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

**Sravya Yandamuri**
Duke University, Durham, NC, USA

**Ittai Abraham**
VMware Research, Herzliya, Israel

**Kartik Nayak**
Duke University, Durham, NC, USA

**Michael K. Reiter**
Duke University, Durham, NC, USA

**Abstract**

Agreement protocols for partially synchronous 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. In particular, we show a version of HotStuff that retains linear communication complexity in each view, leveraging trusted hardware to tolerate a minority of corruptions. Our result uses expander graph techniques to achieve efficient communication in a manner that may be of independent interest.

**2012 ACM Subject Classification** Theory of computation → Communication complexity

**Keywords and phrases** communication complexity, consensus, trusted hardware

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2022.24

**Related Version** *Full Version*: `https://eprint.iacr.org/2021/184.pdf` [57]

**Funding** This research was supported in part by NIFA Award 2021-67021-34252 and by VMware and Novi research grants.

## 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 [21]. However, tolerating fewer 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 fewer 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) [22, 28, 4, 13, 20, 38, 31]. Protocols using synchrony increase the fault threshold by detecting equivocation (assuming signatures) and making deductions based on the absence of messages from other parties (e.g., [42, 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., [21, 17, 27, 53]) 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. Specifically, each node is equipped with hardware that implements the abstraction of an "append-only log," the contents to which it can attest using a *conventional* digital signature with a key that it holds. This capability is supported by numerous *existing*, trusted add-ons (TPMs [1], YubiKeys [49], smartcards, etc.) and is far simpler to implement than secure enclaves for arbitrary computation, as Intel SGX [19] attempts to do – but arguably fails [40, 52, 56, 55, 15].

The use of such small trusted hardware to boost fault tolerance was explored in A2M [17] and TrInc [32], which specifically improved PBFT [11]. However, this came at the expense of an $O(n^3)$ communication complexity per view for consensus among $n$ parties, measured as the (expected) number of words that all honest parties send. On the other hand, in the standard setting, we have recently seen considerable progress in improving communication complexity of consensus protocols. In particular, HotStuff [58] 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 fault tolerance can be boosted (but communication costs retained) in these protocols using small trusted hardware. In this work, we answer these questions affirmatively for a corruption threshold $t \leq (\frac{1}{2} - \epsilon)n$ for an arbitrarily small $\epsilon$, which we term a *minority corruption* adversary. In the following, we describe our results and the key techniques used to achieve these results in the partially synchronous and asynchronous settings. We include $(\frac{n}{2}+1)$-Provable-Broadcast, a broadcast primitive with linear communication complexity that can be used to improve the fault tolerance of such protocols, as well as a version of HotStuff that uses this primitive to withstand minority corruption. In the full version of our work [57], we present an expected linear communication complexity and constant round view synchronization protocol that is a strict generalization of that in [39] and withstands minority Byzantine corruption.

## 1.1 HotStuff-M: HotStuff with Minority Corruption

Our 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). *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 having a small trusted hardware, such that $t \leq (\frac{1}{2} - \epsilon)n$ of the 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 $\epsilon n$-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 $\epsilon n$ 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 [30, 37, 16], 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.

**Future work and open questions.** Our protocol tolerates $t \le (\frac{1}{2}-\epsilon)n$ faults for an arbitrarily small $\epsilon$, but addressing the communication complexity with trusted hardware and *optimal resilience* is still an open question. Also, our solution centers on a novel use of expander graphs. While the solution obtains optimal asymptotic communication complexity, the constants incurred due to the use of expanders may not be realistic for practical values of $n$. Solving this problem for smaller values of $n$ is an interesting open question.

## 2 Model and Preliminaries

We consider $n$ parties (a.k.a. processors) $p_1, \ldots, p_n$ connected by a reliable, authenticated, fully connected network, where up to $t \le (1/2-\epsilon)n$ 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. We consider 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 2** (Validated Byzantine Agreement). *A validated Byzantine agreement protocol among $n$ parties tolerating a maximum of $t$ faults 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 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. [10], 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 2.1). 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 2.2. We measure communication complexity as the number of *words* that all honest parties send and receive; each word is $O(\kappa)$ bits long where $\kappa$ is a security parameter. We also assume all the messages sent by parties are signed using a threshold signature scheme (interface described in Section 2.3). We assume the use of PKI to validate signatures.

