# Consensus in One Shot: Fast Agreement Beyond Classical Limits

Omri Shmueli[*‡] Ilan Tennenhouse[†‡]

**Abstract**

Good-case latency, studied in early fast-consensus protocols (Martin and Alvisi, DSN 2005) and formalized by Abraham, Nayak, Ren, and Xiang (DISC 2020), captures a natural efficiency goal: when the sender is honest, agreement should be reached quickly for the honest parties. In the standard classical model, however, there exists a barrier for any $f \geq n/3$ corrupted parties. Specifically, synchronous broadcast requires good-case latency at least $\Delta + \delta$ for $f \geq n/3$, where $\delta$ is the actual, unknown message-delay bound and $\Delta \gg \delta$ is its known, conservative upper bound. Under partial synchrony, consistency and liveness cannot even both hold for $f \geq n/3$ (Dwork, Lynch, and Stockmeyer, JACM 1988).

In this work we study the use of quantum information to get past these classical barriers, specifically using one-shot signatures (OSS) (Amos et al., STOC 2020, Shmueli and Zhandry, CRYPTO 2025). OSS prevents even a corrupt signer from issuing signatures on different messages under the same verification key, which provides cryptographic non-equivocation without trusted hardware assumptions.

Assuming OSS and a public-key infrastructure, we obtain two main results against quantum polynomial-time static corruptions. First, under synchrony, we construct a state-machine replication (SMR) protocol without clients, with optimal $\delta$ good-case latency, tolerating any $f < n$ corruptions among $n$ parties. Second, under partial synchrony, we obtain SMR with $2 \cdot \delta$ good-case latency which is consistent under any number of corruptions and live assuming $f < \frac{n}{2}$. Specifically, a corrupt majority may cause progress to stop, but cannot create conflicting histories. Such an always consistent protocol is impossible classically (without assumptions like trusted hardware) even if liveness is only required to hold against a single corrupt party. In both our SMR protocols, communication is entirely classical, with only local quantum computation.

[*]NTT Research. This work was done in part while the author was a visiting scientist at the Simons Institute for the Theory of Computing at UC Berkeley.

[†]Tel Aviv University.

[‡]The authors contributed equally to this work.

# Contents


**1 Introduction** **3**
- 1.1 Main Results . . . 4
- 1.2 Related Work . . . 6

**2 Technical Overview** **7**
- 2.1 Synchronous Maximally Resilient Fast Byzantine Broadcast . . . 8
- 2.2 Synchronous Maximally Resilient Fast SMR without Clients . . . 10
- 2.3 Partially Synchronous Fast and Always-Consistent SMR . . . 12
- 2.4 On Proof of Stake . . . 15

**3 Model and Preliminaries** **16**
- 3.1 One-Shot Signatures . . . 17
- 3.2 Problem Definitions. . . . 19
- 3.3 Additional Latency Metrics. . . . 20

**4 Maximally Resilient BB with Hyper Fast Good-Case Latency** **21**
- 4.1 Protocol . . . 21
- 4.2 Proofs. . . . 22

**5 Maximally Resilient SMR without Clients with Hyper-Fast Good-Case Latency** **23**
- 5.1 Chainable Broadcast . . . 24
- 5.2 SMR Protocol . . . 25

**6 Always Consistent Partially Synchronous SMR with Improved Resilience** **27**
- 6.1 Protocol. . . . 27
- 6.2 Proofs. . . . 29

# 1 Introduction

Decentralized systems must reconcile two basic demands: efficiency and security. They should operate quickly when participants act honestly, yet remain secure if participants deviate. In consensus protocols, security requires all honest parties to reach valid, consistent decisions and make progress despite Byzantine faults, i.e. parties which may deviate arbitrarily from the protocol, and potentially coordinate their deviations. An $n$-party protocol is resilient against $f := f(n)$ corruptions if its guarantees hold despite any coalition of up to $f$ Byzantine parties, under the stated network assumptions.

A central efficiency goal for consensus is low *latency*, namely, the time parties wait for a decision. Even a secure protocol may be unsuitable for time-sensitive applications if it imposes long confirmation delays during ordinary operation. This motivates the study of *good-case latency*: how quickly honest parties reach a decision when the designated sender (or current leader in SMR) is honest, even while other parties may behave maliciously [4].

**The classical $\frac{n}{3}$ corruptions barrier.** Consensus protocols are commonly studied under three timing models. Under *synchrony*, a known upper bound on message delays is assumed to hold throughout the execution. Under *partial synchrony*, this message-delay bound is assumed to hold only after some finite, but unknown *global stabilization time GST* [15]. Thus, a partially synchronous protocol must preserve consistency even before the network stabilizes, but may rely on eventual stabilization to guarantee progress. Under *asynchrony*, no fixed upper bound is imposed.

In all three mainstream timing models of consensus, classical protocols face a barrier of $\frac{n}{3}$ corruptions for a number of objectives. Under *synchrony*, while authenticated Byzantine broadcast can tolerate every $f < n$ [14], this resilience comes with a latency cost. Let $\delta$ be the actual message-delay bound and $\Delta$ a known upper bound used by the protocol, which is normally set significantly larger $\delta \ll \Delta$ for safety reasons [4]. Classical synchronous broadcast requires good-case latency $\geq \Delta + \delta$ for any $f \geq \frac{n}{3}$, and for $f \geq \frac{n}{2}$ this grows to $\Omega\left(\frac{n}{n-f} \cdot \Delta\right)$ [4]. Under *partial synchrony* (and therefore also under asynchrony), the limitation becomes one of feasibility rather than just latency: guaranteeing both consistency and liveness for nontrivial consensus against $f$ Byzantine faults is impossible when $f \geq \frac{n}{3}$ [15, 26]. Thus, tolerating a large adversarial coalition can force substantial waiting even under synchrony, despite an honest sender and fast message delivery. Under partial synchrony, it can preclude simultaneous consistency and liveness altogether.

Unfortunately, large-scale Byzantine faults can be difficult to anticipate, do not necessarily involve deliberate collusion among operators, and can cause lasting damage even when short-lived. For example, hidden concentrated control can secretly place many nodes under a small number of decision-makers [18]. Another example can happen without the operating nodes even knowing, when a shared implementation vulnerability compromises many simultaneously [18]. Economic incentives can also play a role in the formation of adversarial coalitions. In *time-bandit attacks*, newly valuable trading opportunities can justify the cost of rewriting recent blockchain history, in order to insert the attacker's transactions [13]. Moreover, bribery models show that miners whose participation is not pivotal to an attack (i.e., given that other miners take the bribe, the attack will go through, with or without that individual miner) can prefer accepting payment from the attacker. Had all miners refused the bribe collectively, there would have been no attack, but individually, without knowledge of other miners' choices (and the fact that accepting the bribe can stay private), refusing the bribe forfeits the payment without necessarily preventing the attack [22]. These are just a few examples which motivate malicious-majority resilience.

**Quantum cryptography for decentralized systems.** The above describes a fundamental limitation for consensus in the classical world. On the one hand, an event involving more than $\frac{n}{3}$ corruptions is not entirely unrealistic. On the other hand, once we seek security against $f \geq \frac{n}{3}$ faults, under synchrony, our protocols become noticeably slower, and under partial synchrony (and asynchrony) they simply cannot work at all[1]. The catch to these barriers is that they do not apply to the *quantum* setting, that is, where quantum computers are given not only to the adversary, but also to honest parties. Indeed, during recent years quantum cryptography has seen success in circumventing classical impossibilities in decentralized systems [29, 12, 5]. Given these understandings, the question we ask in this work is natural:

*Can the use of quantum cryptography in classical consensus simultaneously enable both fast agreement and resiliency against a malicious majority?*

**One-shot signatures and non-equivocation.** We investigate this question using *one-shot signatures* (OSS), which were defined in [5] and constructed in [24]. An OSS key pair consists of a classical public verification key $\mathsf{vk}$ along with a *quantum* signing key $|\mathsf{sk}\rangle$, which enables its holder to produce a classical, publicly verifiable signature $\sigma$. Its defining property is that signing keys are useful only once. To elaborate: a holder of a key pair $(\mathsf{vk}, |\mathsf{sk}\rangle)$ can at any point choose a classical message $m \in \{0,1\}^*$ and use a quantum computer to generate a classical signature $\sigma_m \leftarrow \mathsf{Sign}(\mathsf{vk}, |\mathsf{sk}\rangle, m)$, by a quantum polynomial-time signing algorithm. Then, the generated signature is classically verifiable publicly: $\{0,1\} \leftarrow \mathsf{Ver}(\mathsf{vk}, m, \sigma_m)$. Security says that given any key pair $\left(\mathsf{vk}', |\mathsf{sk}'\rangle\right)$, no one can generate a pair of valid signatures for the same verification key, for two different messages. Namely, $(m_0, \sigma_{m_0})$, $(m_1, \sigma_{m_1})$ such that both $\sigma_{m_0}$, $\sigma_{m_1}$ are valid signatures w.r.t. the same key $\mathsf{vk}'$, but $m_0 \neq m_1$.

Unlike ordinary digital signatures which can authenticate conflicting statements, OSS can rule out *equivocation*. However, preventing conflicting statements does not force a corrupt party to participate or send its messages on time. Thus, we aim to study how to combine non-equivocation, based on quantum information, with classical protocol mechanisms that ensure agreement. Importantly, we seek to achieve resiliency while keeping the protocol efficient, and in particular having low good-case latency.

## 1.1 Main Results

Our main results concern the task of *state-machine replication* (SMR), which we achieve through chaining instances of Byzantine Broadcast. An SMR protocol allows a collection of parties to maintain a common ordered history of inputs, such as transactions or more generally a computational state, despite faulty participants [23]. Unlike Byzantine broadcast, which asks parties to agree on one value and terminate, SMR runs continuously: inputs arrive over time, and each party maintains a growing output log. There are two requirements, under the stated network and corruption assumptions. *Consistency* means that honest parties' logs never conflict – one may be shorter, but must be a prefix of the other. *Liveness* asserts that inputs received by all honest parties eventually appear in all their logs. Our results use one-shot signatures (Definition 5), a public-key infrastructure (PKI), and the SMR protocols additionally require a public random leader election oracle[2]. We show the following, implicitly against quantum polynomial-time adversaries and static corruptions.

[1] That is, without assuming ongoing trusted hardware. This is a non-standard setting which we do not focus on in our discussions in this work.

[2] A public random leader election oracle can be instantiated in the random oracle model, but may be weaker than that.

**Theorem 1.** *For any $f < n$, assuming one-shot signatures and PKI, there is a synchronous Byzantine Broadcast protocol resilient to $f$ corruptions and with good-case latency $\delta$ in the unsynchronized start model.*

Recall classically for any $f \geq \frac{n}{3}$, the good-case latency of Byzantine Broadcast is at least $\Delta + \delta$ in synchrony and broadcast is impossible for this level of corruption in partial synchrony. In synchrony, $\Delta$ must be set very high to ensure it always holds, so classically very fast protocols which tolerate $f > \frac{n}{3}$ in the good case are simply not possible. But we do not stop at $\frac{n}{3}$; our protocol and its good-case latency hold for any $f < n$. Classically, this would require good-case latency of $(\frac{n}{n-f} - 1)\Delta$, which would be extremely slow for large values of $f$. One-shot signatures allow us to have maximal resilience while also proceeding nearly instantaneously in the good case.

We then extend this approach to SMR using what we call a *chainable* broadcast protocol that preserves the one-delay good-case latency across successive iterations. Our SMR protocols use independently and uniformly chosen leaders, each proposing a batch of inputs in its iteration. *Good-case latency* measures the time from an honest leader's entry into the iteration until all honest parties include its proposal in their logs and advance to the next iteration. Our synchronous construction guarantees consistency, liveness, and one-delay good-case latency for the active participants, but does not make such guarantees for inactive or silent participants (e.g. light clients):

**Theorem 2.** *Assuming OSS, a PKI, and a random leader election oracle, we have that for any $f < n$, there is a synchronous SMR without light clients protocol with $\delta$ good-case latency resilient to any $f$ static Byzantine corruptions.*

For partial synchrony, we additionally study *iteration-based expected latency*, which accounts for delays caused by malicious leaders. We fix an iteration $i$ before leader-election randomness, with all honest entries after $GST$ and the input known to each honest party before entry. The metric then measures the expected delay from the last honest entry until the input appears in every honest party's log. Altogether, we show the following.

**Theorem 3.** *Assuming OSS, a PKI, and a random leader election oracle, for any $f < \frac{n}{2}$, there is a partially synchronous SMR protocol resilient to $f$ Byzantine corruptions with $2\delta$ good-case latency and $\frac{f}{n-f} \cdot (2\Delta + \delta) + 2\delta$ iteration-based expected latency such that consistency holds even for $n - 1$ corruptions.*

Note classical protocols cannot tolerate $f \geq \frac{n}{3}$, while our protocol tolerates any $f < \frac{n}{2}$, matching protocols using trusted hardware like [7, 21] to prevent equivocation (we discuss these works more in Section 1.2). Further protocols with good-case latency of $2\delta$ like ours are impossible classically for $f \geq \frac{n}{5}$ [4]. As for expected latency, the current state of the art [25] even for protocols tolerating only $f < \frac{n}{5}$ achieves $\frac{f}{n-f} \cdot (2\Delta + 2\delta) + 2\delta$,[3] which our protocol slightly improves upon. While not explicitly proven, our partially synchronous protocol can tolerate outside parties (light clients) who may join late and simply observe the protocol (receiving messages, but not sending them) as the state of the protocol may be safely inferred from the quorum certificates it creates. As a final contribution, in the body of our paper we prove a generic compilation transformation (Theorem 13) adding low good-case latency to a given Byzantine broadcast in the synchronous setting, in Section 4.

