Proofs of non-Supermajority: the missing link for two-phase BFT with responsive view-change and linear complexity

Christophe Levrat and Matthieu Rambaud

Télécom Paris, LTCI, Institut Polytechnique de Paris

Version 3 - May 2023[1]

Abstract. We consider leader-based Byzantine state machine replication, a.k.a. "BFT", under partial synchrony. We provide a generic solution enabling to match simultaneously, for the first time, three arguably gold standards of BFT: in two phases, with a responsive view change and a linear complexity per view. It is based on a new threshold primitive, which we call *Proofs of non-Supermajority* (or *PnS* for short). A PnS system enables players, each with an input number, to report their input to a leader, with extra hints enabling their efficient aggregation. Out of a threshold number k ($= 2t + 1$) of such reports, calling $v_{\max}$ the highest reported value, the leader can efficiently build a short proof that a threshold number of $k - t$ honest players have their inputs lower than this $v_{\max}$. As highlighted in the state of the art BFT [Abraham et al. Podc'23], any of our lightweight implementations of PnS can be plugged in their BFT, or the one of [Gelashvili et al FC'22], to bring down their complexities from quadratic to linear. Previous BFTs implicitely implemented PnS by either (i) having the leader multicasting the k signed reports, so this had quadratic communication complexity, or (ii) multicasting an aggregate signature on the reports, with verification complexity of $k + 1$ pairings, so this had a total quadratic computation complexity. To match our linear complexity claim, we introduce a very simple constant-sized and constant-verification implementation of PnS, built from any threshold (or multi-) signature scheme. We then bring further optimizations by introducing the following tools of possible independent interest:
(1) a simple and general optimization to BFTs, which applies to any view without a hidden lock. It removes the need for sending and verifying a PnS. Previous optimizations applied to a narrower set of views, essentially, those for which the leader of the previous one was honest and enjoyed synchrony;
(2) a simple compiler from any multisignature scheme, into an aggregate signature scheme of a special-purpose type. It operates over tagged messages, each key can sign at most one message with a given tag.

Contents

[1] Forks on the v1 (November 2020), of which the TSS-based PnS briefly announced to Podc'21. New results of independent interest are the (1) (2) and (3) (cf abstract and introduction). Some previous material of the April 2021 version is now forked in [Abspoel, Rambaud, Tonkikh, Consensus Days'22] (fast tracks) and, in a preliminary form, in section 7 of eprint 2020/1447.

1 Introduction

In order for blockchain systems to be competitive with centralized services, it is imperative to keep the latency of the consensus algorithm as low as possible while maintaining high throughput. To this end, *Byzantine consensus* [LSP82] is typically used to reach agreement on a sequence of blocks of operations. Intuitively, in a single instance of consensus, each participant, which we dub as *player*, starts with some valid value, e.g., a block of transactions. The goal is for each player to irrevocably output a valid value, which we dub as a *commit*. Moreover, all players that follow the protocol must output the same value. This specification is known as *consistency*. Some subset of t out of the n players are called "corrupt", or "Byzantine", which means that they are controlled by a malicious adversary and may arbitrarily deviate from the protocol. Non-corrupt players are called "honest". In this work, we are concerned with a much studied class of protocols designed to operate under a network model known as *partial synchrony* [DLS88]. In this model, there is an unknown moment in time, called the *global stabilization time (GST)*, such that any message sent after this moment is delivered within a known time bound Δ. In order to ensure liveness of the protocols, this bound has to be pessimistic and, in practice, most messages are delivered much faster. Hence, we use δ to denote the unknown actual message delay in a given execution ($\delta \ll \Delta$). In this work we assume that $n = 3t+1$, which is optimal for consensus under partial synchrony [DLS88]. We are concerned with the subclass of Byzantine consensus protocols which are known as *leader-based*. These protocols require for their operation a mechanism that notifies the players, of the identity of one of them, denoted as *leader*, which may change in time. We dub such protocols as *BFTs*, often leaving-out implicit the "leader-based" and the partial synchrony assumption. Beyond its practicality as such, partially-synchronous leader-based consensus is also used as a subroutine in higher-level protocols, e.g., in [AMS19; SDPV22] somehow n instances of Hotstuff [YMR+19a] or PBFT [CL99] are emulated in parallel; in [DZJR23] a parallel instance is run as a fast-path of asynchronous consensus (building on [GKS+22; LLT22]); and in [SLL10; BTA+19; VAFB22] it is used as a means to reach consensus on secret shares. Broadly speaking, we denote as a *view* the time-frame in which all players are aware of the same leader. A view is denoted as *normal* if the leader is honest and GST happens no later than the start of the view. BFTs guarantee that, in a normal view, a value is committed within a fixed delay. A desirable property is when this guaranteed delay is purely in terms of δ. For such BFTs, it is common-place to say that they have a *view-change* which is *responsive*.