### 2.1 Small Trusted Hardware

In this section, we introduce the abstraction of a *small trusted hardware* enforcing non-equivocation with $O(1)$ storage. Such hardware units have been considered in prior works such as A2M [17], TrInc [32], 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), YubiKey [49, 1], and hardware security modules (HSMs) [46]. Without loss of generality, we assume the existence of a functionality similar to that of A2M [17]; our solutions apply to all of these small trusted hardware units.

**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 Table 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. We refer to the trusted hardware of party $p_i$ as $HW_i$.

**Table 1 Interfaces of the trusted hardware component.** $(K_{pub}, K_{priv})$ is public-private key pair associated with the hardware device, $C$ is a monotonic counter representing the number of logs maintained by the hardware, and $c_{id}$ is a monotonic counter representing the length of log indexed by *id*.

| | |
|---|---|
| CreateLog() | Increment $C$<br>Initialize empty log with $id := C$, $c_{id} := 0$<br>**return** $id$ |
| Append<br>$(id, c_{new}, x)$ | **if** $id \leq C$:<br>    **if** $c_{new} = \perp$:<br>        Increment $c_{id}$, $log[id][c_{id}] := x$<br>    **if** $c_{new} > c_{id}$:<br>        $c_{id} := c_{new}$, $log[id][c_{id}] := x$<br>    **return** Lookup$(id, c_{id})$ |
| Lookup$(id, s)$ | **if** $id \leq C$ and $s \leq c_{id}$:<br>    **return** $\langle \text{LOOKUP}, id, s, log[id][s] \rangle_{K_{priv}}$ |
| End$(id, z)$ | **if** $id \leq C$:<br>    **return** $\langle \text{END}, id, c_{id}, log[id][c_{id}], z \rangle_{K_{priv}}$ |
| Counters$(z)$ | **return** $\langle \text{HEAD}, \bigcup_{id<C} \{(id, c_{id})\}, z \rangle_{K_{priv}}$ |

The hardware provides four interfaces. Append$(id, c_{new}, x)$ appends the value $x$ to the log identified by *id*. If $c_{new} = \perp$, the function increments the counter of the log and inserts $x$ into the current position of the log. Otherwise, it appends to position $c_{new}$ if $c_{new}$ is strictly higher than the current log position. Lookup$(id, s)$ and End$(id, z)$ return an attestation of the log with identifier *id* for the value stored at position $s$ and the last position, respectively. Finally, Counters$(z)$ returns the attested current counter values of all logs. The nonce $z$ is used to ensure freshness of an attestation; we omit mentioning the nonce when it is not used. To simplify the description, we imagine that the hardware stores entire logs. In reality, the hardware need only store the end state of a log; a party can always store the attestations for different positions separately.

If party $p_i$ calls Append$(q, \perp, x)$, then it receives $\langle \text{LOOKUP}, q, s, x \rangle_{K_{priv}}$ in response, for the log position $s$ at which $x$ was appended. $p_i$ can forward this response – or another copy, obtained by invoking LOOKUP$(q, s)$ – to party $p_j$ to prove that $p_i$ added value $x$ to its log $q$

at position $s$. Since the hardware allows only appending to the log, $p_j$ can be assured that $p_i$ can attest to no other value at position $s$ of log $q$. The use of the $\textsc{End}(q, z)$ 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.

## 2.2 Expander Graphs

Expander graphs are sparse graphs with a high degree of connectivity between groups of nodes. We refer to 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 3.** *An $(n, \alpha, \beta)$-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 $\alpha n$ vertices has at least $\beta n$ unique neighbors.*

▶ **Lemma 4.** *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 $\epsilon n$ 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 $|U| > \frac{n}{2}$.*