[3] This expected latency was derived setting their $\Delta_{timeout}$ to $2\Delta$ as any lower does not guarantee progress with a good leader.

**On the Tightness of Our Results.** Our synchronous protocols can be trivially seen as having the *optimal* good-case latency because all honest parties must deliver output and have no prior information on the sender's input, so at least one message delay is necessary. The resilience of our synchronous protocols is optimal as well, as they tolerate arbitrary corruptions. The partially synchronous side is somewhat more complex, however. We do not explicitly prove impossibility results in this work for arbitrary quantum-enabled protocols. However, we do note here that one-shot signatures alone intuitively should not make a difference over digital signatures in the crash fault setting in which corrupt parties may not deviate from the protocol but may crash. This is because equivocation would never occur in the crash fault setting as parties never deviate from the protocol. Since the crash fault setting is easier (or no harder) than the Byzantine fault setting, this suggests classical crash fault impossibility results likely hold for Byzantine fault tolerant protocols using OSS, but not necessarily more generally exploiting quantum computation. In the crash fault setting, [20] shows $2\delta$ good case latency is optimal and [15] shows minority corruption resilience is optimal, matching our results. We caution the reader our results *are not* necessarily optimal for protocols able to do arbitrary quantum computation.

### 1.2 Related Work

**Consensus without Equivocation.** Ours is not the first work to consider what is possible when corrupt parties may not *equivocate*. Equivocation may loosely be defined as when a corrupt party sends contradictory messages to two different honest parties such as sending a vote for one proposal to party $A$ and sending a vote for a different proposal to party $B$. [11] show that combining digital signatures and a guarantee the adversary does not equivocate allows for a transformation from any crash fault-tolerant consensus protocol to a Byzantine fault-tolerant one. The transformation has several weaknesses, and it was improved upon by [9], but it still does not precisely preserve the runtime of the original protocol. So, we prove our results directly using one-shot signatures, and using a different transformation.

**On trusted hardware assumptions.** Many recent works [6, 7, 27, 21] use trusted hardware to prevent corrupt parties from equivocating. There are two main separations between our work and theirs. First, one-shot signatures are not an assumption of trust, but rather a constructable cryptographic primitive[4]. The works using trusted hardware are assuming not only one-time trust, but *ongoing* trust. This means that, as long as the system is running, the hardware needs to act as a trusted party for its designated task. In particular, it assumes that not even once during the entire lifetime of the protocol does the adversary successfully tamper with the hardware in order to equivocate. Second, even compared against trusted hardware assumptions, to the best of our knowledge, in the Byzantine fault tolerant setting, our synchronous protocols are the first to guarantee single-delay good-case latency even for minority corruption, let alone arbitrary corruption, and our partially synchronous protocol is the first explicitly proven or claimed to be consistent no matter the number of corruptions.

**Classical Good-Case Latency.** [4] characterizes what good-case latency is feasible for the task of broadcast, under every corruption threshold, across both synchrony and partial synchrony. Their lower bounds that are relevant to our work are summarized in Table 1, which itself is simply a subtable of the table in [4].

[4]OSS only uses a common reference string (CRS), which is a classical string $\mathsf{CRS}$ that is sampled once in the beginning and is useful for the arbitrary continuation of the system. For example, it can be generated by a malicious-majority-resilient multi-party computation, that needs to be executed only once.

| Problem | Timing Model | Resilience | Lower Bound |
|---|---|---|---|
| Psync-BB | Partial Synchrony | $n \geq 5f - 1$ | 2 rounds |
| | | $3f + 1 \leq n \leq 5f - 2$ | **3 rounds** |
| BB | Synchrony | $0 < f < n/3$ | $2\delta$ |
| | | $f = n/3$ | $\boldsymbol{\Delta + \delta}$ |
| | | $n/3 < f < n/2$ | **sync start** $\boldsymbol{\Delta + \delta}$ **unsync start** $\boldsymbol{\Delta + 1.5\delta}$ |
| | | $n/2 \leq f < n$ | $\boldsymbol{(\lfloor \frac{n}{n-f} \rfloor - 1)\Delta}$ |

Table 1. Good-Case Latency Lower Bounds from [4]. Psync-BB is simply the natural extension of Byzantine Broadcast to the partially synchronous setting.

Along the way to the complete characterization in [4], works such as [1, 3, 2] gave positive results for good-case latency as well.

**The Classical Consistency-Liveness Tradeoff.** [26] showed that partially synchronous SMR protocols in the classical world (without trusted hardware) carry a distinct tradeoff between how many corruptions they can tolerate for safety vs for liveness, even without light clients. In particular, in partial synchrony, if the protocol must guarantee consistency under $f_c$ corruptions and liveness under $f_\ell$ corruptions, then it must be that $f_c + 2f_\ell < n$. So, each corruption tolerated for liveness removes two corruptions that could have been tolerated for consistency. In particular, this makes a protocol like ours, which is consistent no matter the number of corruptions, impossible classically without foregoing liveness in the presence of any corruption.

**Corrupt Majority Protocols and Expected Latency.** Dolev-Strong [14] showed a synchronous Byzantine Broadcast protocol tolerating any $f$ corruptions, but with $\Omega(f) \cdot \Delta$ latency, even in expectation. The current state of the art [28] achieved expected $O\left(\left(\frac{n}{n-f}\right)^2\right) \cdot \Delta$ latency, but the tightest lower bound known (for any $f$, even below one half) is $\Omega\left(\frac{n}{n-f}\right) \cdot \Delta$, proved in [16]. Our synchronous protocol focuses only on good-case latency and does not consider expected latency. Nonetheless, we expect our ideas could be used in conjunction with [28] to improve our protocol's expected latency to $O\left(\left(\frac{n}{n-f}\right)^2\right) \cdot \Delta$, although it should be noted the protocol in [28] is not automatically post-quantum secure just by using post-quantum secure signatures, as the protocol uses the Decision Linear assumption in bilinear groups. Furthermore, one-shot signatures do not appear to help close the gap between [28] and the lower bound in [16], at least not in a straightforward manner. The reason is that the central issue involves parties simply not sending all the messages the protocol instructs them to as opposed to any equivocation which OSS could help with.

## 2 Technical Overview

In this section we give a high level explanation of our constructions and proofs. We start by recalling what one-shot signatures are, and their context in our work. We then proceed to describe a generic Byzantine broadcast compiler in the synchronous model (Section 2.1), a state-machine replication (SMR) protocol in the synchronous model (Section 2.2) and then our always-consistent SMR in

the partially-synchronous setting (Section 2.3). All of our protocols satisfy good-case latency, as described in Section 1.1.

**One-shot signatures.** Let us recall the definition of one-shot signatures (OSS) (Definition 5). An OSS scheme is given by polynomial-time algorithms $\mathsf{TokenGen}$, $\mathsf{Sign}$, $\mathsf{Ver}$, where there is an additional classical probabilistic polynomial-time algorithm $\mathsf{Setup}$, for initializing a common reference string (CRS), once for the entire lifetime of the system: $\mathsf{CRS} \leftarrow \mathsf{Setup}\left(1^\lambda\right)$, where $\lambda \in \mathbb{N}$ is the security parameter.

- $\mathsf{TokenGen}$ is a quantum algorithm which allows sampling a pair $(\mathsf{vk}, |\mathsf{sk}\rangle) \leftarrow \mathsf{TokenGen}\left(\mathsf{CRS}\right)$ consisting of a public classical verification key and a quantum signing key. Implicitly in the definition, sampling can be done arbitrarily many times, and with high probability (due to the security of the OSS scheme, as explained next) the sampled key pairs are all going to be distinct.
- The quantum signing algorithm $\mathsf{Sign}$ works as follows: given an arbitrary classical message $m \in \{0,1\}^*$ and $|\mathsf{sk}\rangle$ we can generate a classical signature $\sigma_{\mathsf{vk},m} \leftarrow \mathsf{Sign}\left(\mathsf{CRS}, \mathsf{vk}, |\mathsf{sk}\rangle, m\right)$.
- $\mathsf{Ver}$ is a classical algorithm for verifying signatures. The verifier checks the match between three components: a specific verification key, a message, and a signature $\mathsf{Ver}\left(\mathsf{CRS}, \mathsf{vk}, m, \sigma_{\mathsf{vk},m}\right) \in \{0,1\}$.

On the side of security, even after arbitrary quantum polynomial time and in particular sampling arbitrarily many key-pairs and corresponding classical signatures for messages, an adversary which is given the $\mathsf{CRS}$ cannot find a pair of two valid signatures, for two different messages $(m_0, \sigma_{m_0})$, $(m_1, \sigma_{m_1})$, $m_0 \neq m_1$ such that both signatures are valid w.r.t. the same public key $\mathsf{vk}$.

The first thing to observe is that OSS immediately gives non-equivocation with respect to a given verification key. Specifically, if one sees a triplet $(\mathsf{vk}, m, \sigma_{\mathsf{vk},m})$ consisting of a verification key, a message and a valid signature for that key and message, that party knows that there is nowhere else in the system, either now or later, that there is a different triplet $\left(\mathsf{vk}, m', \sigma_{\mathsf{vk},m'}\right)$, having a valid signature but for a different message $m' \neq m$ – otherwise, the security of the OSS will be violated.

Even given non-equivocation, OSS does not seem to solve the issue on its own. A malicious sender can still withhold its signature, reveal it to only one party, or release it just before another party's timeout. It could be that one of the above will cause a recipient to immediately decide the signed value while other honest parties eventually decide $\perp$ – a situation which a secure protocol should prevent.

In what follows, we describe our different solutions. Recall $\delta$ is the actual message-delay and $\Delta$ is the upper bound $\delta \leq \Delta$. The bound $\Delta$ is known and $\delta$ is not; furthermore $\delta$ may vary from execution to execution. Party (and clock) start times may differ by up to $\sigma$ where $\sigma \leq \delta$ in synchrony and $\sigma \leq GST$ in partial synchrony. For the full technical specification of the setting in our synchronous and partially-synchronous models we refer the reader to Section 3.

## 2.1 Synchronous Maximally Resilient Fast Byzantine Broadcast

Our first result is a generic good-case latency compiler (Theorem 13). We show how to turn any (post-quantum) synchronous broadcast protocol into one with the same fault tolerance, but having $\delta$ good-case latency. In particular, assuming OSS and PKI, we construct a Byzantine broadcast for every $f < n$, even with un-synchronized starts: honest parties agree and eventually terminate, and an honest sender's input is output everywhere within one actual message delay $\delta$.

In the classical setting (i.e., not using quantum information and computation) authenticated broadcast already tolerates $f < n$ [14]. If we want good-case latency, classical lower bounds require $2\delta$ for $0 < f < n/3$, jump to $\delta+\Delta$ latency once $f \geq n/3$, and $\Omega\left(\frac{n}{n-f} \cdot \Delta\right)$ in the dishonest-majority regime [4], namely, where $n/2 \leq f < n$. Here, we will explain how to use quantum information to get the optimal $\delta$, while keeping security against a dishonest majority.

**Early output by using a signature as a proof.** Our goal here is to structure the protocol so that if any honest party receives a valid OSS-signed message $m$ from the sender in a good-case timeout window, they can output $m$ immediately and then convince the other parties to output $m$ as well. One-shot signatures ensure that there can be only one valid, signed message from the sender, but we still need to be careful all parties terminate with agreement on whether anyone saw such a signed message in time. For this, we use the assumed broadcast protocol: parties wait a small timeout period ($2\Delta$) and then broadcast the signed value they received if any and otherwise broadcast $\perp$. If any of the broadcasts were a valid signed message from the original sender, parties output that value. The broadcast protocol ensures all parties agree on each other's broadcasted values, so any honest party receiving a valid, signed message from the sender early on may output it knowing all other honest parties will see this value as its broadcast value. Nonetheless, they still need to participate in the protocol until the broadcasts are finished to ensure the other honest parties output the value as well.

**Protocol outline.** A bit more formally, let $S$ be a sender with input message $m \in \mathcal{D}$ (and input space $\mathcal{D}$, where $\perp \notin \mathcal{D}$) and a publicly authenticated OSS verification key $\mathsf{vk}$. For a known sender $S$ and sender's key $\mathsf{vk}$, we denote by $s = \langle m \rangle_S$ a signature on the message $m \in \mathcal{D}$ with key $\mathsf{vk}$. Let $BB$ be any post-quantum synchronous Byzantine broadcast protocol tolerating $f$ corruptions and unsynchronized starts. Its instances below use separate signature tags. All deadlines are local, and messages arriving at a deadline are processed before the corresponding timeout action.