PBFT [CL99] is by far the most famous BFT. It has latency of only 3δ if the first view is normal. However, PBFT has quadratic authenticator complexity in the number n of players. An *authenticator* is, roughly speaking, a string which is as fast (or nearly as fast) to verify as one signature. The *authenticator complexity* ([YMR+19a]) is, roughly, an equivalent metric of the number of signatures sent or verified by honest players. Hence, BFTs with worst-case linear authenticator complexity in every view were proposed. The first and simplest is called "two-phase Hotstuff" [YMR+19b] (which is the one put forth in their first version). If the first view is normal, then the leader decides within 4δ. However in later views it decides within $\Delta + 4\delta$. The extra delay Δ, which happens when the leader of a new view starts its role, is known as *view-change latency*. Hence, [YMR+19b] does not have responsive view-change. As will be detailed in Section 3.9, such non-responsive view change, in Δ, is paid by all protocols which follow a specific view-change mechanism, which is credited to "Tendermint" [Buc16] and "Casper" [BG17] by [YMR+19b]. Later, leader-based BFTs were proposed [YMR+19a; SDZ22b], which have linear "authenticator complexity" and a responsive view-change. The former, Hotstuff, was implemented in the late blockchain Diem [Tea21] and now in its successor [Tea22]. However, they both pay an extra *phase*, i.e., an extra latency of 2δ, at least in executions which are not "happy", as detailed in Section 5.2. Other BFTs made the converse trade-off, i.e., are always in two-phases, but at the cost of paying a quadratic authenticator complexity in views which are not "happy": Fast-Hotstuff [JNFG20] and also Jolteon [GKS+22], implemented in the former Diem [Tea21]. So it seems that the design of BFT protocols is driven by trade-offs between three main performance criteria: the number of phases in a view, responsivity of view-change, and the communication complexity. These three criteria are the ones put forth in their comparison tables by a number of leading authorities in BFT: [YMR+19a, Table 1], [GKS+22, Table 1], [ACG+23, Table I], [MN23, Table I] (and also [BBB+23, Table 1] although their synchronous setting requires orthogonal techniques). The recent statement of knowledge on BFT [SDZ22a, Table 1], by the authors of [SDZ22b], confirms that no existing BFT with optimal corruption tolerance $n = 3t+1$ achieves simultaneously two-phases, and a responsive view-change which is both responsive and with worst-case linear authenticator complexity.

Contributions In this work we provide a generic tool which enables to match, for the first time, all of the three aforementioned arguably gold-standards of BFT: in two-phases, with a responsive view change and with linear complexity per view. It is based on a new threshold cryptographic primitive, which we call *Proofs of non-Supermajority* (or *PnS* for short), which may be of independent interest. A PnS system enables players, each with an input number, to report their input to a leader, with extra hints enabling their efficient aggregation. Out of a threshold number k (=2t+1) of such reports, calling $v_{\max}$ the highest reported value, the leader can efficiently build a short proof that a threshold number of $k-t$ honest players have their inputs lower than this $v_{\max}$. Our first contribution is to single-out this primitive of PnS, and illustrate where it can be plugged in place of existing view changes. This is done in Section 3. Then, to match our complexity claim, our second contribution is a very simple constant-sized and constant-verification implementation of PnS, described in Section 4. It is based on any, black-box, non-interactive threshold (or multi-) signature scheme. A corner-case limitation is that the complexity for each player to help the leader to build a PnS, grows linearly with the gap between the highest lock of the player and the current view number v. Hence, in catastrophic scenarios where many consecutive views elapsed since the beginning of the instance, without any commit, then at some point it may be desirable to switch to another more robust implementation of PnS. Another contribution which we make is the observation (Section 6.1) that PnS can be instantiated from SNARKs of signed messages satisfying a public predicate. Such one is implemented in [AR23]. When specialized to the predicate of being in a public range $[0, v_{\max}]$, this yields an $O(\log(n))$-sized PnS with constant pairing complexity. This concludes our main complexity claims. The application of our primitive of PnS was very recently reported in the state of the art BFT [ACG+23, Table I] (which they call SNARK, probably referring to the aforementioned specific instantiation of [AR23]). There, they confirm our claim that PnS can be plugged as a replacement of the view-change of state of the art BFTs [GKS+22; JNFG20; ACG+23] in order to bring down their complexity from quadratic to linear. To be self-contained, we incorporated this confirmation of their Table I, as the last line of the comparison Table 1 below.