## 2.3 Cryptographic Abstractions

**Threshold signatures.** In addition to (and separate from) the signatures generated by hardware modules, we make use of a $k$ out of $l$ threshold signature scheme [47] 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$(m, s_j, pk_j) =$ true$, \forall s_j \in S$.
- threshold-verify$(m, \sigma)$: returns true if $\sigma$ was a result of computing threshold-combine$(m, S)$ where $|S| \geq k$ and share-validate$(m, s_j, pk_j) =$ 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 2.1. 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 $\sigma_{\text{in}}$ 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 $\sigma_{\text{out}}$ 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$(id, (m, \sigma_m))$ = true, where $\sigma_m$ is the proof of validity for $m$.
- **Provability.** If a sender can produce two valid proofs $\sigma_{\text{out}}$ and $\sigma'_{\text{out}}$ s.t. 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 sender is honest, no honest party invokes abandon$(id)$ (meaning they immediately terminate their execution of this instance of $(\frac{n}{2}+1)$-Provable-Broadcast), messages among honest parties arrive, 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 $\sigma_{\text{out}}$.

**Algorithm 1** $(\frac{n}{2}+1)$-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 $\sigma_{\text{out}}$
```

The requirements described above have minor differences from those in 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 $\sigma_{\text{out}}$ 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 2.1. 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 4; 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 $\sigma_{\text{out}}$ consisting of all the attestations it received. However, the proof $\sigma_{\text{out}}$ 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 $\sigma_{\text{out}}$ remains $O(1)$ without relying on the hardware to generate threshold signatures.

**Algorithm 2** $(\frac{n}{2}+1)$-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:              $(\text{att}_{logId}, \text{att}_{counters})$ := createAttestations($id$, $(v, \sigma_{\text{in}})$)
7:              send "$id$, $send$, $((\text{att}_{logId}, \text{att}_{counters}), \sigma_i)$" to parties in $\rho(i)$
8:              wait for $id$, $send$, $((\text{att}_{logId,j}, \text{att}_{counters,j}), \sigma_j)$
                from $(\frac{1}{2}+\frac{\epsilon}{2})d$ parties $p_j$ in $\rho(i)$
                s.t. validateNeighbor($id, p_j, (v, \sigma_{in})$,
                $(\text{att}_{logId,j}, \text{att}_{counters,j}), \sigma_j$) = 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:     upon abandon($id$) do
13:         $stop$ := true
14: procedure createAttestations($id$, $(v, \sigma_{\text{in}})$)
15:     $logId$ := log($id$), $seqNo$ := seq($id$)                    ▷ parse $id$
16:     $\text{att}_{logId}$ := APPEND($logId, seqNo, (v, \sigma_{\text{in}})$)
17:     deliver $((v, \sigma_{\text{in}}), (\text{att}_{logId}))$
18:     return $(\text{att}_{logId}, \text{COUNTERS}())$
```

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 $\epsilon n$, 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 1), out of which at least $\epsilon n$ 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 $\sigma_{\text{out}}$ 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.

Algorithms 1 and 2 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 $(v, \sigma_{\text{in}})$; $v$ is the value to be sent and $\sigma_{\text{in}}$ 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 1 line 3).

On receiving the message from the sender, $p_i$ invokes validate$(id, (v, \sigma_{\text{in}}))$ (Algorithm 2 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, $\sigma_i$, 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 2 lines 6, 14-18). 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 $(v, \sigma_{\text{in}})$ 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 $(v, \sigma_{\text{in}})$. On receiving messages from a majority of its neighbors (specifically $(\frac{1}{2}+\frac{\epsilon}{2})d$ neighbors) that satisfy validateNeighbor() (Algorithm 2 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 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$_{counters}$ attestation are used.

On collecting threshold-sign$_i((v, \sigma_{in}))$ from a majority of replicas, the sender combines the signature shares to generate $\sigma_{\text{out}}$ and delivers $\sigma_{\text{out}}$ (Algorithm 1 lines 8-10).

Here are the key ideas that ensure provability and termination. We present detailed proofs in Appendix B.

1. **Any set of $\epsilon n$ 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 4 and it guarantees provability since if a sender receives $vote$ messages from a majority of parties, at least $\epsilon n$ of them are honest and they will ensure that at least $\frac{n}{2}+1$ parties have attested to $(v, \sigma_{\text{in}})$ 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 $\epsilon n$ 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 4 and it guarantees termination since if an honest sender sends a valid $(v, \sigma_{\text{in}})$ to all $(1/2+\epsilon)n$ honest parties, then at least $\frac{n}{2}+1$ honest parties will send $vote$ messages, sufficient to generate $\sigma_{\text{out}}$.