**Protocol 1:** Broadcast with immediate good-case output

1. **Send.** At its start, $S$ sends $s = \langle m \rangle_S$, signed with its OSS key, to all parties.
2. **Early output.** If $P$ has not output and receives a valid sender-OSS-signed $s' = \langle m' \rangle_S$ by local time $2\Delta$, it stores $s'$, outputs $m'$, and continues participating.
3. **Fallback.** At local time $2\Delta$, $P$ starts $BB_P$ as sender, with input its stored $s'$ if it output early, and $\perp$ otherwise. It also participates in $BB_Q$ for every other party $Q$.
4. **Finish.** After all $BB_Q$ instances complete, a party that has not output, outputs $m'$ if any instance outputs a valid sender-OSS-signed $\langle m' \rangle_S$, and $\perp$ otherwise. Every party then terminates, preserving any earlier output.

**Proof outline.** We next explain at a high level why the protocol satisfies three properties: (1) consistency, (2) validity and good-case latency, and (3) termination and maximal resilience.

*Consistency.* Any two non-$\perp$ decisions agree due to OSS security, because they require signatures under the same sender OSS key $\mathsf{vk}$. To rule out disagreement between a value and $\perp$, consider an honest party that outputs early. It supplies the signed value to its own fallback instance, so validity of that instance ensures that everyone eventually obtains the value. Thus nobody can decide $\perp$. If no honest party outputs early, agreement of the fallback instances gives everyone the same vector of fallback outputs, and hence the same final decision. The fallback is therefore resolving

whether to use the unique value, not choosing between competing values.

*Validity and good-case latency.* An honest sender's message reaches everyone within $\delta$ of its start. Even the earliest-starting recipient sees it by local time $\sigma + \delta \leq 2\Delta$, so everyone outputs the sender's input on receipt. Notice, the deadline is set to $2\Delta$ instead of just $\Delta$ to accommodate the clock skew $\sigma$.

*Termination and resilience.* The common local fallback deadline preserves the start skew required by $BB$. Every instance therefore terminates under its assumed fault bound, as does the outer protocol. Note that outputting early does not mean terminating early: an early-deciding party must remain available for its fallback instance but luckily, this can be done in parallel, that is, an early-deciding node can open a thread to deal with the fallback subroutine, but continue to the next iteration in the meanwhile. Using Dolev–Strong with post-quantum digital signatures and the unsynchronized-start adjustment gives resilience to every $f < n$. Finally, due to the OSS signature acting as a proof, an early-deciding node can continue safely.

## 2.2 Synchronous Maximally Resilient Fast SMR without Clients

We next explain our protocol and proof behind Theorem 2. Here, we construct a synchronous SMR without clients, with a good-case latency of just $\delta$ for every $f < n$, assuming OSS, PKI for OSS keys (as elaborated in Section 3.1, specifically in paragraph 3.1), and independent random leader election. As mentioned in Section 1, in the classical setting a Byzantine Broadcast protocol with good case latency below $\Delta + \delta$ for $f \geq n/3$ is impossible and this is easily seen to hold for iteration-based good case latency of SMR as well.

**Protocol ideas and outline.** Our previous generic compiler is not directly used here, and we improve on the generic compiler in two ways. First, rather than instantiate its fallback with $n$ Dolev-Strong broadcasts, we construct a single modified Dolev-Strong protocol, saving roughly a factor of $n$ in communication. Second, we arrange for each honest party to terminate as soon as it outputs, with all honest parties finishing within $\delta$ of one another. This lets them immediately begin the next broadcast while preserving the start-skew bound, and hence the one-delay good-case latency, across iterations. We call this intermediate component a *chainable broadcast*.

Let $f < n$ and let $S$ be the sender. A valid $k$-certificate for a value $m \neq \perp$ consists of $S$'s OSS signature on the instance-tagged value and ordinary signatures on that same value from $k-1$ other distinct parties. Thus a 1-certificate is simply the sender's message. Only the sender needs OSS here; the additional signatures record that more parties have encountered the value.

A $k$-certificate is accepted up to local time $2k\Delta$. Each added signature buys another $2\Delta$ of acceptance time, enough for both a forwarding delay and the difference between local clocks. This is what lets an honest recipient relay the value and safely stop without waiting for feedback from other parties. As in the previous section, eligible messages at a deadline are handled before timing out.

**Protocol 2:** Chainable broadcast $\Pi_{chainable}$

1. **Send.** At its start, $S$ sends its OSS-signed input to all parties, including itself, as a 1-certificate.

2. **Accept and relay.** While active, if $P$ receives $(m', \pi)$, where $\pi$ is a valid $k$-certificate by local time $2k\Delta$, where $1 \leq k \leq f+1$, it appends its classical signature over $m'$ unless $P$ is the sender. It sends the resulting certificate to all parties, outputs $m'$, and terminates.

3. **Timeout.** If still active at local time $2(f+1)\Delta$, $P$ outputs $\perp$ and terminates.

**Proof outline for the chainable broadcast.** Let us briefly argue why the protocol is chainable.

*Consistency.* As in the case of Byzantine broadcast, the security of OSS rules out two different non-$\perp$ outputs. Now, the question is whether a value can be accepted too late to prevent someone else's $\perp$ output, making two honest parties output conflicting outputs and thus breaking consistency. Suppose the sender is corrupt, and consider the first honest party $P$ to accept a value. Its certificate has at most $f$ signatures: otherwise it already contains an honest signature, whose signer must have accepted earlier. If $P$ accepts a $k$-certificate by its local time $2k\Delta$, its relay reaches every honest party by that party's local time $2k\Delta + \delta + \sigma \leq 2(k+1)\Delta$. At this point, the relayed certificate has $k+1 \leq f+1$ signatures, so it is eligible everywhere and arrives no later than anyone's final deadline. In particular, no honest party could already have timed out. It follows that everyone outputs the same value and additionally, if no honest party ever accepts a value, everyone outputs $\perp$.

*Termination and synchronized completion.* In the value case (i.e., where some honest party accepts a non-$\perp$ value $m$), everyone finishes within $\delta$ of the first honest acceptance. In the all-$\perp$ case, everyone finishes at the same local deadline, so completion times differ by at most $\sigma \leq \delta$. Every party terminates by local time $2(f+1)\Delta$. This gives the mildly synchronized output property needed for composition (this is all proved formally in Section 5.1).

*Validity and good-case latency.* With an honest sender, everyone receives its 1-certificate by local time $\sigma + \delta \leq 2\Delta$, and outputs and terminates within $\delta$ of the sender's start. Note that similarly to the case of Byzantine broadcast, for the case of fast termination, we are using the fact that an OSS signature can serve as a proof of consensus. The OSS lets a party stop after the first acceptable value, knowing that no later certificate can introduce a competing one. The timed relays separately ensure that stopping does not leave anyone behind.

**From chainable broadcast to SMR.** Once we have chainable broadcast, turning it into SMR is fairly natural. In a nutshell, the protocol proceeds in iterations, and in each iteration one party is chosen as the leader and acts as the sender of that iteration's broadcast. Every replica (that is, every honest party running the SMR protocol and maintaining its own copy of the output log) keeps a growing output log together with a buffer of inputs that have not yet been committed. For each iteration $i$, a leader-election oracle independently chooses a uniform leader $L_i$, who broadcasts a block containing its pending inputs. To keep the different iterations separate, each broadcast uses its own signature tag and a fresh, publicly authenticated OSS verification key for its leader. As before, we only consider the replica logs here, and do not give a separate confirmation rule for external clients.

**Protocol 3:** Synchronous SMR $\Pi_{SMR}$

Start in iteration 1 with an empty output log.

1. **Enter $i$.** Activate $\Pi^i_{chainable}$ with sender $L_i$. The leader supplies an ordered block containing all its buffered inputs not already in its output log.

2. **Append and advance.** On broadcast output $B$, append its entries in order if $B \neq \perp$; otherwise leave the log unchanged. Immediately enter iteration $i + 1$.

**Proof outline for SMR.** Let us go over the main arguments for why our SMR protocol satisfies its needed guarantees.

*Start synchronization and consistency.* The first iteration has the model's prescribed start skew. Inductively, synchronized completion of one broadcast gives start skew at most $\delta$ for the next, which implies every instance satisfies the assumptions used above. Broadcast agreement then gives the same block, or the same skipped iteration, at every honest replica. Replicas may be at different iterations, but they append the same sequence of blocks in order, making their logs prefix-consistent even when compared at different times.

*Liveness.* Note that the bounded broadcast termination ensures that a corrupt leader cannot stop iteration changes. Once an input has reached every honest replica, a subsequent honest leader includes it unless it is already in the agreed log. Such a leader appears with independent probability at least $(n - f)/n$ per iteration, so accordingly, for a fixed run of $r$ elections, the probability that all leaders are corrupt is bounded by

$$(f/n)^r \leq \exp\left(-r \cdot \frac{n-f}{n}\right) .$$

So, for security parameter $\lambda$, choosing $r$ of order $\frac{n}{n-f} \cdot \omega(\log \lambda)$ makes this probability negligible. By union bound, the probability that the adversary is able to find such a string of $r$ bad leaders is negligible, since the adversary is polynomially bounded (this holds even for a quantum adversary). Together with the per-iteration time bound, this gives eventual liveness, even when only one replica is honest (note that the price we pay in the case where there are many corrupt parties is that the wait for an honest leader may simply be much longer).

*Good-case latency.* This case is the simplest of the three. An honest leader's broadcast both outputs and terminates within $\delta$ of its entry. Thus all honest replicas append their block and enter the next iteration within that same $\delta$, rather than waiting for the (worst-case) deadline. This establishes the result in Theorem 2. OSS preserves the fast decision also here, while synchronized completion is the extra protocol ingredient that preserves it under repetition.

### 2.3 Partially Synchronous Fast and Always-Consistent SMR

We now turn to partial synchrony. Unlike synchrony, partial synchrony allows arbitrarily long message delays before an unknown *global stabilization time* $GST$, only after which messages between honest parties arrive within $\delta \leq \Delta$. Similarly to the synchronous setting, the parties know $\Delta$ but neither $\delta$ nor $GST$. Consistency must hold throughout the execution, whereas liveness is required only after the network stabilizes.

Our protocol guarantees progress only for $f < n/2$, however it maintains consistent logs *always*, for any $f < n$. The challenge is that, before $GST$, a party cannot distinguish a silent leader from one whose messages are honestly delayed. In particular, not receiving a proposal does not mean that nobody else has received it, voted for it, or even committed it. We therefore need a way

to move past unsuccessful iterations without allowing later leaders to undo earlier decisions. We develop the construction by first considering a single iteration, then explaining how to connect successive iterations without losing either consistency or progress. We explain our protocol in the following steps. Our protocol takes much inspiration from [10]. The main change is the use of one-shot signatures, which enables us to reduce the quorum size and remove one round of voting per leader. This results in improving both resilience and latency.

**Step 1: certifying a proposal.** Suppose the leader $L_i$ proposes a block $B_i$. A natural way to support a decision is to ask parties to send OSS-signed votes for $B_i$, and require more than $n/2$ votes before committing it. We call such a collection a *vote certificate*, denoted $QC_i(B_i)$. Each party has separate, publicly authenticated OSS keys for proposing and voting in each iteration. In particular, even a corrupt party cannot issue two different votes under its iteration-$i$ voting key.

The useful property here is that two conflicting certificates cannot exist, even when most parties are corrupt. Indeed, any two sets of more than $n/2$ voters intersect. If they certified different values, a party in their intersection would have signed both under the same OSS key, contradicting OSS security. An honest majority is needed to guarantee that enough votes will be sent, but not to rule out conflicting certificates. In our context, note that this separates the role of the honest-majority assumption in progress from its usual role in consistency.

**Step 2: certifying that an iteration can be skipped.** This observation handles conflicting certificates within one iteration, but says nothing yet about later iterations. For example, what should happen if the leader sends nothing, or its proposal receives too few votes? On the other hand, a party cannot safely discard the iteration only because it has not seen a certificate, because that certificate might already exist elsewhere.

We resolve this by letting parties use their vote to support *skipping* the iteration as in [10]. Specifically, a party that has not entered the next iteration by the end of its $2\Delta$ timer sends an OSS signature on $(\mathsf{vote}, i, \perp)$, using the same key it would have used to vote for a block. More than $n/2$ such votes form a skip certificate $QC_i(\perp)$. The same intersection argument implies that $QC_i(\perp)$ cannot coexist with $QC_i(B_i)$. Thus, rather than guessing from silence that a block certificate does not exist, parties obtain evidence that such a certificate cannot exist. Notice that the timeout only determines when an honest party votes to skip; the certificate, not the timeout, justifies leaving the iteration out of a proposed chain.

**Step 3: separating leader change from deciding on the proposal.** It may now seem that we can require each iteration to end with either a block certificate or a skip certificate. There is still a problem: the honest votes can split between the two. For example, with five parties, two honest parties may receive the proposal in time and vote for it, while the third times out and votes for $\perp$. If the two corrupt parties remain silent, neither choice obtains the three votes required for a certificate. This can happen despite an honest majority.