	Complexity	# of phases	Responsiveness
Tendermint-like view-changes: Casper [BG17], 2-phase Hotstuff [YMR+19a, §4.4], [MN23]	$O(n)$	2	✗
Hotstuff [YMR+19a]	$O(n)$	3	✓
Fast-HotStuff [JNFG20], Jolteon [GKS+22], BeeGees [ACG+23]	$O(n^2)$	2	✓
Marlin [SDZ22b]	$O(n)$	3	✓
{Fast-HotStuff [JNFG20] or Jolteon [GKS+22] or BeeGees [ACG+23]} **with lightweight PnS's**	$O(n)$	2	✓

Table 1: Comparison of leader-based BFT protocols under partial synchrony (excluding view synchronizer). The last line considers the modification of any of [JNFG20; GKS+22; ACG+23], consisting in replacing their view-change by plugging in efficient proofs of non-supermajority (PnS). A number of previous BFTs nevertheless enjoyed better parameters in so-called "happy" views. A view is happy if the leader is responsively reported a lock certificate formed in the *previous* view. In particular, a view cannot be happy if the leader of the previous one did nothing. Optimizations for "happy views" in previous works are: $O(n)$ complexity for [JNFG20; GKS+22], two phases for [SDZ22b], and responsiveness for [MN23].

We then bring further optimizations by introducing the following general tools:

(1) In Section 5 we introduce a simple and general optimization to BFTs, which applies to any *good view*. A good view is one in which there is no hidden lock ([ACG+23]), i.e., in which the leader is responsively reported the highest lock certificate ever formed. It removes the need for sending and verifying a PnS (be it in the naive form of a batch of k signed reports). The technical idea is to run *in parallel* a tentative Tendermint-like unlocking, and the detection of hidden locks. Previous optimizations either applied to a narrower set of views (called "happy": see Table 1), or required an extra round-trip in case of a hidden lock. More detailed analysis of previous works is provided in Section 5.2.

(2) A very simple compiler from *any* multisignature scheme, into an aggregate signature of a special-purpose type. It operates over tagged messages, where each key can sign at most one message with a given tag. When instantiated with BLS multi-signatures, its verification costs only two pairings for any k messages, vs $k+1$ pairings for [BGLS03]. In our use-case of aggregated messages with 6 variable bits, then this instantiation: (i) brings a $40\times$ speedup w.r.t. [BGLS03]; and (ii) removes the need for sequential signing compared to a related scheme of [LOS+13]. So it yields an alternative implementation of PnS, usable as a back-up in extremely bad scenarios of dozens of consecutive views without synchrony or an honest leader. Instantiating the compiler with the multisignature scheme of [RY07], enables to recover the scheme called Wendy in [GHA+21].

(3) We finally address two minor issues. The first is an observation made in [SDZ22b] that the verification cost of a multisignature with the latter instantiation, would be of $O(n)$ pairings. Hence, they count it as having $O(n)$ authenticator complexity. On the one hand, the verification cost is only of two pairings. On the other hand, it is true that the one-shot offline verification, inherited from the proof of possession (PoP) of [RY07], is $O(n)$ pairings. The second issue is that, surprisingly, no existing multi-signature is formally proven in the *multi-users* setting. Doing so would legitimate their usage as a k-out-of-n threshold signature, as considered in [DGNW20; GL22; DCX+23; GJM+23]. Precisely, $(n-t)$-users security allows the adversary to *choose* the target honest key, out of a set of $n-t$ honest keys. In Appendix C we address both issues simultaneously, using a PoP $40\times$ faster than the one of [RY07], although under the AGM.

