplicity, we define an *attempted* [] array to keep track of number of attempted relays for a given view, however in practice, a single counter can be used. The parts of the protocol that differ from the original protocol are in blue.

We now present the intuition for how our modified view synchronization protocol achieves the properties of validity, stability, and progress. As in the original protocol, validity is trivially guaranteed by the $t+1$ threshold for σ_{wish}. For stabilization, prior to GST and after GST if the leader of view e is not honest, if an honest party receives $\sigma_{finalize,e}$, then ϵn honest parties received $\sigma_{move,e}$ and sent it to their neighbors, meaning that at least $\frac{n}{2}+1$ honest parties receive it and move to view e. If no correct process recieves $\sigma_{finalize,e}$, the at least ϵn honest parties that contributed to the $\sigma_{move,e}$ received by the first honest party to move to view e sent $\sigma_{wish,e}$ to their neighbors. Therefore, at least $\frac{n}{2}+1$ honest parties commence the view synchronization protocol for view e, and eventually some honest relay receives $t+1$ **wish** messages and completes the rest of the stages of the view synchronization protocol for view e. Therefore, all honest parties eventually move to view e. If relay(e, 1) is honest, and the first party to move to view e does so at time $T_0 \geq$ GST, ϵn honest parties received $\sigma_{wish,e}$ by time T_0 and sent it to their neighbors, who receive it

Algorithm 9 Expected Linear Communication and Constant Latency View Synchronizer for Minority Corruption (party p_i)

```
1: $curr := 1$
2: $next\_round := 1$
3: $\forall i \in N : attempted[i] := 1$
4: $finalized := true$
5:
6: // as a party:
7: on wish-to-advance():
8:     if $curr < next\_view$: then return
9:     $next\_view := curr + 1$
10:    send "wish, $next\_view$, 1" to relay($next\_view$, 1)
11:
12: upon receiving the first "$\sigma_{wish,e}$, $e$, $k$" with a valid $\sigma_{wish,e}$:
13:    if $e < next\_view$ then return
14:    if $e > next\_view$ then $next\_view := e$, send "wish, $e$, 1" to relay($e$, 1)
15:    send "$\sigma_{wish,e}$, $e$, $k$" to all parties in $\rho(i)$
16:    send "move, $e$, $k$" to relay($e$, $k$)
17:
18: upon receiving the first "$\sigma_{move,e}$, $e$, $k$" with a valid $\sigma_{move,e}$:
19:    if $e < curr$ then return
20:    if $e > curr$ then //enter view $e$
21:        $curr := e, finalized := false$
22:        send "move, $e$, 1" to relay($e$, 1)
23:        propose-view($e$)
24:    send "$\sigma_{move,e}$, $e$, $k$" to all parties in $\rho(i)$
25:    send "finalize, $e$, $k$" to relay($e$, $k$)
26:
27: upon receiving the first "$\sigma_{finalize,e}$, $e$, $k$" with a valid $\sigma_{finalize,e}$ from relay($e$, $k$):
28:    if $e = curr$ then $finalized := true$
29:
30: on sending wish or move and not receiving matching $\sigma_{wish,e}$ or $\sigma_{move,e}$ in $2\Delta$ time:
31:    if $attempted[next\_view] < t + 1$ then
32:        $attempted[next\_view] := attempted[next\_view] + 1$
33:        send "wish, $next\_view$, $attempted[next\_view]$" to relay($next\_view$, $attempted[next\_view]$)
34: on sending finalize and not receiving matching $\sigma_{finalize,e}$ within $2\Delta$ time:
35:    if $finalized = false$ and $attempted[curr] < t + 1$ then
36:        $attempted[curr] := attempted[curr] + 1$
37:        send "wish, $curr$, $attempted[curr]$" to relay($curr$, $attempted[curr]$)
38:
39: // as relay($e, k$):
40: upon receiving a set $M$ of valid threshold signature shares in "wish, $e$, $k$" messages from $t + 1$ unique parties:
41:    $\sigma_{wish,e}$ :=threshold-combine($M, e, k$)
42:    send "$\sigma_{wish,e}$, $e$, $k$" to all parties
43: upon receiving a set $M$ of valid threshold signature shares in "move, $e$, $k$" messages from $\frac{n}{2} + 1$ unique parties:
44:    $\sigma_{move,e}$ :=threshold-combine($M, e, k$)
45:    send "$\sigma_{move,e}$, $e$, $k$" to all parties
46: upon receiving a set $M$ of valid threshold signature shares in "finalize, $e$, $k$" messages from $\frac{n}{2} + 1$ unique parties:
47:    $\sigma_{finalize,e}$ :=threshold-combine($M, e, k$)
48:    send "$\sigma_{finalize,e}$, $e$, $k$" to all parties
```

by time $T_0 + \Delta$. Since every party sends **wish** to each relay in order, relay(e, 1) receives $t+1$ **wish** messages by $T + 2\Delta$ and sends $\sigma_{wish,e}$ to all parties. By $T + 3\Delta$, all honest parties receive this $\sigma_{wish,e}$ and send **move** to relay(e, 1), who then broadcasts the $\sigma_{move,e}$. Therefore, all honest parties enter view e by time $T + 5\Delta$.

For progress before GST and when the first relay for a view $e + 1$ is not honest, let T_0 be the time by at which at least $t+1$ honest parties call wish-to-advance() in view e and not in a view greater than e. Again, by validity, no honest party enters view $e + 2$ or greater. If an honest party receives $\sigma_{finalize,e+1}$ for view $e + 1$, then at least ϵn honest parties received $\sigma_{move,e+1}$ and sent it to their neighbors, so at least $n/2 + 1$ honest parties move to view $e+1$. If no party receives $\sigma_{finalize,e+1}$, the honest parties that called wish-to-advance() continue to send **wish** to the relays in order. Eventually, an honest relay receives a sufficient number of **wish** messages to create a $\sigma_{wish,e+1}$ and completes the stages of the protocol. In the case that relay$(e + 1, 1)$ is honest and $T_0 \geq$ GST, relay$(e+1, 1)$ sends $\sigma_{wish,e+1}$ to all honest parties by time $T_0 + \Delta$, who receive it by time $T_0 + 2\Delta$ and send **move** to relay$(e + 1, 1)$. Then, by time $T_0 + 4\Delta$ all honest parties receive $\sigma_{move,e+1}$ and move to view $e + 1$.

By defining the leader of a given view to be the first relay for that view as in the original protocol, using the same idea as in [38], one can prove that with a time out value of $\eta + \delta$, where $\eta = 5\Delta$, the new protocol also ensures an infinite number of synchronized views after GST (for an infinite run of the protocol). This modified protocol achieves expected linear communication complexity, as the communication through the expander graph only adds a constant number of additional constant-sized messages per message sent by a party in the original protocol. As in the original model, there are an expected constant number of Byzantine relays (≤ 2) until there is an honest relay for any given sequence of relays. The proofs therefore follow from that of the original protocol presented in [38].

Note that we did not address the assumption of a source of randomness. For our model, we can assume that all the parties agree on a common seed s prior to the beginning of the protocol. This is sufficient since we assume a static, oblivious adversary. For each view e, the randomness $rand_e$ is calculated using a hash function $rand_e = h(rand_{e-1}||e)$, and $rand_1$ is calculated as $rand_1 = h(s||1)$.