Our solution, similar to [10], is to separate two things: (1) advancing to another leader, from (2) committing or discarding a proposal. A party may advance after receiving a valid proposal, even if that proposal has no certificate. The proposal is kept, and a later leader can extend it. Going forward, a certificate for the later block then commits the earlier proposal as part of its history. The point of all this is that the parties can continue working without first settling each iteration individually. Recall [10] achieved this by having two separate rounds of voting for each iteration with the first round controlling slot advancement; we are able to instead advance on seeing a valid proposal without first voting due to one-shot signatures.

**Step 4: carrying earlier proposals into later decisions.** To implement this idea, each proposed block specifies both a batch of inputs and a predecessor:

$$B_i = \left(\mathsf{Propose}, i, \left(\overrightarrow{\mathsf{txns}}_i, j\right)\right) \quad ,$$

where $j < i$ is the predecessor's iteration number, and $j = 0$ denotes the empty starting chain. The leader OSS-signs the entire proposal, including $j$. Since there is at most one signed proposal message per iteration, the index $j$ identifies a unique predecessor; following these references gives a unique chain.

Let us explain what makes such a proposal valid. Besides verifying the leader's signature, a party must have the valid predecessor $B_j$, unless $j = 0$, and a skip certificate $QC_\ell(\perp)$ for every intervening iteration $j < \ell < i$. These checks apply recursively to the predecessor. For example, a block from iteration 4 can directly extend a block from iteration 1 only if skip certificates for iterations 2 and 3 are available. An iteration may therefore be missing from a chain, but only with evidence justifying the gap.

Once a party obtains a certificate for a valid block, it commits the entire chain ending at that block. The output log consists of the batches in this chain in their order. Importantly, the predecessors do not need to have certificates of their own. Also, a skip certificate permits omitting an iteration; it does not require omitting it. Finally, a proposal that received too few votes may still be included as an ancestor of a later certified block.

**Protocol outline.** We can now collect these ideas into the following protocol. Write $\mathsf{ledger}(B_j)$ for the ordered inputs in the chain ending at $B_j$, with $\mathsf{ledger}(B_0)$ empty. As before, the leaders $L_i$ are chosen independently and uniformly.

**Protocol 4:** Always-consistent partially synchronous SMR

Each party $P$ starts in iteration 1 with an empty output log and buffers incoming inputs. All sends are to every party, including oneself, and signatures use the designated OSS keys. Parties retain earlier messages and timers, and reconsider buffered messages when their iteration or validation evidence becomes available. Messages arriving at a deadline are processed before the timeout. The exact formal protocol proven is given in Section 6.

1. **Enter $i$ and propose.** Start timer $T_i$ for $2\Delta$; cancel $T_i$ on entering $i+1$. If $P = L_i$, choose the largest $j < i$ for which $P$ knows a valid proposal $B_j$, or $j = 0$ if none exists. Send $\langle B_i \rangle_{L_i}$ extending $B_j$ with all buffered inputs absent from $\mathsf{ledger}(B_j)$.

2. **Vote and advance.** On obtaining a valid proposal $B_i$ for the current iteration, send $\langle(\mathsf{vote}, i, B_i)\rangle_P$ if $P$ has not voted in $i$ and $T_i$ has not expired. Whether or not a vote is sent, enter iteration $i+1$.

3. **Timeout.** If $T_i$ fires, send $\langle(\mathsf{vote}, i, \perp)\rangle_P$. This alone does not advance the iteration.

4. **Skip.** On obtaining $QC_i(\perp)$ for the current iteration, enter iteration $i+1$.

5. **Forward.** For each current or earlier iteration, forward the first valid signed proposal and the first valid vote certificate learned for that iteration if $P$ has not already done so. Include the evidence needed to validate any forwarded proposal.

6. **Commit.** On obtaining a valid block $B_j$ and its certificate $QC_j(B_j)$ for a current or earlier iteration, replace the output log with $\mathsf{ledger}(B_j)$ if this ledger is longer.

**Proof outline.** We briefly explain why these rules give consistency, progress, and the claimed latency.

*Consistency.* Consider two committed chains, certified at iterations $i \leq i'$. If the later chain contains iteration $i$, proposal uniqueness fixes the same block and ancestry, making the earlier chain a prefix. Otherwise, the later chain jumps over $i$, requiring a skip certificate $QC_i(\perp)$. This contradicts the block certificate $QC_i(B_i)$ that committed the earlier chain. Thus, committed logs cannot conflict, regardless of the corruption count or message delays.

*Progress through iterations.* Assume $f < n/2$. Forwarding lets honest parties catch up: reaching an iteration requires evidence for passing every earlier iteration, and this evidence reaches everyone within $\delta$ after stabilization. In particular, honest entry times into an iteration first reached after $GST$ differ by at most $\delta$. If everyone enters iteration $i$ by $t > GST$, then by $t + 2\Delta$ every honest party has voted. If all voted for $\perp$, their votes form a skip certificate. Otherwise, an honest party has forwarded a valid proposal, which also lets everyone advance. Either way, everyone reaches $i + 1$ by $t + 2\Delta + \delta$, even if neither choice obtains a block certificate.

*Good-case latency.* Let an honest leader enter iteration $i$ at time $s > GST$, with no honest entry before $GST$. Its proposal is valid: every iteration between its chosen predecessor and $i$ was passed using a skip certificate. The proposal and its supporting evidence reach everyone by $s + \delta$. Since honest entries differ by at most $\delta$, no honest timer expires before $s - \delta + 2\Delta \geq s + \delta$. Everyone therefore votes and advances within one delay; another delay delivers an honest majority of votes and commits the chain everywhere. This gives good-case latency $2\delta$.

*Liveness and expected latency.* Once an input reaches every honest party, a subsequent honest leader includes it or extends a chain already containing it. Together with bounded iteration duration, the same random-leader argument used in the synchronous case gives post-$GST$ liveness. For the expected bound, fix an iteration before leader-election randomness, with all honest entries after $GST$ and the input known to each honest party before entry. There are at most $f/(n - f)$ corrupt leaders in expectation before the first honest one. Charging $2\Delta + \delta$ for each preceding iteration and $2\delta$ for the honest-led iteration gives

$$E(\delta, \Delta) \leq \frac{f}{n - f} \cdot (2\Delta + \delta) + 2\delta \ ,$$

measured from the latest honest entry. Our expected-latency definition adopts a technical convention: we treat cryptographic primitives as having no correctness or security failures. Otherwise, even a negligible positive probability that an input is never committed by all honest parties would make the unconditional expectation infinite. This convention affects only this metric; our security and liveness guarantees retain their usual negligible failure probability. This completes the guarantees of Theorem 3: an honest majority ensures progress, whereas consistency does not depend on that majority.

### 2.4 On Proof of Stake

In proof of stake (PoS) blockchains such as [17, 8], parties keep track of a shared ledger similar to SMR and maintain a balance table of coins called "stake" on this ledger. The active participants are given by the set of parties holding stake according to the ledger, weighted by the number of stake coins they own. This comes with multiple challenges. First, in contrast to permissioned SMR, in PoS, the set of parties participating changes over time. Moreover, they may have not even been present let alone active before they acquired stake. Second, in permissionless settings, it is much more desirable to have security against an adaptive adversary: i.e. one who may corrupt parties adaptively as the execution goes on instead of only at the beginning. This creates history

rewriting issues in which parties who long ago held stake become corrupt and attempt to "fork" the chain by simulating an alternate history in which they never sold their stake. Parties newly joining cannot identify which history is correct. The standard solution for this is to use the secure erasure assumption, i.e. assume parties are able to erase secrets from their state so that, if a party is later corrupted, the adversary cannot learn the erased secrets. [17] showed how to use this assumption to avoid the history rewriting attack by erasing signing keys after they are used and create a secure PoS blockchain. Unfortunately, parties have no incentive to perform such erasures even if it was guaranteed they could, and in fact they may be incentivized not to in order to later sell their old keys long after selling their stake and losing any incentive to behave honestly.

It has already been pointed out by [5] that one-shot signatures can take the place of the secure erasure assumption for this purpose in the same way our protocols in this work prevent the adversary from equivocating. This is certainly valuable: it replaces an assumption we suspect is often false with a cryptographic assumption. But our protocols in this work had some very strong properties, and a natural next question is whether or not the protocols could be adapted to the proof of stake setting without losing these properties. On the synchronous side, we note our SMR protocol heavily relies on parties being present throughout the protocol and actively participating always, hence there does not seem to be a natural adaptation to proof of stake. For a similar reason, the classical protocol in [14] is not used in proof of stake, despite being resilient to arbitrary corruption. For our partially synchronous protocol, it is not clear to us if the good-case latency of the protocol would be maintained, but we conjecture that using a VRF and random seed as in [17] would permit a proof of stake version of the protocol which remains always consistent. This would be an incredibly strong property for a proof of stake blockchain to have, ensuring there can never be forks regardless of how much of the stake becomes corrupted. In particular, we conjecture the following:

**Conjecture 4.** *Assuming OSS, there is a proof of stake protocol which is consistent regardless of the fraction of stake owned by corrupt parties and live when* $> \frac{1}{2}$ *of the stake is owned by honest parties.*

We believe the always consistent property would survive the transformation because the central argument that two conflicting certificates cannot exist should not meaningfully change assuming certificates now require over half of the stake voting behind them as opposed to over half the parties. As a first work exploring the applications of one shot signatures to consensus, however, we do not attempt to prove the conjecture here and stick to the permissioned setting. We leave it as an exciting open problem, which could meaningfully change the landscape of proof of stake.

# 3 Model and Preliminaries

We next formalize our main notions and default settings which apply throughout this work.

**Parties.** We operate in the permissioned environment in which there are $n$ parties such that all parties are aware of each other. Parties are modeled as polynomial time quantum interactive Turing Machines.

**Adversarial model.** We consider a static adversary who may choose up to any $f \leq n$ parties to be spawned as corrupt but may not corrupt parties after. Corrupt parties are controlled completely by the adversary and may behave arbitrarily (i.e., we consider what is usually referred to in the literature as Byzantine corruptions). We model the adversary as a quantum polynomial-time (in $\lambda$) interactive Turing machine, formalized as follows.

**Quantum polynomial-time (QPT) interactive adversary.** Throughout, efficient adversaries are modeled as quantum circuits with non-uniform quantum advice (i.e. quantum auxiliary input). Formally, *a polynomial-size adversary* $\mathcal{A} = \{\mathcal{A}_\lambda, \rho_\lambda\}_{\lambda\in\mathbb{N}}$ consists of a polynomial-size non-uniform sequence of quantum circuits $\{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$, and a sequence of polynomial-size mixed quantum states $\{\rho_\lambda\}_{\lambda\in\mathbb{N}}$.

For an interactive quantum adversary in a classical protocol, it can be assumed without loss of generality that its output message register is always measured in the computational basis at the end of computation. This assumption is indeed without loss of generality, because whenever a quantum state is sent through a classical channel then qubits decohere and are effectively measured in the computational basis.

**Clocks and Starts.** All parties have access to a local clock initialized to 0. Each clock proceeds at the same pace, but parties (and hence their clocks) may begin executing the protocol up to some clock skew $\sigma$ time away from each other (we will bound $\sigma$ differently depending on the network type as described below). We will say an event happens by/at/after *global time* $t$ if the event happens by/at/after local time $t$ of the earliest activated honest party.

**Network.** Parties may send messages to each other through authenticated channels. We consider two different kinds of networks: synchronous and partially synchronous. In synchrony, there are $\delta \leq \Delta$ such that any message sent by an honest party at global time $t$ is delivered by or at the very beginning of time $t + \delta$. However, only $\Delta$ is known to the protocol/parties. Beyond this, the adversary may schedule message delivery arbitrarily. If a synchronous protocol's properties hold for any clock skew $\sigma \leq \delta$, it is said to hold in the *unsynchronized start* model, and otherwise it holds in the *synchronized start* model.

In partial synchrony, there is some global time $GST$ such that messages sent at global time $t$ are delivered by or at the very beginning of $max(t, GST) + \delta$. This time both $GST$ and $\delta$ are unknown to the protocol/participants but again $\Delta$ is known. The only bound on clock skew is given by $\sigma \leq GST$ but it always holds.

We adopt the standard simplification that each party $P$ also sends itself a message $m$ when it is sending $m$ to all parties; messages sent from a party to itself are delivered instantaneously and are not subject to adversarial scheduling.

### 3.1 One-Shot Signatures

In this work we are using one-shot signatures, formally defined below.