## 4 HotStuff-M: HotStuff with Minority Corruption

In this section, we present HotStuff-M, a version of the HotStuff [58] protocol that tolerates minority Byzantine corruption under partial synchrony assuming a minimal trusted hardware at each party. Similar to HotStuff, the protocol has 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 [58] tolerating $n = 3t + 1$. The protocol proceeds in a sequence of consecutive views where each view has a unique leader. A view consists of four phases: PROMOTE, KEY, LOCK and COMMIT, as first formalized in [5]. We use $\sigma_{phase}$ to denote the threshold signature collected by the leader of a given view during the *phase* phase of that view. Each view of HotStuff progresses as follows:

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

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 $\sigma_{key}$ to the leader of the next view. The next view's leader collects $2t + 1$ such values, picking the highest $\sigma_{key}$ as $\sigma_{highKey}$. The safety and liveness of HotStuff follow from the following:

**Uniqueness within a view.** Since parties only vote once in each phase, a $\sigma_{commit}$ 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 commits 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 $\sigma_{key}$ 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.

**Table 2 Validation functions passed to provable broadcast in different phases of view $e$.** We assume end attestations $\text{att}_{lock}$ (containing $\sigma_{lock}$) is invoked during validate() call as needed. Also note that $(\text{att}_{logId,i}, \text{att}_{counters,i})$ are sent by every party to their neighbor as a part of provable broadcast and generated in the invocation of createAttestations$(id, (v, \sigma_{\text{in}}))$.

| Phase | validate$(id, (v, \sigma_{\text{in}}))$ | validateNeighbor$(id, (v, \sigma_{\text{in}}), (\text{att}_{logId,j}, \text{att}_{counters,j}), \sigma_j)$ |
|---|---|---|
| PROMOTE | **cond:** ext-valid$(v)$ = true, $\sigma_{\text{in}}.val = \sigma_{lock}.val$ or view$(\sigma_{\text{in}})$ > view$(\sigma_{lock})$ **proof:** $\sigma_i := \text{att}_{lock}$ | $(\sigma_{\text{in}}.val = \sigma_{lock}.val$ or view$(\sigma_{\text{in}})$ > view$(\sigma_{lock}))$, and LOCK log in $\text{att}_{counters,j}$ is at view$(\sigma_{lock})$, and PROGRESS log in $\text{att}_{counters,j}$ 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$(\sigma_{\text{in}})$ = PROMOTE **proof:** $\sigma_i := \perp$ | PROGRESS log in $\text{att}_{counters,j}$ is at $e-1$, and KEY log in $\text{att}_{counters,j}$ 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$(\sigma_{\text{in}})$ = KEY **proof:** $\sigma_i := \perp$ | PROGRESS log in $\text{att}_{counters,j}$ is at $e-1$, and LOCK log in $\text{att}_{counters,j}$ 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 $e$ |

## 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 $\sigma_{key}, \sigma_{lock}, and \sigma_{commit}$ by partitioning the honest parties. Similarly, across views, Byzantine parties could send an incorrect (stale) $\sigma_{key}$ 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., absence of conflicting information and absence of a designated message indicating that the party quit a given view 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 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