Further comparison with related works is provided in Sections 3.2, 3.9, 5.2, 7.1 and 8 and appendix C.

2 Toy Model and Background

In this section we introduce a toy model, in which we will cast the simple consensus protocol described in Section 3 and called 2Pc. The purpose of 2Pc is to illustrate, as simply as possible, how to plug PnS into existing responsive view-changes in order to reduce their complexity. Also, 2Pc will be the toy baseline to illustrate our further techniques for achieving zero overhead in the good case (protocol $\mathsf{2Pc}^{\mathsf{0PnS}}$, in Section 5.1). The techniques which we illustrate in Section 3 can be directly used in more optimized end-to-end blockchain protocols [GAG+19; GKS+22; ACG+23], as explained in [ACG+23, Table I]. Hence, in Section 2.9 we list some extra features which we *purposedly not* included in this toy model, to preserve its simplicity.

2.1 Notations. The size of a finite set E is denoted $|E|$. The integers between a included and b excluded are denoted $[a,b[$, conversely for $]a,b]$, and $[a] := [1,\dots,a]$. The size of a bitstring b is denoted $|b|$, and the empty string $\perp$. $\mathbb{N} := \{0,1,2,\dots\}$ denotes the set of non-negative integers, of which the positive ones $\mathbb{N}^* := \{1,2,\dots\}$. Strings of characters are denoted in "quotes". To *multicast* a message means to send it to every player.

2.2 Partially-synchronous network and corruptions. We consider a set $\mathcal{P} = \{P_1,\dots,P_n\}$ of probabilistic polynomial time (PPT) machines connected by pairwise authenticated channels. We call them *players*. We consider a PPT machine, denoted as the *Environment* $\mathcal{E}$, which can read all messages sent and, without further specification, alter, reroute, delay or replay them. It can even suppress them (before GST: see below), whereas asynchronous BFTs [**bdt**; GKS+22] assume eventual delivery. $\mathcal{E}$ has full control of up to t players, where t is a parameter known as *corruption threshold*. We denote them as maliciously *corrupt*, also known as *Byzantine*.

For simplicity, we consider the maximal corruption threshold, i.e., we assume that the number of players is $n = 3t+1$ and that corruptions happen at the beginning of the execution. The remaining (at least $2t+1$) players are said to be *honest*.

From some point in time denoted as GST [DLS88], $\mathcal{E}$ commits to delivering all messages sent within a fixed delay δ. Both GST and δ are arbitrarily set by $\mathcal{E}$ in every execution, and are not disclosed to the honest players. However, there is a fixed upper bound $\Delta \geq \delta$ which holds for any execution, and which is publicly known in advance. In some executions, Δ may be much larger than the actual message delay δ. If GST $= 0$, then this means that the δ delay holds since the beginning of the execution. An event which is guaranteed to happen within a delay depending only on δ, not on Δ, is denoted as *responsive*.

2.3 Views and leaders. For simplicity, we assume a global clock that publicly ticks every Δ, starting at the time $t = 0$ when the execution begins. For simplicity we define *views* as the following consecutive timeframes. The first view, numbered as $v = 1$, is $[0, 3\Delta[$. Then every other view takes 4Δ, i.e., view $v \geqslant 2$ is $\big[3\Delta + (v-2)4\Delta\,,\, 3\Delta + (v-1).4\Delta\big]$. For every view number v, the clock publicly designates a player, denoted L_v, no later than at the end of the previous view $v-1$. It is called the *leader of view* v, it can possibly remain the same, i.e., $L_{v-1} = L_v$, or not. Nonwithstanding these simplifying assumptions, our protocols remain safe even if players do not receive clock ticks simultaneously, or are notified conflicting leaders.

2.4 *Non-interactive* threshold signature schemes (NI-TSS), in a generalized sense.

Definition 1 (NI-TSS). *A non-interactive* threshold signature scheme (TSS) in the general sense *(*NI-TSS*) consists of a setup phase, and of four locally computable algorithms* ($\textsc{Sign}$, $\textsc{Verify}$, $\textsc{aggregate}^{=}_{\bullet}$, $\textsc{Verify}^{=}_{k}$). *It is parametrized by a number N of participants, dubbed as* machines*, and an integer $k \leqslant N$ called the* threshold*. The setup phase publicly returns a public threshold key PK and one public* verification key pk_i *per machine, and, privately to each machine i, a secret signing key* sk_i.