**Definition 5** (One-Shot Signature Scheme)**.** A one-shot signature (OSS) scheme is a tuple of algorithms $(\mathsf{Setup}, \mathsf{TokenGen}, \mathsf{Sign}, \mathsf{Ver})$ satisfying the following:

- $\mathsf{CRS} \leftarrow \mathsf{Setup}\left(1^\lambda\right)$: A classical probabilistic polynomial-time algorithm that, given the security parameter, samples the classical common reference string (CRS).
- $(\mathsf{vk}, |\mathsf{sk}\rangle) \leftarrow \mathsf{TokenGen}\,(\mathsf{CRS})$: A quantum polynomial-time algorithm that takes the classical $\mathsf{CRS}$ and samples a classical public key $\mathsf{vk}$ and quantum secret key $|\mathsf{sk}\rangle$.
- $\sigma \leftarrow \mathsf{Sign}\,(\mathsf{CRS}, \mathsf{vk}, |\mathsf{sk}\rangle, m)$: A quantum polynomial-time algorithm that given $\mathsf{CRS}$, $\mathsf{vk}$, and the quantum key $|\mathsf{sk}\rangle$, given any message $m \in \{0,1\}^*$ produces a classical signature $\sigma$.
- $\mathsf{Ver}\,(\mathsf{CRS}, \mathsf{vk}, m, \sigma) \in \{0,1\}$: A classical deterministic polynomial-time algorithm which verifies a message $m$ and signature $\sigma$ relative to a public key $\mathsf{vk}$.

- **Correctness:** There exists a negligible function $\mathsf{negl}$ such that the following holds for every $\lambda$. For a fixed $\mathsf{CRS}$, let $\mathcal{G}_{\mathsf{CRS}}$ be the set of key pairs $(\mathsf{vk}, |\mathsf{sk}\rangle)$ such that, for every message $m \in \{0,1\}^*$,
$$\Pr_{\sigma \leftarrow \mathsf{Sign}(\mathsf{CRS}, \mathsf{vk}, |\mathsf{sk}\rangle, m)} [\mathsf{Ver}(\mathsf{CRS}, \mathsf{vk}, m, \sigma) = 1] \geq 1 - \mathsf{negl}(\lambda),$$
where this probability is over the randomness and quantum measurements of the signing algorithm, with $\mathsf{CRS}$, $\mathsf{vk}$, $|\mathsf{sk}\rangle$, and $m$ fixed. We require
$$\Pr_{\mathsf{CRS} \leftarrow \mathsf{Setup}(1^\lambda)} \left[ \Pr_{(\mathsf{vk}, |\mathsf{sk}\rangle) \leftarrow \mathsf{TokenGen}(\mathsf{CRS})} [(\mathsf{vk}, |\mathsf{sk}\rangle) \in \mathcal{G}_{\mathsf{CRS}}] \geq 1 - \mathsf{negl}(\lambda) \right] \geq 1 - \mathsf{negl}(\lambda).$$
- **Security:** For any QPT algorithm $\mathcal{A}$, there exists a negligible function $\mathsf{negl}$ such that for all $\lambda$ the following probability is bounded by $\leq \mathsf{negl}(\lambda)$,
$$\Pr\left[ (m_0 \neq m_1) \wedge (\mathsf{Ver}(\mathsf{CRS}, \mathsf{vk}, m_0, \sigma_0) = 1) \wedge (\mathsf{Ver}(\mathsf{CRS}, \mathsf{vk}, m_1, \sigma_1) = 1) \right],$$
where the above probability is over $\mathsf{CRS} \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{vk}, m_0, m_1, \sigma_0, \sigma_1) \leftarrow \mathcal{A}(\mathsf{CRS})$.

**Instantiation.** Amos, Georgiou, Kiayias, and Zhandry [5] defined the notion of OSS. Shmueli and Zhandry [24] constructed OSS in the classical oracle model and in the standard model. Specifically, they construct OSS unconditionally with respect to a quantumly-accessible classical oracle, and in the standard model, assuming (1) quantum polynomial security of LWE with sub-exponential noise-to-modulus ratio, (2) quantum sub-exponentially secure one-way functions and (3) quantum sub-exponentially secure indistinguishability obfuscation of classical circuits. Followup work by Huang, Shmueli, Vaikuntanathan and Zhandry [19] showed that the OSS construction of [24] can be used with asymptotically optimal quantum memory.

**Public-key infrastructure (PKI).** All our protocols assume a post-quantum secure digital signature PKI such that each party has a publicly known public key. Additionally, we assume a PKI for one-shot signatures (defined below).

**Authenticating keys across iterations.** Throughout our SMR protocols, we assume that in each iteration we essentially have a PKI for all $n$ parties, for the classical and public OSS verification keys $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$. This assumption can be made without loss of generality, assuming OSS and a PKI that happens *once* in the beginning. That is, fresh PKI is not needed for each iteration. This is inspired by the key-chaining technique underlying the ordered signatures of Amos et al. [5], with the tag-wise verification rule defined as follows. The initial PKI fixes one OSS verification key per party. Before signing a message $m$ with label $t = (\mathsf{sid}, i, \mathsf{role})$, the party generates its next OSS key pair and signs $(t, m, \mathsf{vk}')$ under its current key. By keeping an internal memory and updating it only upon seeing a valid OSS signature on a new verification key per party, each of the $n$ parties can know the current verification key of each party at all times. This adds no communication rounds, and as long as each party has memory at least $n \cdot |\mathsf{vk}|$, it does not need to remember the long chain of signatures leading to the current verification key. We now describe exactly what this would look like for each of our protocols.

For both our broadcast protocols, we simply assume a PKI consisting of the sender's OSS verification key as well as post-quantum secure digital signature public keys for the rest of the parties.

For the synchronous SMR protocol, we assume a PKI consisting of one OSS verification key and one digital signature public key for each party. The protocol is divided into increasing iterations

such that a party is only instructed to sign messages using a one shot signature from its current iteration and at most once per iteration. When instructed to sign a message $m$ using their OSS key, party $P$ generates a new OSS key pair with verification key $vk_{next}$ and signs over $(m, vk_{next})$ using the quantum signing key corresponding to their current verification key $vk_{cur}$. On receiving $(m', vk')_{vk_{old}}$ from some $P'$ (or relayed by another party but in theory signed by $P'$), $P$ waits until $vk_{old}$ is the current verification key they have recorded for $P'$ and then updates the recorded verification key to $vk'$ and processes $m'$ in addition to relaying $(m', vk')_{vk_{old}}$ to all parties (even if the protocol does not instruct them to relay the message). It is not necessary to store the prior verification keys after they have been updated. One shot signatures ensure that honest parties never differ on what the $k$'th verification key of any party was (though of course some parties may not have seen the key yet). To ensure corrupt parties do not attempt to use two different keys for the same iteration, honest parties should require each next key is used for a strictly higher iteration than the previous one and ignore messages violating this rule.

For the partially synchronous protocol, we use a PKI consisting of two OSS verification keys for each party, corresponding to the two types of messages used in the protocol: **propose** and **vote**. The same technique is used to manage parties' records of each other's verification keys but now parties sign using their propose (resp. vote) verification key for propose (resp. vote) type messages and keep track of both types of verification key for each other party.

It should be noted for the mechanisms we just gave for the SMR protocols, parties keep up-to-date records of each other's next keys but they do not necessarily know what iteration the keys correspond to. This was done intentionally because parties do not always send an OSS signed message in every iteration, particularly in the synchronous SMR protocol. Fortunately, it is not important that the keys be tied to a specific iteration, although the protocols are presented in this manner. When a key is used, however, its signed message identifies the iteration, and the sequence of iteration labels must satisfy the ordering rule that each next key of a given party be used for a strictly higher iteration (for a given message type).

**Signature Notation.** We often use the notation $\langle m \rangle$ to indicate the message $m$ is accompanied by a signature (OSS or classical depending on the context). When the signer, $P$, is known, we instead write $\langle m \rangle_P$.

### 3.2 Problem Definitions.

We consider two main problems in this work: Byzantine Broadcast and State Machine Replication (SMR). As with all of the statements in our proofs, the following definitions need hold only except with $\mathsf{negl}(\lambda)$ probability and only for the class of adversary defined in Section 3 run on input $1^\lambda$ where $\lambda$ is the security parameter.

**Byzantine Broadcast** As we only show results in synchronous networks for Byzantine Broadcast, we only give the synchronous definition. In Byzantine Broadcast, there is a distinguished party known as the sender, $S$, who receives a message $m$ in some domain $\mathcal{D}$ as input. We assume the existence of a special symbol $\perp$ which is not included in $\mathcal{D}$. In particular, when we say "any adversary", we mean any adversary respecting the conditions laid out in Section 3.

**Definition 6.** A protocol $\Pi$ solves synchronous Byzantine Broadcast with resilience $f$ if the following hold against any adversary corrupting up to $f$ parties in a synchronous network:

1. *Safety.* All honest parties output the same value.

2. *Validity.* If the sender is honest and has input $m$, all honest parties output $m$.

3. *Termination.* All honest parties eventually terminate.

**State Machine Replication.** In SMR, parties each keep track of an output log, a sequence of ordered values. We say $LOG \preceq LOG'$ if output log $LOG$ is a prefix of $LOG'$. We first give the synchronous definition.

**Definition 7.** We say a protocol $\Pi$ satisfies synchronous SMR with $f$ resilience and $T(\delta, \Delta)$-liveness if $\Pi$ satisfies the following against any adversary corrupting up to $f$ parties in a synchronous network:

1. *Consistency.* For any pair of honest output logs, $LOG$ and $LOG'$, we must have $LOG \preceq LOG'$ or $LOG' \preceq LOG$.

2. $T(\delta, \Delta)$-*Liveness.* If all honest parties receive $m$ as input by $t$, all honest parties append $m$ to their output log by $t + T(\delta, \Delta)$.

In partial synchrony, the same definition can be used but where liveness holds only for $t > GST$:

**Definition 8.** We say a protocol $\Pi$ satisfies partially synchronous SMR with $f$ resilience and $T(\delta, \Delta)$-liveness if $\Pi$ satisfies the following against any adversary corrupting up to $f$ parties in a partially synchronous network:

1. *Consistency.* For any pair of honest output logs, $LOG$ and $LOG'$, we must have $LOG \preceq LOG'$ or $LOG' \preceq LOG$.

2. $T(\delta, \Delta)$-*Liveness.* If all honest parties receive $m$ as input by $t > GST$, all honest parties append $m$ to their output log by $t + T(\delta, \Delta)$.

### 3.3 Additional Latency Metrics.

We focus on two main latency metrics in this work: good-case latency and expected latency.

**Good-Case Latency.** The good-case latency of a protocol seeks to measure the speed of the protocol when parties are behaving honestly. Our definitions will not rely on all parties being honest but only the sender or current leader being honest. The good case is extremely important because it reflects the "normal" execution of the protocol and determines the speed of the protocol while not under attack.

For Byzantine Broadcast, we define good-case latency as follows:

**Definition 9.** A synchronous Byzantine Broadcast protocol $\Pi$ has good-case latency $G(\delta, \Delta)$ if, when the sender is honest, all honest parties deliver output by the sender's local time $G(\delta, \Delta)$.

For SMR, the situation is a bit more complex and as a result definitions typically assume the protocol observes a rotating leader structure, i.e. the protocol goes through iterations such that in each iteration a leader proposes a set of values to add to the output log. We now formally define good-case latency for synchronous SMR where $L_i$ denotes the leader of iteration $i$:

**Definition 10.** We say a synchronous SMR protocol $\Pi$ has iteration-based good-case latency $G(\delta, \Delta)$ if the following holds: if $L_i$ is honest and enters $i$ by $t$ and receives input $m$ before entering $i$, all honest parties include $m$ in their output logs and advance to $i + 1$ by $t + G(\delta, \Delta)$.

And similarly we have the following definition in partial synchrony:

**Definition 11.** We say a partially synchronous SMR protocol $\Pi$ has good-case latency $G(\delta, \Delta)$ if the following holds. If $L_i$ is honest, enters $i$ by $t > GST$, receives input $m$ before entering $i$, and no honest party enters $i$ before $GST$, all honest parties include $m$ in their output logs and advance to $i+1$ by $t + G(\delta, \Delta)$.

**Expected Latency.** As we only consider the expected latency of our partially synchronous SMR protocol, we will only formally define expected latency for partially synchronous SMR. Again, we will consider a rotating leader based definition:

**Definition 12.** We say a partially synchronous SMR protocol $\Pi$ has expected iteration-based liveness $E(\delta, \Delta)$ if the following holds. For any iteration $i$ fixed before the randomness of the leader election, if all honest parties enter $i$ by $t > GST$ but not before $GST$, and $m$ is given as input to all honest parties before entering $i$, then $m$ is in the output log of all honest parties by $t + E(\delta, \Delta)$ in expectation.

For expected latency, we will assume ideal cryptographic primitives instead of ones with $\mathsf{negl}(\lambda)$ failure probability as otherwise we cannot bound the expectation since there could be a non-zero probability $m$ is never in the output log of all honest parties if e.g. the OSS scheme fails.