**Algorithm 3** HotStuff-M: HotStuff with Minority Corruption (for party $p_i$).

```
1: for $e := 1, 2, 3, \dots$ do
2:     as a leader                                                      ▷ 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 logs are valid, sequence numbers in $\mathrm{att}_{progress}$ and $\mathrm{att}_{highQC}$ match
           those in the COUNTERS attestation for the respective logs, and counter value of
           PROGRESS log in the COUNTERS attestation is $e-1$
4:         For each $m \in M$, let $\sigma^{m}_{highKey}$ denote the highest key QC from party $p_m$
5:         $\sigma_{highKey} := (\arg\max_{m \in M}\{\mathrm{view}(\sigma^{m}_{highKey})\})$   ▷ $\mathrm{view}(\sigma^{m}_{highKey})$ is the view in which
           $\sigma^{m}_{highKey}$ was formed
6:         if $\sigma_{highKey} = \perp$ then proposal := client's command else proposal := $\sigma_{highKey}.val$
7:     as a party                                                       ▷ NEW-VIEW phase
8:         go to this line if no progress happens during the "wait" step in any phase
9:         $\mathrm{att}_{progress} := \langle \text{LOOKUP}, \text{PROGRESS}, e, e \rangle := \text{APPEND}(\text{PROGRESS}, e, e)$
10:        $\mathrm{att}_{highQC} := \langle \text{END}, \text{KEY}, seqNoHighQC, highKeyQC \rangle := \text{END}(\text{KEY})$
11:        send "$((e, \text{NEW-VIEW}), send, (\mathrm{att}_{progress}, \mathrm{att}_{highQC}, \text{COUNTERS}()))$" to view $e+1$
   leader
12:    as a leader
13:        $\sigma_{key} := (\frac{n}{2}+1)$-PB-Initiate($(e, \text{PROMOTE}), (\text{proposal}, \sigma_{highKey})$)    ▷ PROMOTE
14:        $\sigma_{lock} := (\frac{n}{2}+1)$-PB-Initiate($(e, \text{KEY}), (\text{proposal}, \sigma_{key})$)            ▷ KEY phase
15:        $\sigma_{commit} := (\frac{n}{2}+1)$-PB-Initiate($(e, \text{LOCK}), (\text{proposal}, \sigma_{lock})$)        ▷ LOCK phase
16:        send "$((e, \text{COMMIT}), send, \sigma_{commit})$"                                   ▷ COMMIT phase
17:    as a party
18:        $(\frac{n}{2}+1)$-PB-Respond($(e, \text{PROMOTE})$, validate(), validateNeighbor())   ▷ PROMOTE
19:        $(\frac{n}{2}+1)$-PB-Respond($(e, \text{KEY})$, validate(), validateNeighbor())          ▷ KEY phase
20:        $(\frac{n}{2}+1)$-PB-Respond($(e, \text{LOCK})$, validate(), validateNeighbor())        ▷ LOCK phase
21:        wait for "$((e, \text{COMMIT}), send, \sigma_{commit})$" from view $e$ leader       ▷ COMMIT phase
22:        if $\sigma_{commit}$ is a signature from view $e$ from COMMIT phase then commit $\sigma_{commit}.val$
```

$\sigma_{lock}$ 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. At the end of this section, we discuss the intuition for how the COUNTERS attestation can be replaced with the use of nonces.

We present our protocol in Algorithm 3 and Table 2. 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 9) to obtain attestation $\mathrm{att}_{progress}$. In addition, it obtains an end attestation $\mathrm{att}_{highQC}$ for its *keyQC* (line 10). The party sends this information together with the COUNTERS() attestation to the leader in a NEW-VIEW message (line 11).

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 $\mathrm{att}_{highQC}$ and $\mathrm{att}_{progress}$ matches the ones in the COUNTERS attestation (line 3). Thus, even if the sending party is Byzantine, the $\text{att}_{highQC}$ 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 13-15 and 18-20). As described in the previous section, if the leader successfully receives a $\sigma_{key}$ (respectively $\sigma_{lock}$ and $\sigma_{commit}$), 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 2). 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, $\sigma_{lock}$. 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 $\text{att}_{counters,j}$ and comparing them to the sequence numbers in the other attestations). In the KEY, LOCK, and COMMIT phases, the parties check if $\sigma_{key}$, $\sigma_{lock}$, and $\sigma_{commit}$, 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 $\sigma_{commit}$ as proof of commit. Each of the parties can then commit $\sigma_{commit}.val$.

Due to space constraints, we present formal proofs in Appendix C and a view synchronization protocol in the full version of our paper [57]; the protocol is a generalization of the expected linear communication complexity protocol of [39] withstanding minority corruption.