- $\textsc{Sign}(\mathsf{sk}\,,\, m \in \{0,1\}^*) \to \sigma$ *we will sometimes dub the output a signature* share*;*
- $\textsc{Verify}(\mathsf{pk}, m, \sigma) \to \{\mathsf{accept}$ *or* $\mathsf{reject}\}$*. It the output is* accept*, then we say that σ is a* valid *signature (share) on the data m.*
- $\textsc{aggregate}^{=}_{\mathcal{I}}\,(m, (\sigma_i)_{i\in\mathcal{I}}) \to \sigma$ *is a (possibly randomized) aggregation algorithm. It takes as input k signatures $(\sigma_i)_{i\in\mathcal{I}}$ for a k-subset $\mathcal{I} \subset [N]$ and a message $m \in \{0,1\}^*$ (the superscript $=$ is to stress that the message content m is the same for all signers in $\mathcal{I}$.) We will sometimes dub the output a* threshold *signature;*
- $\textsc{Verify}^{=}_{k}(PK, m, \sigma) \to \{\mathsf{reject}$ *or* $\mathsf{accept}\}$*. It the output is* accept*, we say that σ is a* valid *threshold signature on m.*

They must furthermore satisfy the following conditions.

Definition 2 (Correctness & robustness). *For every execution of the setup, then*

(Correctness-of-signature-shares) $\qquad \textsc{Verify}\,(\mathsf{pk}_i, m, \textsc{Sign}(\mathsf{sk}_i, m)) = \mathsf{accept}\ \ \forall i \in [N]\ ;\ \forall m \in \{0,1\}^*.$

The next condition, called robustness-of-aggregation, states that any k valid signature shares always aggregate into a valid threshold signature. It is stronger than standalone correctness-of-aggregation, since the latter applies only to correctly formed signature shares.

(Robustness of aggregation)
$$\textsc{Verify}^{=}_{k}\Big((\mathrm{PK}, m, \textsc{aggregate}^{=}_{\mathcal{I}}\,\big(m, (\sigma_i)_{i\in\mathcal{I}}\big)\Big) = \mathsf{accept}$$
$$\forall \mathcal{I} \subset \{1,\dots,N\}\ ,\ \forall m \in \{0,1\}^*\ ,\ \forall (\sigma_i)_{i\,\in\,\mathcal{I}}\ \textit{s.t.}\ \textsc{Verify}(\sigma_i) = \mathsf{accept}\ \forall i \in \mathcal{I}$$

Definition 3 (TS-UF-1 Unforgeability). *Is defined in [BCK+22]. Informally, it states that if an adversary corrupting a number t of machines, $t < k$, can forge a valid threshold signature on some m, then it must be that the adversary previously observed signature shares on m issued by at least $k - t$ honest machines.*

The BLS threshold signature scheme [Bol03, §3] satisfies Definition 1. It requires a setup which securely assigns, to each machine, its secret key equal to a k-out-of N secret share of a randomly sampled common secret signing key. To interactively establish this setup in our use-case of a *high threshold* $k = 2t+1$, the state of the art asynchronous distributed key generation (ADKG) is [DXKR23]. The $\textsc{aggregate}^{=}_{k}$ algorithm has quadratic complexity, since it requires a Lagrange interpolation of a degree $k+1$ polynomial. However its practical cost was recently optimized ([Rob20] "aggregate a 130,000 out of 260,000 BLS threshold signature in just 6 seconds (down from 30 minutes)"). The threshold signature output has the same format as a standalone BLS signature, in particular its verification costs only 2 pairings. Notice that TS-UF-1 unforgability for threshold BLS was shown only very recently [Gro21; BCK+22] (and its robustness is claimed in the latter). So these works have legitimated prior uses of threshold BLS in BFT. Before these works, it was not excluded that an adversary could possibly forge a threshold signature as soon as it would observe one signature share issued by *one* honest machine.