# 4 Maximally Resilient BB with Hyper Fast Good-Case Latency

Formally, we find:

**Theorem 13.** *Assuming a post-quantum secure synchronous Byzantine Broadcast protocol resilient to up to $f$ corruptions for some $f$ in the unsynchronized start model, there is a simple transformation using OSS to a synchronous Byzantine Broadcast protocol resilient to $f$ corruptions and with good-case latency $\delta$ in the unsynchronized start model.*

Using Dolev-Strong[5] with post-quantum signatures as the broadcast subroutine then immediately gives:

**Theorem 1.** *For any $f < n$, assuming one-shot signatures and PKI, there is a synchronous Byzantine Broadcast protocol resilient to $f$ corruptions and with good-case latency $\delta$ in the unsynchronized start model.*

## 4.1 Protocol

We now show how to transform any post-quantum broadcast protocol into one with good-case latency $\delta$. Suppose we have a post-quantum secure, synchronous $BB$ protocol resilient to any $f \leq t$ static corruptions in the unsynchronized start model for some $t$. Then consider the following protocol where we assume each instance of $BB$ has a special tag corresponding to its iteration and sender identifier which is used to create separate cryptographic primitives like signatures (e.g. by signing over both the message and the tag):

1. **Sender.** $S$ sends $\langle m \rangle_S$ to all parties, using their OSS signing key to sign.

[5] It should be noted Dolev-Strong is easily adjusted to work in the unsynchronized start model by using delay bound $\Delta' = 2\Delta$ instead of $\Delta$. Instead of formally proving this here, we prove a stronger result later on in Section 5.1.

2. **Early Output.** Any honest party receiving $\langle m' \rangle_S$ by local time $2\Delta$ for some $m'$ outputs $m'$ immediately on reception but does not immediately terminate.

3. **Vote.** At local time $2\Delta$, each honest party $P$ initiates a BB execution $BB_P$ where they are the sender. $P$ uses input $\langle m' \rangle_S$ if they have delivered output $m'$ and otherwise uses $\perp$ as input. Additionally, $P$ joins $BB_{P'}$ for each other party $P'$.

4. **Output and Terminate.** On receiving output from each $BB_{P'}$, each party $P$ does the following if they have not already output: if any $BB$ has output $\langle m'' \rangle_S$ for some $m''$, output $m''$ and otherwise output $\perp$. In any case, terminate.

We can describe the protocol informally as follows. The designated sender signs their input message using their OSS key and sends the OSS-signed message to all parties. Parties receiving such an OSS-signed message from the sender prior to $2\Delta$ output the message immediately but continue to execute the protocol. $2\Delta$ is chosen as the timeout since the sender may start up to $\Delta$ time later than any given honest party and $\Delta$ is the only known bound on the time for the sender's messages to be delivered. After $2\Delta$ time, parties each initiate their own Byzantine broadcast instance using the OSS-signed message from the sender if they have received one and otherwise use $\perp$. After all "inner" Byzantine Broadcast protocols have terminated, if any ended with output which is a message OSS-signed by the sender, parties who have not yet delivered output now output this message. Otherwise, parties who have not yet delivered output now output $\perp$. All parties then terminate.

Notice in particular, using post-quantum secure digital signatures, Dolev-Strong [14] is resilient to any $f < n$ corruptions and can be used as the broadcast subroutine with $t = n - 1$ with a small modification. Thus, we obtain a broadcast protocol with maximum resilience and nearly immediate good-case latency of $\delta$.

### 4.2 Proofs.

**Lemma 14.** *If honest $P$ and $P'$ output $v \neq \perp$ and $v' \neq \perp$, then $v = v'$.*

*Proof.* Suppose not, then both $\langle v \rangle_S$ and $\langle v' \rangle_S$ were in honest view. But this contradicts the one-shot property of OSS. □

**Lemma 15.** ***Consistency.*** *All honest parties output the same value.*

*Proof.* Suppose not, so there are two honest parties, $P$ and $P'$, who output $v$ and $v'$ respectively such that $v \neq v'$. Immediately from the previous lemma, we have that one of $v, v'$ is $\perp$. WLOG take $v = \perp$. Since $v \neq v'$, we also have $v' \neq \perp$. Then $P'$ either output $v'$ in the early output phase or the output and terminate phase. In the first case, $P$ would have received valid $\langle v' \rangle_S$ as output from $BB_{P'}$ and hence could not have output $\perp$. On the other hand if $P'$ output $v'$ in the output and terminate phase, there must be some $P''$ such that all honest parties output valid $\langle v' \rangle_S$ in $BB_{P''}$. Then again we have that $P$ could not have output $\perp$, giving our contradiction. □

**Lemma 16.** ***Validity.*** *If the sender is honest and has input $m$, all honest parties output $m$.*

*Proof.* If $S$ is honest, then all honest parties receive $\langle m \rangle_S$ by sender time $\delta$ which in particular must be $\leq 2\Delta$ local time for each honest party since honest parties may start at most $\sigma \leq \delta \leq \Delta$ time before the sender. So all honest parties output $m$ in the early output phase. □

**Lemma 17.** ***Termination.*** *All honest parties eventually terminate.*

*Proof.* This is immediate from the termination property of the $BB$ subroutine used. □

**Lemma 18.** ***Good-case Latency.*** *The protocol has good-case latency $\delta$.*

*Proof.* From the argument in Lemma 16, we know all honest parties deliver output in the early output phase when the sender is honest. Since early output occurs immediately on receiving the sender's message in the case of an honest sender, all honest parties deliver output by sender time $\delta$ when the sender is honest. □

**Theorem 13.** *Assuming a post-quantum secure synchronous Byzantine Broadcast protocol resilient to up to $f$ corruptions for some $f$ in the unsynchronized start model, there is a simple transformation using OSS to a synchronous Byzantine Broadcast protocol resilient to $f$ corruptions and with good-case latency $\delta$ in the unsynchronized start model.*

*Proof.* Immediate from Lemmas 15 to 18. □

And we also have:

**Theorem 1.** *For any $f < n$, assuming one-shot signatures and PKI, there is a synchronous Byzantine Broadcast protocol resilient to $f$ corruptions and with good-case latency $\delta$ in the unsynchronized start model.*

*Proof.* Immediate from Theorem 13 using an unsynchronized start, post-quantum secure version of Dolev-Strong (which as mentioned is easy to construct; instead of formally proving this here we prove a stronger result in Section 5.1). □

# 5 Maximally Resilient SMR without Clients with Hyper-Fast Good-Case Latency

Byzantine broadcast protocols can easily be chained into SMR (without clients) by selecting a random sequence of leaders and sequentially having each leader broadcast a proposal of values to add to the chain using the Byzantine Broadcast protocol. After waiting for the worst-case termination time, all parties append the output of the broadcast to their output log and move on to the next broadcast. This does not however preserve the good-case latency of each broadcast since parties wait for the worst-case amount of time before moving on. Fortunately, this issue is usually easily fixed by making some protocol-specific modifications. In this section we show how to change the protocol from the previous section to create an SMR protocol preserving the good case latency, but it no longer is built from an arbitrary underlying broadcast protocol. As a side effect, however, we are able to improve the efficiency by essentially running one instance of Dolev-Strong instead of $n$ per broadcast.

Recall in SMR, parties each keep track of an output log. These output logs should remain consistent in the sense that for each pair of logs, one should be a prefix of the other. We say $L \preceq L'$ if output log $L$ is a prefix of $L'$. We also want the protocol to be live: values all honest parties have received as input should eventually be added to all parties' output logs. Finally, as an analog of the good-case latency of broadcast, we would also like that once an honest leader enters their iteration $i$, all honest parties add their proposal to their output log within $\delta$ time and advance to the next iteration. In our case, we specifically want to have a good-case latency of $T = \delta$. Formal definitions of our desiderata are given in Sections 3.2 and 3.3. Our full result from this section is as follows:

**Theorem 2.** *Assuming OSS, a PKI, and a random leader election oracle, we have that for any $f < n$, there is a synchronous SMR without light clients protocol with $\delta$ good-case latency resilient to any $f$ static Byzantine corruptions.*

### 5.1 Chainable Broadcast

As mentioned, we first need to make some tweaks to the previous section's broadcast protocol before we can turn it into an SMR protocol. In particular, the missing property we need is that all parties always terminate within $\delta$ time of each other (while maintaining the good-case latency).

Moving on to the details of the protocol, we first need to define what we call a $k$-certificate, similar to the signature certificates in Dolev-Strong (and indeed our protocol will be a slightly tweaked version of Dolev-Strong).

**$k$-Certificate.** We will say $(\sigma_1, \ldots, \sigma_k)$ is a valid $k$-certificate for a value $m$ if $\sigma_1$ is a valid OSS signature by $S$ over $m$ and $\sigma_2, \ldots, \sigma_k$ are valid digital signatures of distinct parties (also not $S$) over $m$.

**Protocol.** Now we can state our protocol, $\Pi_{chainable}$:

1. **Sender.** $S$ sends $\langle m \rangle_S$ to all parties, using their OSS signing key to sign.
2. **All Parties.** Any honest party receiving a valid message/certificate pair $(m, \pi)$ where $\pi$ is a $k$-certificate for $m$ by $2k\Delta$ (local time) for $1 \leq k \leq f + 1$ does: sign over $m$ to create a $k + 1$-signature chain (unless they were the sender), send the new signature chain to all parties, and then output $m$ and terminate.
3. **Late Output.** At local time $2(f + 1)\Delta$, output $\perp$ and terminate if still active.

**Proofs.** We now turn to proving our protocol satisfies all the properties we need.

**Lemma 19.** *If honest $P$ and $P'$ output $v \neq \perp$ and $v' \neq \perp$, then $v = v'$.*

*Proof.* Same as Lemma 14. □

**Lemma 20.** ***Validity and $\delta$-Good-Case Latency.*** *If the sender is honest and has input $m$, all honest parties output $m$ by sender time $\delta$.*

*Proof.* Since the sender's signed message counts as a 1-certificate and all honest parties receive an honest sender's first message by their local time $2\Delta$ (as the sender may start at most $\Delta$ time after them) and they may not receive a valid certificate for any other value since certificates require the sender's signature, all honest parties must output $m$ upon receiving the sender's signed message. And this must necessarily happen by sender time $\delta$ since the sender sends their signed message immediately on being activated. □

**Lemma 21.** ***Mildly Synchronized Output.*** *If any honest party delivers output by global time $t$, all honest parties deliver output by global time $t + \delta$.*

*Proof.* First note if $S$ is honest, the lemma is immediately true per Lemma 20 as the sender is the first to deliver output in this case (since we defined self-messages to be instantaneous) and all honest parties deliver output by sender time $\delta$.

So, let's suppose the sender is corrupt, and let $t^*$ be the earliest global time an honest party delivers output. If $t^* \geq 2(f+1)\Delta$, the lemma is trivially true since all honest parties output no later than their local time $2(f+1)\Delta$ which is at most $2(f+1)\Delta + \sigma \leq 2(f+1)\Delta + \delta$ global time.

So, assume $t^* < 2(f+1)\Delta$. Then let $P$ be the first honest party to deliver output, which cannot be $\perp$ since $t^* < 2(f+1)\Delta$ global time and global time can never be behind any honest party's local time. So, before ever adding their signature to a certificate (since $P$ is not $S$ as $S$ is corrupt), $P$ saw a $k$-certificate for some $m$ by their local time $2k\Delta$ for some $1 \leq k \leq f+1$. Taking $k$ to be the lowest such $k$, $P$ must then have signed over $m$ to create a $k+1$ certificate received by all honest parties within $\delta$ time. If $k < f+1$, then all honest parties output $m$ by the time they receive this $k+1$ certificate because they will have received a $k+1$ certificate by their local time $2k\Delta + 2\delta \leq 2(k+1)\Delta$. Since they receive the certificate by $t^* + \delta \leq t + \delta$, the lemma holds for $k < f+1$.

Now if $k = f+1$, then note at least one honest party has already signed since there are at most $f$ corrupted parties. But then some honest party delivered output before $P$, contradicting our definition of $P$. □

**Lemma 22.** ***Consistency.*** *All honest parties output the same value.*

*Proof.* Suppose not, then by Lemma 19, we have two honest parties outputting different values such that one must be $\perp$. Consider the first honest party $P$ to output a non-$\perp$ value $m$, hence they were the first honest party outside of the sender to sign over $m$. If the sender is honest, Lemma 20 gives the lemma, hence let's consider the case of a corrupt sender. Immediately, before adding their own signature, $P$ saw a $k$-certificate for $m$ by $P$'s local time $2k\Delta$ for $k < f+1$ where we know $k < f+1$ because no honest party could have signed over $m$ yet since the sender is corrupt. But then the argument in Lemma 21 implies all honest parties output $m$ within $\delta$ time, hence we have a contradiction. □

**Lemma 23.** ***Termination.*** *All honest parties eventually deliver output and terminate.*

*Proof.* All parties deliver output and terminate at their local time $2(f+1)\Delta$ if they have not already. □

**Theorem 24.** *For any $f \leq n$, there is a Byzantine Broadcast protocol resilient to up to $f$ static corruptions and with good-case latency $\delta$ and somewhat synchronized output in the unsynchronized start model.*

*Proof.* Immediate from Lemmas 20 to 23. □

### 5.2 SMR Protocol

Finally, we can easily use the "chainable" broadcast protocol to construct SMR by randomly rotating through leaders who each broadcast the next block of inputs to add to the output log. We assume a random leader election oracle such that for each $i$, there is a publicly known $L_i$ independently randomly chosen from the set of parties. Each instance of chainable broadcast is assumed to have its own signature tag (to prepend each signing query with) as well as its own fresh OSS PKI.