**Communication complexity.** From Theorem 14, 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.

**Replacing the Counters attestation.** We now present intuition for how nonces can be used to replace the COUNTERS attestation. We use the COUNTERS attestation to:

1. Prevent parties from presenting the stale state of a log to other parties
2. Force parties to quit a view before sending messages in a subsequent view

We address concern 1 using the COUNTERS attestation by requiring a party to append a value to position $e$ in their PROGRESS log before presenting the state of their log in view $e+1$. The COUNTERS attestation shows that a party did in fact append the value to their PROGRESS log in position $e$ prior to presenting the state of their logs in view $e+1$. The following handshake between a proving party and a verifying party can replace the use of the COUNTERS attestation for this scenario:

1. To send a message proving the state of their logs in view $e + 1$, a party sends an End attestation from their progress log to the verifying party, proving that they have quit view $e$
2. Upon receiving a valid attestation showing that the proving party quit view $e$, the verifying party sends the proving party a random nonce
3. The proving party uses the nonce for the End attestations it gathers to send the current state of its logs

We address concern 2 using the Counters attestation by requiring a party to send a Counters attestation showing that they have appended a value to their progress log in position $e$ with any view $e + 1$ messages that they send. A similar handshake to that described above can be used to replace the Counters attestation for this scenario.

## 5 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 [19], Flicker [36], Aegis [48], XOM [33], and Bastion [12]). They have been used to provide confidentiality [19, 48, 33, 12, 45] as well as to improve resilience and performance in the context of consensus protocols [7, 25, 45, 9, 2, 34]. 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., [50]). Consequently, these platforms can present a large attack surface, giving way to attacks from outside the TEE (e.g., [14]).

Our work focuses on using small trusted hardware with a fixed, limited functionality (e.g., YubiKeys [49]). There have been several works using such a hardware to improve the performance of BFT protocols [27, 53, 54, 17, 32, 41, 3]. Notable works include A2M [17] and TrInc [32]. Chun et al. [17] 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 [32], 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 [30, 29, 37]. Chlebus et al. [16] 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. The protocol for leader election by King et al. withstands a one-third Byzantine adversary in synchrony using expanders to achieve polylogarithmic per-process communication complexity [30]. They extend this work to obtain $o(n^2)$ total bits of communication against an adaptive adversary. Recently, Momose and Ren [37] 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 [23, 43, 24]. The way we use expander graphs in our work differs from their use in previous works, as we separate the messages that must go through the hardware and those that do not. The separation enables better communication complexity for messages sent from the hardware through the use of expander graphs and better communication complexity for other messages through the use of cryptography.

**Non-equivocation.** The past two decades have seen various works on non-equivocation [17, 18, 44, 6, 35]. Clement et al. [18] define equivocation and show that non-equivocation alone is not sufficient to increase the threshold of Byzantine parties in a network when trying to reach agreement; doing so requires transferable authentication. Ruffing et al. [44] 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. [8] and [18] both explore the gap between omission failures and the non-equivocation property achieved by the use of trusted hardware. 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. URL: `https://trustedcomputinggroup.org/`.

2 Hyperledger sawtooth, 2019. URL: `https://sawtooth.hyperledger.org/`.

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 Naama Ben-David, Benjamin Y Chan, and Elaine Shi. Revisiting the power of non-equivocation in distributed protocols. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 450–459, 2022.

9 Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint*, 2018. `arXiv:1805.08541`.

10 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.

11 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

12 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.

13 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.

14 Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *18$^{th}$ International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2013.

15 Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy*, June 2019.

16 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.

17 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.

18 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.

19 Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

20 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

21 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

22 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

23 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.

24 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi, and Yann Vonlanthen. Scalable byzantine reliable broadcast (extended version). *arXiv preprint*, 2019. arXiv:1908.01738.

25 Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016, 2016.

26 Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

27 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.

28 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.

29 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.

30 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.

31 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. ACM, 2019.

32 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.

33 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.

34 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.

35 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.

36 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.

37 Atsuki Momose and Ling Ren. Optimal communication complexity of byzantine consensus under honest majority. *arXiv preprint*, 2020. `arXiv:2007.13175`.

38 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

39 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint*, 2020. `arXiv:2002.07539`.

40 Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint*, 2020. `arXiv:2006.13598`.

41 Vincent Rahli, Francisco Rocha, Marcus Völp, and Paulo Esteves-Verissimo. Deconstructing minbft for security and verifiability.