2.5 Instantiation of NI-TSS from multi-signatures. The so-called non-interactive *multi-signature schemes* [RY07; BDN18; DGNW20; KCLM22; FSZ22; BCG+23] can be bootstrapped in our generalized Definition 1 of NI-TSS. Their setups do not require interaction between participating machines. They do not either require that the subset $\mathcal{I}$ of signers interact in a preliminary round in which they would all behave honestly. Such requirements are made in [PW23; TZ23], which are thus incompatible with the definition of NI-TSS. For simplicity of the exposition we restrict the definition of multisignatures to our use case, where the list of the N potentially participating machines is pre-determined. In this context, each subset $\mathcal{I}$ of k-out-of-N machines is efficiently encoded as an N-sized binary array. We will dub *multi-signature* the output of $\textsc{aggregate}^{=}_{\mathcal{I}}\big((\sigma_i)_{i\in\mathcal{I}}, m \in \{0,1\}^*\big)$. The setup of these schemes is that each machine locally generates a key pair $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \textsc{KeyGen}()$, along possibly with extra data called a *proof of possession* (PoP) [RY07]. Then it publishes pk_i on a bulletin board, along with its PoP if required. Before aggregators and verifiers can use these keys, they need to perform some extra local computation (verification of the PoP's and/or scalar multiplication of keys in [BDN18; BCG+23]). We will benchmark these computations in Appendix C. To make it simple, we define the threshold key as the list of the N published keys $\mathrm{PK} = [\mathsf{pk}_1, \ldots, \mathsf{pk}_N]$. Verifiers simply consider as $\perp$ the published keys which are not appended with a *valid* PoP. Hence, without loss of generality, we assume from now on that all published keys are appended with a valid PoP. Notice that this restricted context, of a set of N fixed public keys, is also the one considered in recent works on NI-TSS based on BLS [ACR21; BCG+23; DCX+23; GJM+23]. In multisignature schemes, $\textsc{Verify}^{=}_{k}$ also takes as input the k-subset $\mathcal{I} \subset [N]$ of signers:

- $\textsc{Verify}^{=}_{k}(PK, \mathcal{I}, m, \sigma) \rightarrow \{\mathsf{reject}\text{ or }\mathsf{accept}\}$

So to bootstrap this syntax into the one of NI-TSS, we may just consider that the multisignature output by $\textsc{aggregate}^{=}_{\mathcal{I}}$ comes appended with the N-sized bit array $\mathcal{I}$. The pairing-based multi-signature schemes [Bol03, §4][RY07; DGNW20; BCG+23] have the computational benefit that their aggregation cost is *linear* in k. It simply consists in the addition of k points, which is in practice highly amortized for large k. Notice that their robustness follows trivially from the additivity of their verification formulas. The verification complexity of the schemes [Bol03, §4][RY07; BDN18; BCG+23] is of only *two* pairings (plus two additions of k points, more precisely two linear combinations of k points in [BDN18]), so is close to the one of a single BLS signature [Ben04; Dar10]. Finally, one last condition for multisignatures to implement NI-TSS, is that they satisfy unforgeability in the $(N-t)$*-users setting*. This is formalized by [Lac18] for the related notion of aggregate signatures. Let us just summarize that it consists in a security game where the adversary is given, not only one target key, but a *list* of the possible target keys, i.e., the $N-t$ ones of the honest machines among $[\mathsf{pk}_1, \ldots, \mathsf{pk}_N]$. It wins as soon as it forges a multisignature which is valid with respect to a set $\mathcal{I}$ containing one of the target keys pk^* (and provided it did not query pk^* for a signature on the forgery message). As observed in (4) of the introduction, we are not aware of any existing formal security reduction for the multi-users security of multisignatures (reductions are done in [Lac18] for the related problem of aggregate signatures over *pairwise distinct* messages). We will provide one in Appendix C, for an efficient PoP which we introduce. We leave for future work the analysis of other existing multisignature schemes in the multi-users setting, for which we have good hope that their security loss is not higher than the number of honest machines.

2.6 Aggregate signature schemes: over *different* message contents. A *non-interactive* aggregate signature scheme [BGLS03] consists of a setup, in which each machine locally generates a pair of keys $(\mathsf{sk}_i, \mathsf{pk}_i)$, and, in our context, publishes pk_i on a public bulletin board, and of four algorithms. The first two, $\textsc{Sign}$ and $\textsc{Verify}$, have the same syntax as in Definition 1. The last two are as follows.