**Protocol.** Consider the following $\Pi_{SMR}$:

1. The protocol proceeds in iterations $1, 2, \ldots$. Each party does the following in iteration $i$. Inputs are stored in a buffer.
2. All honest parties activate $\Pi^i_{chainable}$ on entering $i$, where $L_i$ acts as the sender and supplies input block $B$ where $B$ contains all pending inputs (not already in their output log) the sender has received in vector form.
3. On receiving output $B'$ from $\Pi_{chainable}$, all honest parties append each value in $B'$ to their output log unless $B'$ is $\perp$ and advance to $i+1$.

**Proofs.** The proofs all essentially follow immediately from the properties of the chainable broadcast protocol, but we first need to verify parties start each iteration within $\delta$ time of each other to make sure these properties hold.

**Lemma 25.** *If any honest party starts iteration $i$ by $t$, all honest parties start $i$ by $t+\delta$.*

*Proof.* We proceed inductively. The base case of $i = 1$ is trivial. Now assume the lemma holds for all $1 \leq i' < i$, we aim to show it holds for $i$. So, all honest parties start the $(i-1)$'th broadcast within $\delta$ time of each other. This ensures the properties of the $(i-1)$'th instance of $\Pi_{chainable}$ hold, and in particular the "Mildly Synchronized Output" property holds. But then immediately all honest parties enter $i$ within $\delta$ time of each other, and the lemma follows. □

Lemma 25 in particular implies that the properties of $\Pi_{chainable}$ hold for every iteration $i$. Then, we have:

**Lemma 26.** ***Consistency.*** *For any pair of honest output logs, $L$ and $L'$, we must have $L \preceq L'$ or $L' \preceq L$.*

*Proof.* Immediate from safety of $\Pi_{chainable}$. □

**Lemma 27.** ***Iteration-Based Good-Case Latency.*** *If the leader of iteration $i$, $L_i$, is honest and enters $i$ by $t$ and has received $m$ before entering $i$, all honest parties include $m$ in their output log and advance to $i+1$ by $t+\delta$.*

*Proof.* Immediate from good-case latency of $\Pi_{chainable}$. □

**Lemma 28.** ***Liveness.*** *If all honest parties receive $m$ as input by $t$, all honest parties include $m$ in their output log by $t + O(f)\frac{n}{n-f}\omega(\log\lambda)\Delta$.*

*Proof.* Let $i^*$ be the highest iteration of any honest party at $t$. Then note all honest parties will have entered $i^*$ by $t+\delta$. Further, let $j > i^*$ be the smallest iteration greater than $i^*$ such that $L_j$ is honest. Note parties spend at most $2(f+1)\Delta$ in each iteration as this is the maximum time parties spend in a given instance of $\Pi_{chainable}$. Then, all honest parties enter $j$ by $t+\delta+(j-i^*)*2(f+1)\Delta$. Per Lemma 27, all honest parties include $m$ in their output logs by $t+\delta+(j-i^*)*2(f+1)\Delta+\delta \in t + O(f)(j-i^*)\Delta$ then. Since the leader of each iteration is independently randomly selected after the corruptions are chosen, each leader is honest with probability $\geq \frac{n-f}{n}$. Then, if $i^*$ were fixed before the randomness was revealed, we would have that $\Pr[j - i^* > g(\lambda)] \in \mathsf{negl}(\lambda)$ for all $g \in \frac{n}{n-f}\omega(\log\lambda)$. While the adversary does get to select $i^*$, since the execution is polynomially bounded, they only have $poly(\lambda)$ choices of $i^*$, and hence a union bound tells us it still holds $\Pr[j - i^* > g(\lambda)] \in \mathsf{negl}(\lambda)$ for all $g \in \frac{n}{n-f}\omega(\log\lambda)$. Substituting this back in we have that all honest parties include $m$ in their output logs by $t + O(f)\frac{n}{n-f}\omega(\log\lambda)\Delta$. □

**Theorem 2.** *Assuming OSS, a PKI, and a random leader election oracle, we have that for any $f < n$, there is a synchronous SMR without light clients protocol with $\delta$ good-case latency resilient to any $f$ static Byzantine corruptions.*

*Proof.* Immediate from Lemmas 26 to 28. □

# 6 Always Consistent Partially Synchronous SMR with Improved Resilience

We now turn to the partially synchronous setting. Consensus (agreement/broadcast/SMR) protocols in this setting are known to only be able to tolerate $f < \frac{n}{3}$ Byzantine faults, however similar to [7][6] which uses trusted hardware, we show that with OSS, there is a protocol tolerating any $f < \frac{n}{2}$ faults. To our knowledge a benefit not proven in any prior work on consensus using additional assumptions is that our protocol remains *consistent* regardless of the number of corruptions, i.e. honest parties never output conflicting values even if all but two parties are corrupt. Since consistency is often considered more important to maintain than liveness as in for example [22] (in the case both cannot simultaneously hold), this is a particularly neat advancement. Classically any protocol satisfying liveness for any $t_\ell$ corruptions may not simultaneously satisfy consistency for $n - 2t_\ell$ corruptions [26] in partial synchrony. In other words, in a classical world, increasing liveness resilience by one corruption reduces the possible consistency resilience by *two* corruptions. For example, a classical partially synchronous SMR protocol tolerating just under 80% corruption for consistency can only guarantee liveness when less than 10% of parties are corrupt. In contrast our protocol tolerates any number of corruptions for consistency and is live when less than half the parties are corrupt. Formally, we show the following:

**Theorem 3.** *Assuming OSS, a PKI, and a random leader election oracle, for any $f < \frac{n}{2}$, there is a partially synchronous SMR protocol resilient to $f$ Byzantine corruptions with $2\delta$ good-case latency and $\frac{f}{n-f} \cdot (2\Delta + \delta) + 2\delta$ iteration-based expected latency such that consistency holds even for $n - 1$ corruptions.*

## 6.1 Protocol.

**OSS Setup.** Each protocol message will consist of a **type**, an iteration, and an inner message such as $(\textbf{vote}, i, m)$. The protocol proceeds in iterations, and in each iteration there are two types of messages sent, **propose** and **vote**. We assume an OSS PKI such that for each iteration and message type pair, parties have an associated verification key publicly known. This can be easily bootstrapped from a single key PKI by having parties sign over their next verification key as they use each key.

**Certificates** For this section, a valid certificate $\pi$ over a message $(\textbf{type}, i, val)$ consists of any $> \frac{n}{2}$ OSS signatures from distinct parties over $(\textbf{type}, i, val)$ (under the proper OSS keys for the message type and iteration). The only certificates we use in this protocol are of type **vote**, and we often refer to these as vote certificates.

**Leaders.** Each iteration $i$ of the protocol has an associated leader $L_i$ who we assume is publicly known and randomly chosen.

[6] Their work was done in the asynchronous setting; we see no reason why our protocol could not be similarly modified to asynchrony but chose to address the simpler and more practical case of partial synchrony.

**Valid Block Proposals.** The leader of each iteration sends a signed block proposal to all parties. An iteration $i$ proposal, $\langle(\textbf{Propose}, i, (\overrightarrow{txns}, prev))\rangle$, is valid for party $P$ if it is signed by $L_i$ (with a valid signature under the appropriate OSS verification key), is not $\perp$, and either of the following holds:

1. $i = 1$ and $prev = 0$.
2. $0 \leq prev < i$ and for each $prev < i' < i$, $P$ has seen an iteration $i'$ vote certificate for $\perp$, and $P$ has seen a valid iteration $prev$ proposal or $prev = 0$.

**Blockchains.** Notice each block proposal references a previous iteration. Since we will ensure each iteration only has a single valid proposal in honest view, this lets us define blockchains in a well defined manner. The blockchain ending at a valid (in honest view) proposal $B = \langle(\textbf{Propose}, i, (\overrightarrow{txns}, prev))\rangle$ is defined as $B$ prepended with the valid block from iteration $prev$, prepended with the valid block from that block's referenced previous iteration, and so on until reaching a block with referenced previous iteration 0.

**Ledger.** Each blockchain then induces a ledger of transactions/messages by taking the sequence of values in each block's $\overrightarrow{txns}$ field, ordered in ascending block order and in the order given by each individual block's $\overrightarrow{txns}$ field.

**Buffered Messages.** Sometimes a party may receive a message before it is ready to do anything with it; in this case the message is placed in storage so that it may be retrieved when the party is ready to process it (such as holding on to a message pertaining to the next iteration until reaching the next iteration).

Now we can describe the protocol. The following lists the actions taken by each party on entering iteration $i$:

1. **Proposal.** Let $j < i$ be the last iteration $L_i$ has seen a valid proposal, $B_j$, for; $j$ should be set to 0 in the case $L_i$ has not seen such a proposal. And let $\overrightarrow{txns}$ be the set of inputs $L_i$ has received which are not already included in the ledger induced by the blockchain ending at $B_j$. $L_i$ then sends $\langle(\textbf{Propose}, i, (\overrightarrow{txns}, j))\rangle_{L_i}$ to all parties.
2. **Timer.** $P$ sets a timer $T_i$ to fire in $2\Delta$ time. On entering iteration $i + 1$, $P$ cancels $T_i$ if it has not yet fired. If $T_i$ fires, $P$ sends $\langle(\textbf{vote}, i, \perp)\rangle_P$ to all parties.
3. **Vote.** On seeing a valid iteration $i$ proposal $\langle B_i\rangle_{L_i}$, $P$ sends $\langle(\textbf{vote}, i, B_i)\rangle_P$ to all parties if $T_i$ has not fired and advances to $i + 1$ regardless of whether the timer has fired.
4. $\perp$ **Certificate.** On receiving a vote certificate for $\perp$ for $i$, $P$ advances to $i + 1$.
5. **Echo.** On seeing a valid iteration $i'$ vote certificate (resp. proposal) for $i' \leq i$, $P$ sends the certificate (resp. proposal) to all parties if $P$ has not already received a valid iteration $i'$ certificate (resp. proposal).
6. **Output.** On seeing a blockchain ending at a valid block proposal, $B_{i'}$, for any iteration $i' \leq i$ with a valid vote certificate, $P$ replaces their output log with the ledger induced by the blockchain ending at $B_{i'}$ if it is longer than the current output log.

### 6.2 Proofs.

We first show safety for any number of corruptions, but all subsequent lemmas after the consistency lemma will assume the number of corruptions is at most $f$.

**Lemma 29.** *For any iteration $i$, if iteration $i$ vote certificates are in honest view for both $B$ and $B'$ (which may potentially be $\perp$), then $B = B'$ regardless of the number of corruptions (i.e. even if say $n-1$ parties are corrupt).*

*Proof.* Notice each party has only one OSS verification key in the PKI for voting in a given iteration. Since certificates consist of $> \frac{n}{2}$ signatures, if $B \neq B'$ this would immediately imply there were signatures over both $(\textbf{vote}, i, B)$ and $(\textbf{vote}, i, B')$ by one party. However, this contradicts the one-shot property of OSS. □

**Lemma 30.** *For any iteration $i$, if $\langle B \rangle$ and $\langle B' \rangle$ are both considered valid iteration $i$ proposals by some honest parties, then $B = B'$.*

*Proof.* This is immediate from the one shot property of $L_i$'s signature of its proposal. □

It should be noted, however, we do not necessarily have $\langle B \rangle = \langle B' \rangle$ as the signatures may be different since one shot signatures are only guaranteed to ensure there are not signatures on different messages, not that there are not two different signatures on the same message.

**Lemma 31.** *For any given block $\langle B \rangle$ which is valid in honest view, there is only one blockchain ending at $B$ in honest view.*

*Proof.* Suppose not, then there are two distinct blockchains ending at $B$: $b^1, b^2, \ldots, b^{h-1}, B$ and $b'^1, b'^2, \ldots, b'^{h'-1}, B$ of heights $h$ and $h'$ respectively. But notice both $b^{h-1}$ and $b'^{h'-1}$ must be from the same iteration (in particular the *prev* iteration referenced in $B$) and must be valid in honest view. By Lemma 30 then, we must have $b^{h-1} = b'^{h'-1}$. Iteratively applying this logic backwards, we find the two blockchains are equivalent. □

**Lemma 32.** ***Consistency.*** *If two honest parties, $P$ and $P'$, have output logs $L$ and $L'$ respectively (potentially from different times), then $L \preceq L'$ or $L' \preceq L$, regardless of the number of corruptions.*

*Proof.* Assume for the sake of contradiction there were such $P$ and $P'$ with inconsistent output ledgers. Immediately both $L$ and $L'$ must be non-empty, else they would necessarily be consistent. Now $P$ must have set its output log based on seeing an iteration $i$ vote certificate for a valid block proposal $B_i$; in particular the output log of $P$ is the ledger induced by the blockchain ending at $B_i$. We can say the same of $P'$ for some $i'$ and $B'_{i'}$. If $i = i'$, we immediately have a contradiction due to Lemma 30 and Lemma 31 as the two blockchains must be equivalent. Thus $i \neq i'$, so WLOG let's say $i < i'$. Then we have two cases.