42 Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. Practical synchronous byzantine consensus. *arXiv preprint*, 2017. `arXiv:1704.02397`.

43 Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. *Available [online].[Accessed: 4-12-2018]*, 2018.

44 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.

45 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.

46 Jinho Seol, Seongwook Jin, Daewoo Lee, Jaehyuk Huh, and Seungryoul Maeng. A trusted iaas environment with hardware security module. *IEEE Transactions on Services Computing*, 9(3):343–356, 2015.

47 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

48 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.

49 Suresh Thiru, Shamalee Deshpande, and Stina Ehrensvard. Yubikey strong two factor authentication, January 2021. URL: `https://www.yubico.com/`.

50 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.

51 Salil P. Vadhan. *Pseudorandomness*, volume 7 of *Foundations and Trends in Theoretical Computer Science*. Now Publishers, Inc., 2012.

52 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 [56].

53 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.

54 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.

55 Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM Conference on Computer and Communications Security*, October 2017.

56 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 [52].

57 Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. Communication-efficient bft protocols using small trusted hardware to tolerate minority corruption. *Cryptology ePrint Archive*, 2021.

58 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint*, 2018. `arXiv:1803.05069`.

## A Proofs for Expander Graph Lemmas

▶ **Lemma 5.** *For every constant $0 < \alpha < \beta < 1$ and sufficiently large $n$, there exists a $d$-regular graph that is an $(n, \alpha, \beta)$-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 $k$th 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 $\alpha n$ nodes, $S$, and a set of $\beta n$ nodes, $T$. Now, consider the matching of the first set of $\alpha n$ 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 $i$th 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 $\alpha n$ nodes $S$, and any set of $\beta n$ 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 $\alpha n$ vertices does not expand to more than $\beta n$ 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 $(n, \alpha, \beta)$-expander. Thus, $G_{n,\alpha,\beta}$ exists. ◀

▶ **Lemma 6.** *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 5, 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 5 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 $\epsilon n$ 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 5, we first bound the probability that for a given set of nodes $R$ of size $\epsilon n$ 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 7.** *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 5, 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 5 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 $\epsilon n$. Using the same technique as that in the proof for Lemma 5, 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 $\epsilon n$ 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 8.** *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 6, as any graph that violates this property also violates the property in Lemma 6. ◀

The following lemma and theorem can be found in any resources on expander graphs, such as Hoory et al. [26].

▶ **Lemma 9** (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 10** (Theorem 4.12 of Vadhan [51], restated)**.** *For any constant $d \in \mathbb{N}$, 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 $\infty$ (and $d$ is held constant).*

▶ **Lemma 11.** *For all sufficiently large integers $n$ and positive constants $\epsilon$ and $\beta$ such that $0 < \epsilon < \beta < 1$ there exists an $d$-regular expander $G_{n,\epsilon,\beta}$ such that for any set $S$ of $\epsilon n$ 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 5, we know that a random $d$-regular expander $G_{n,\epsilon,\beta}$ exists for a sufficiently large constant $d$. By the Expander Mixing Lemma [26], 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 [51, 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 4

**Proof.** By Lemma 11, 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 $S$, where $S$ is any set of $\epsilon n$ nodes in the graph. Further, by Lemmas 6, 7, 8, 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. To show that property 4 holds, we will assume that it doesn't hold and then arrive at a contradiction. Consider an arbitrary set $S$ of $\epsilon n$ nodes in the graph. Assuming that property 4 does not hold, 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 $\epsilon n$ 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 $Y$. In order for the set $U'$ to exist as defined, within $Y$ there must be a set $T$ of $(\frac{1}{2} + \frac{\epsilon}{c})n$ nodes containing $U'$ such that there are more than $(\frac{1}{2} + \frac{\epsilon}{2})\epsilon dn$ edges with one vertex in $S$ and one in $T$, where an edge exists between two nodes if they are neighbors in $G$. We have arrived at a contradiction, as this violates that the graph satisfies the property in Lemma 11. Therefore the lemma holds. ◀