- $\textsc{aggregate}^{\neq}_{\mathcal{I}}\big((m_i, \sigma_i)_{i\,\in\,\mathcal{I}}\big)$ takes as input a list of indices of keys $\mathcal{I}$ (possibly with repetitions, although we will not use this feature), and a list of message-signature pairs: $(m_i, \sigma_i)_{i\,\in\,\mathcal{I}}$ where each σ_i is valid over m_i for pk_i. The superscript $\neq$ stresses that the contents $(m_i)_i \in \mathcal{I}$ can now be *different*.
- $\textsc{Verify}^{\neq}\big([\mathsf{pk}_1, \ldots, \mathsf{pk}_N], \mathcal{I}, (m_i)_{i\,\in\,\mathcal{I}}, \sigma\big)$ outputs either reject or accept. In the latter case we say that σ is a valid aggregate signature on the $(m_i)_{i\,\in\,\mathcal{I}}$ with respect to $\mathcal{I}$.

They have the expected properties. Namely, $\textsc{aggregate}^{\neq}_{\bullet}$ over valid inputs returns a valid aggregate signature. Then, the $(N-t)$-users unforgeability [Lac18] states that any polynomial adversary given $N-t$ honestly generated keys and oracle access to their signatures, cannot possibly forge a valid aggregate signature involving a message m_j credited to one of the honest keys pk_j, unless it has already made signature query on m_j to pk_j.

The scheme of [BGLS03] furthermore requires that the contents of the aggregated messages are different. The verification complexity of [BGLS03; BNN07] of an aggregate signature over k messages costs $k+1$ pairings. This is roughly the cost of verifying $k/2$ individual BLS signatures, so we count an aggregate signature as $k/2$ authenticators. We will confirm empirically this complexity gap in Section 8.1. We will present an optimization of [BGLS03] in Appendix D, possibly folklore. We present another way to build aggregate signatures in Section 7, and benchmark it with the previous ones in Section 8.

2.7 BFT. Let us consider any public efficiently computable predicate $\mathsf{ExtValid} : \{0,1\}^* \to \{\mathsf{accept}, \mathsf{reject}\}$. A value which is returned as accept is said to be *externally-valid* ([AMS19]). We denote by $\mathcal{X} := \left\{x \in \{0,1\}^* \mid \mathsf{ExtValid}(x) = \mathsf{true}\right\}$ the set of *externally-valid* values. We assume that each player starts the protocol with a valid input.

Definition 4 (BFT). A *partially synchronous leader-based Byzantine fault-tolerant consensus with external validity*, or *BFT* for short, is a protocol in which each player outputs at most one value: we say that the player commits the value, and such that:

Consistency: no two players decide different values;
External Validity: if a player commits x, then $x \in \mathcal{X}$;
Eventual Termination: every player commits in any infinite execution in which there is a view v which starts at, or after, GST, and in which the leader L_v is honest.

2.8 Latency and authenticator complexity. One authenticator is, concretely, a data which has comparable size and verification complexity as a MAC or a standalone signature. Both the BFT protocols 2Pc and $\mathsf{2Pc}^{\mathsf{0PnS}}$ presented in Section 3 enjoy the following:

Theorem 5 (Responsive view-change and two-phase latency). *Consider the first time after GST where a view v starts with a leader L_v which is honest. Then a value becomes* committed *(by L_v) within 5δ.*

More particularly, they enjoy some form of *quality* ([AMS19]) since, if started at GST with an honest leader, then the committed value is its input. Our main contribution is that, when instantiated with any of the implementations of PnS proposed in Sections 4, 6.1 and 7, then 2Pc and $\mathsf{2Pc}^{\mathsf{0PnS}}$ have *linear authenticator complexity per view*. This means that the total number of authenticators sent and verified by honest players in each view is *linear in* n, i.e., is $O(n)$.

2.9 What is *not* in this toy model The following techniques are well-investigated, and can be directly plugged in our toy protocols (i) how to implement the global clock and/or the leader designation, and make them advance at the actual pace of the system (in particular, in views with synchrony and an honest leader, *not* wait for 4Δ to elapse before moving to the next view) [BCG22b; NK20; CGK+22; BCG22a; LL22; LA23; MN23; EEK+23] (ii) how to pipeline a new instance of consensus after every other phase, possibly by replacing the leader (leaders in 2Pc are *stateless*, so they *could* be replaced after speaking once, in the sense called LSO in [ACG+23]); and what is then the validity condition on values (roughly: being a batch of transactions extending a block of previous height in a lock certificate) [BG17; YMR+19a; ACG+23] (iv) *if* leaders rotate every other phase, then how to preserve liveness despite non-consecutive honest leaders [ACG+23] (v) how possibly external *learners* can read the state machine ([Lam06; GV10]): this can be conveyed by the outputs produced by 2Pc. For the sake of simplicity and concreteness, we define in a very low-level manner these outputs as *decision certificates*.