In the first case, the blockchain ending at $B'_{i'}$ contains a block from iteration $i$, $B'_i$. Immediately then from Lemmas 30 and 31, $B'_i = B_i$ and there is only one blockchain ending at this block, hence the blockchain ending at $B_i$ is a prefix of the blockchain ending at $B'_{i'}$.

In the second case, the blockchain ending at $B'_{i'}$ does not contain a block from $i$. Then since $i' > i$, there must be some iteration $j$ such that there is a block $B'_j$ with $j > i$ and previous iteration $prev_j < i$. But notice each block in a blockchain ending at a valid block must itself be valid, so $P'$ must have seen an iteration $i$ vote certificate for $\perp$, but this immediately gives a contradiction from Lemma 29 since $P$ saw an iteration $i$ vote certificate for $B_i \neq \perp$. □

From here on, we assume a known bound $f < \frac{n}{2}$ on the number of corrupt parties.

**Lemma 33.** *If honest party $P$ receives and considers valid an iteration $i$ proposal $B$ by $t$, then all honest parties receive and consider valid $B$ by $max(t, GST) + \delta$.*

*Proof.* Per Lemma 30, there may be only one valid iteration $i$ proposal in honest view. So, the echo instruction ensures all honest parties receive $B$ by $max(t, GST) + \delta$. Now for each iteration $prev < j < i$ where $prev$ is the referenced previous iteration in $B$, $P$ has seen a vote certificate for $\perp$. Per the echo instruction, all honest parties see these by $max(t, GST) + \delta$. And (assuming $prev > 0$, else we are done) similarly, $P$ sends $B_{prev}$, a proposal valid to $P$ from iteration $prev$, received by all honest parties by $max(t, GST) + \delta$. It remains to show all honest parties consider $B_{prev}$ valid by $max(t, GST) + \delta$. However, we can do so iteratively using the same argument on $B_{prev}$ until we get to a previous proposal with previous iteration 0, finishing the proof. □

We similarly have:

**Lemma 34.** *If any honest party enters iteration $i$ by global time $t$, all honest parties enter $i$ by $max(t, GST) + \delta$.*

*Proof.* Suppose honest $P$ enters $i$ by $t$. Then $P$ has seen either a valid proposal or a vote certificate for $\perp$ in each prior iteration and has sent at least one of each by $t$. Thus all honest parties receive them by $max(t, GST) + \delta$, and per Lemma 33 they consider each proposal received from $P$ valid by $max(t, GST) + \delta$. Then, all honest parties enter $i$ by $max(t, GST) + \delta$. □

**Lemma 35.** *If there is an iteration $i$ vote certificate in honest view, at least one honest party signed over $(\textbf{\textit{vote}}, i, v)$.*

*Proof.* Since certificates consist of $> \frac{n}{2}$ signatures by distinct parties, and $f \leq \frac{n}{2}$, the lemma is immediate. □

**Lemma 36.** *If all honest parties enter $i$ by $t > GST$, all honest parties enter $i + 1$ by $t + 2\Delta + \delta$.*

*Proof.* All honest parties' timers for $i$ fire by $t + 2\Delta$, thus all honest parties have sent a vote message in iteration $i$ by $t + 2\Delta$ (either for $\perp$ or for a proposal) or have entered $i + 1$. Per Lemma 34 the second case immediately implies all honest parties enter $i + 1$ by $t + 2\Delta + \delta$. In the first case, if all honest parties sent a vote message for $\perp$ in $i$, then immediately all honest parties enter $i + 1$ by $t + 2\Delta + \delta$, because they receive a vote certificate for $\perp$ for $i$ by this time. Else, at least one honest party sent a vote message for $B \neq \perp$ in $i$ and hence saw a valid iteration $i$ proposal by $t + 2\Delta$. Then, per Lemma 33 all honest parties receive and consider this proposal valid by $t + 2\Delta + \delta$ and hence enter $i + 1$ by this time. □

**Lemma 37.** $2\delta$ ***Good-case Latency.*** *If all honest parties enter $i$ by $t > GST$ but not before $GST$ and $L_i$ is honest and has received input $m$ before entering $i$, all honest parties add $m$ to their output ledger and advance to $i + 1$ by $t + 2\delta$.*

*Proof.* Now let $t^*$ be the earliest time such that $L_i$ entered $i$ by $t^*$. Then $L_i$ sends their proposal $B$ by $t^*$, received by all honest parties by $t^* + \delta$. Per Lemma 34, no honest party could have entered $i$ before $t^* - \delta$. Then no honest timer for $i$ fires by $t^* - \delta + 2\Delta \geq t^* + \delta$. But all honest parties receive and consider valid $L_i$'s proposal by $t^* + \delta \leq t^* + \Delta$ per Lemma 33 since it is valid to $L_i$ by $t^*$. Thus all honest parties send a vote message for $B$ in $i$ by $t^* + \delta$, and hence all honest parties receive a vote certificate for $B$ by $t^* + 2\delta \leq t + 2\delta$. Since $L_i$ received $m$ prior to entering $i$, it is guaranteed the ledger induced by the blockchain ending at $B$ includes $m$. Furthermore, all honest parties advance to $i + 1$ by $t + \delta$ as they have seen $B$ and consider it valid by $t^* + \delta$. □

**Lemma 38.** ***Liveness.*** *If all honest parties have received $m$ by $t > GST$, all honest parties' output ledgers include $m$ by $t + O(1) * \omega(\log \lambda)\Delta$.*

*Proof.* Let $i$ be the highest iteration of any honest party by $t$ and let $k$ be the lowest iteration greater than $i$ such that $L_k$ is honest. Then all honest parties enter $i$ by $t+\delta$. Repeatedly applying Lemma 36, all honest parties enter $k$ by $t + O(k-i)\Delta + \delta \in t + O(k-i)\Delta$. And per Lemma 37, $m$ will be in the output ledgers of all honest parties by $t + O(k-i)\Delta + 2\delta$. We now seek to bound $k - i$. Notice each iteration has $> \frac{1}{2}$ chance of an honest leader, as the corrupt parties are chosen prior to the randomness of the leader election. Then for any *fixed* iteration $j$ and positive integer $r$, the probability all $r$ iterations after $j$ have a corrupt leader is $\leq 2^{-r}$. However, $i$ is not fixed: the adversary may adaptively choose $i$ after seeing the randomness of the leader election. Fortunately, the adversary is polynomially bounded, and hence so is the maximum iteration of any honest party (as the adversary must schedule the necessary messages from each prior iteration to be delivered to honest parties in order for them to advance). Thus via a union bound, there is a polynomial $p(\lambda)$ such that the probability each iteration in $[i+1, i+r]$ has a corrupt leader is $\leq p(\lambda) * 2^{-r}$. Taking $r \in \omega(\log \lambda)$, this probability becomes negligible. Hence, except with negligible probability $k - i < \omega(\log \lambda)$ (i.e. $k - i < r(\lambda)$ for all $r \in \omega(\log \lambda)$). So, we have that $m$ will be in the output of all honest parties by $t + O(1) * \omega(\log \lambda)\Delta + 2\delta$ (and the $2\delta$ is absorbed since $\delta \leq \Delta$). □

**Lemma 39.** ***Expected Iteration-Based Liveness.*** *Assuming ideal cryptographic primitives, for any iteration $i$ fixed before the randomness of the leader election, if all honest parties enter $i$ by $t > GST$ but not before $GST$, and $m$ is given as input to all honest parties before entering $i$, then $m$ is in the output log of all honest parties by $t + \frac{f}{n-f}(2\Delta + \delta) + 2\delta$ in expectation.*

*Proof.* Let $j$ be the first iteration at least $i$ such that $L_j$ is honest. Note then from Lemmas 36 and 37 all honest parties add $m$ to their output log by $t + (j-i)(2\Delta + \delta) + 2\delta$. Since each elected leader is honest with independent probability $\geq \frac{n-f}{n}$, we have $E[j - i] \leq \frac{n}{n-f} - 1 = \frac{f}{n-f}$. Thus in expectation all honest parties add $m$ to their output log by $t + \frac{f}{n-f}(2\Delta + \delta) + 2\delta$. □

**Theorem 3.** *Assuming OSS, a PKI, and a random leader election oracle, for any $f < \frac{n}{2}$, there is a partially synchronous SMR protocol resilient to $f$ Byzantine corruptions with $2\delta$ good-case latency and $\frac{f}{n-f} \cdot (2\Delta + \delta) + 2\delta$ iteration-based expected latency such that consistency holds even for $n - 1$ corruptions.*

*Proof.* Immediate from Lemmas 32, 37 and 38. □

## References


[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. Cryptology ePrint Archive, Paper 2019/270, 2019. URL: https://eprint.iacr.org/2019/270.

[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Brief Announcement: Byzantine Agreement, Broadcast and State Machine Replication with Optimal Good-Case Latency. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:3, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2020.47, doi:10.4230/LIPIcs.DISC.2020.47.

[3] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Byzantine agreement, broadcast and state machine replication with near-optimal good-case latency, 2020. URL: https://arxiv.org/abs/2003.13155, arXiv:2003.13155.

[4] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization, 2022. URL: https://arxiv.org/abs/2102.07240, arXiv:2102.07240.

[5] Ryan Amos, Marios Georgiou, Aggelos Kiayias, and Mark Zhandry. One-shot signatures and applications to hybrid quantum/classical authentication. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 255–268, 2020.

[6] Yackolley Amoussou-Guenou, Lionel Beltrando, Maurice Herlihy, and Maria Potop-Butucaru. Byzantine reliable broadcast with one trusted monotonic counter. Cryptology ePrint Archive, Paper 2024/774, 2024. URL: https://eprint.iacr.org/2024/774.

[7] Yackolley Amoussou-Guenou, Maurice Herlihy, and Maria Potop Butucaru. Asynchronous byzantine consensus with trusted monotonic counters. Cryptology ePrint Archive, Paper 2024/1937, 2024. URL: https://eprint.iacr.org/2024/1937.

[8] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 913–930. ACM, 2018. doi:10.1145/3243734.3243848.

[9] 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*, PODC'22, page 450–459, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538427.

[10] Benjamin Y Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. Cryptology ePrint Archive, Paper 2023/463, 2023. https://eprint.iacr.org/2023/463. URL: https://eprint.iacr.org/2023/463.

[11] 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*, PODC '12, page 301–308, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2332432.2332490.

[12] Andrea Coladangelo and Or Sattath. A quantum money solution to the blockchain scalability problem. *Quantum*, 4:297, 2020.

[13] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[14] Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983. doi:10.1137/0212045.

[15] 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.

[16] Juan A. Garay, Jonathan Katz, Chiu-Yuen Koo, and Rafail Ostrovsky. Round complexity of authenticated broadcast with a dishonest majority. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, page 658–668, USA, 2007. IEEE Computer Society. doi:10.1109/FOCS.2007.61.

[17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017. doi:10.1145/3132747.3132757.

[18] Ruomu Hou and Haifeng Yu. Optimistic fast confirmation while tolerating malicious majority in blockchains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2481–2498. IEEE, 2023.

[19] Andrew Huang, Omri Shmueli, Vinod Vaikuntanathan, and Mark Zhandry. Uncloneable cryptography in linear quantum memory. In *Annual International Cryptology Conference*, pages 247–278. Springer, 2026.

[20] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, October 2006. doi:10.1007/s00446-006-0155-x.

[21] Marc Leinweber and Hannes Hartenstein. Brief announcement: Let it tee: Asynchronous byzantine atomic broadcast with $n \geq 2f + 1$. volume 281, pages 43:1–43:7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2023.43, doi:10.4230/LIPICS.DISC.2023.43.

[22] Jacob D. Leshno, Elaine Shi, and Rafael Pass. On the viability of open-source financial rails: Economic security of permissionless consensus, 2025. URL: https://arxiv.org/abs/2409.08951, arXiv:2409.08951.

[23] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Acm Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[24] Omri Shmueli and Mark Zhandry. On one-shot signatures, quantum vs. classical binding, and obfuscating permutations. In *Annual International Cryptology Conference*, pages 350–383. Springer, 2025.

[25] Victor Shoup, Jakub Sliwinski, and Yann Vonlanthen. Kudzu: Fast and simple high-throughput bft, 2025. URL: https://arxiv.org/abs/2505.08771, arXiv:2505.08771.

[26] Srivatsan Sridhar, Ertem Nusret Tas, Joachim Neu, Dionysis Zindros, and David Tse. Consensus under adversary majority done right. In Christina Garman and Pedro Moreno-Sanchez, editors, *Financial Cryptography and Data Security*, pages 108–126, Cham, 2026. Springer Nature Switzerland.

[27] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.*, 62(1):16–30, January 2013. doi:10.1109/TC.2011.221.

[28] Jun Wan, Hanshen Xiao, Elaine Shi, and Srinivas Devadas. Expected constant round byzantine broadcast under dishonest majority. *J. ACM*, 72(4), August 2025. doi:10.1145/3748254.

[29] Mark Zhandry. Quantum lightning never strikes the same state twice. or: Quantum money from cryptographic assumptions: M. zhandry. *Journal of Cryptology*, 34(1):6, 2021.