## B Proofs for Provable Broadcast

▶ **Lemma 12** (Provability). *In $(\frac{n}{2}+1)$-PB-Initiate, if the sender delivers two valid proofs $\sigma_{out}$ and $\sigma'_{out}$ corresponding to values $(v, \sigma_{in})$ and $(v', \sigma'_{in})$ 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 $\sigma_{\text{out}}$ contains a threshold signature for $(v, \sigma_{\text{in}})$ 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*, $\xi_i$" 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 $\sigma_j$ satisfy the criteria in the validateNeighbor() function. Thus, $(\text{att}_{logId}, \text{att}_{head,j})$ were created by running createAttestations(*id*, $(v, \sigma_{\text{in}})$) and $\sigma_j$ proves that $(v, \sigma_{\text{in}})$ is valid for $p_j$'s state based on the conditions in validate(*id*, $(v, \sigma_{\text{in}})$). By Lemma 4, any $\epsilon n$ 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. ◀

▶ **Lemma 13** (Termination). *If the sender is honest, no honest party invokes abandon(id), all messages among honest parties arrive, and validate(id,* $(v, \sigma_{in})$*) = true for all honest parties, then (i) eventually all honest parties deliver v, and (ii) the sender delivers v with a valid proof* $\sigma_{out}$.

**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 6, 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 $(v, \sigma_{\text{out}})$. ◀

▶ **Theorem 14.** *The* $(\frac{n}{2}+1)$*-Provable Broadcast algorithm in Algorithms 1 and 2 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 12 and 13. ◀

## C Proofs for HotStuff-M

▶ **Lemma 15.** *At the end of a view e, (i) if a party receives a* $\sigma_{commit}$ *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* $\sigma_{lock}$ *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 12 and the criteria for validateNeighbor() in Table 2. ◀

▶ **Lemma 16.** *Suppose the earliest view in which a value $v$ is committed by an honest party is $e$. For all views $> e$, a valid $\sigma_{key}$ 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 $\sigma_{key}$ for $v' \neq v$ exists in view $e' > e$. Let $e^*$ be the earliest view in which a $\sigma_{key}$ for a value $v^*$ is formed such that $v^* \neq v$ and $e^* > e$. It follows that $e^* \leq e'$. Since there is a $\sigma_{key}$ for $v^*$ in $e^*$, by Lemma 12, a set $Q$ of at least $\frac{n}{2}+1$ parties have sent messages with attestations $(\text{att}_j, \text{att}_{counters,j}), \sigma_j$ 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. $p$'s log and its attestation satisfying validateNeighbor(). Since view $e^*$ is the first view where a higher $\sigma_{key}$ was formed for a different value, the end state of LOCK in view $e^*$ must be for value $v$. Thus, the predicate view$(\sigma_{\text{in}}) >$ 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 $\text{att}_{counters,j}$ 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, $\sigma_{key}$ in view $e^*$ cannot be formed for value $v'$, a contradiction. ◀

▶ **Theorem 17** (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 that of $v'$ s.t. $v \neq v'$. By Lemma 12, $e \neq e'$. Suppose, without loss of generality, $e > e'$. By Lemma 16, no $\sigma_{key}$ can be formed in view $> e$ for value $\neq v'$. Honest parties only vote for $\sigma_{lock}$ in the LOCK phase in view $e'$ if it was generated in view $e'$; thus, a $\sigma_{commit}$ cannot be formed in view $e'$. ◀

▶ **Theorem 18** (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 $\sigma^*_{lock}$ is the $\sigma_{lock}$ stored in a party's LOCK log from the highest view $e^*$ for a value $v$. By Lemma 15, at least $\frac{n}{2}+1$ parties must have the $\sigma^*_{key}$ for value $v$ stored in their LOCK logs at position $e^*$ prior to creating the COUNTERS() attestation to report their highest $\sigma_{lock}$ in $e$. Since the leader waits for $\frac{n}{2}+1$ valid $\sigma_{key}$ messages during a view change, it will obtain $\sigma_{key}$ from a view $\geq e^*$. Let $\sigma'_{key}$ be the highest valid $\sigma_{key}$ 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 12). Since $e$ is after GST, all messages will arrive within the bounded delay $\Delta$, and thus for each of the three phases, the termination property for provable broadcast from Lemma 13 should be satisfied. Thus, all honest parties will commit. ◀