3 Toy Two-Phase Consensus from any PnS

In this section we illustrate our generic black-box tool for responsive view change in two-phase consensus. In Section 3.1 we convey the main idea of how to do a responsive-yet-safe view-change. In particular, we give the intuition of our new primitive, called a *proof of non-supermajority system* (PnS), and where it can be used in existing BFTs to

achieve responsiveness in two-phases with linear communication complexity. In Section 3.2 we describe how existing works fit in the framework of PnS's. Then in Section 3.3 we formalize PnS. In Section 3.4 we illustrate the application of PnS to view-change, by describing a simple single-instance consensus, called 2Pc. We refer to Section 2 for the long list of simplifications which we purposedly made to 2Pc, in order to keep the simplicity of this toy-model. Further details are provided in Appendix A.

3.1 Overview and roadmap. Very concretely, to commit a value x, there must be $2t+1$ signed so-called dec vote's which are cast by $2t+1$ players, for x, in a given view v'. The threshold signature out of these votes forms what we call a *decision certificate for* x. Any player receiving a decision certificate for a value x, commits x. Also, a decision certificate can be forwarded to any entity external to the system (formalized as a "learner of the state machine" in [Lam06; GV10]), to prove to it that x was committed. When a new view v starts, players send to the new leader, L_v, messages of a specific format called new-view. Roughly, in its new-view message, each player, P_i, testifies of the most recent view number, $v_i < v$ in which it cast a commit vote . We call v_i its *locked view number*, and, denoting as $\widetilde{x_i}$ the value for which it voted, we say that P_i is *locked* on $\widetilde{x_i}$. So far, all this dates back to at least [DLS88] and Paxos [Lam98], and was inherited in most recent leader-based BFTs to our knowledge.

The leader of the new view, L_v, waits until it receives a set NV_{set} of messages, from $2t+1$ players. Let $v_{\max} < v$ be the highest locked view number reported in one of those new-view messages, and $x_{\max}$ be the value for which dec vote's were cast in $v_{\max}$. How does the leader know for sure that $x_{\max}$ is the *unique* value for which honest players could have cast a commit vote in $v_{\max}$? It is because it is also reported a quorum of preliminary votes for $x_{\max}$ in $v_{\max}$, which we will call a *lock certificate*, shortened as lock cert$\big[v_{\max}, x_{\max}\big]$. This quorum of preliminary votes, called lock vote s, ensure the unicity. The leader sends a proposal to players to vote for $x_{\max}$. To convince them that they can safely vote for $x_{\max}$, it appends to its proposal a proof for the following predicate called (1):

$$\text{safe}(v, x_{\max}) : \text{No other value than, possibly, } x_{\max}\text{, could have been committed in any earlier view } v' < v. \tag{1}$$

By this we mean precisely that, if $2t+1$ commit vote messages (a "supermajority") were cast in some earlier view $v' < v$, then these must have been votes for $x_{\max}$.

To prove this predicate safe$(v, x_{\max})$, the leader appends two evidences to its proposal. Each of them proves one half of the predicate. The first evidence is the lock certificate for $x_{\max}$. It proves that, *in no previous view* $v_1 \leqslant v_{\max}$, no other value than $x_{\max}$ could have been committed. The assertion is clear for $v_1 = v_{\max}$, since a lock certificate is made of a supermajority of votes in $v_{\max}$ on $x_{\max}$ (and since honest players do not vote for conflicting values in the same view). Then for the lower views $v_1 < v_{\max}$, we will see in Section A.1 that the assertion follows from a very nice and old induction argument that, for all previous views, votes were cast for values respecting Equation (1). This dates back to at least PBFT [CL99] (which is itself the "byzantization" of Paxos [Lam11]). The second evidence is brought by what we call a PnS$(v, v_{\max})$. It is a bitstring (hopefully short) which proves to any verifier, possibly external, the following predicate: *no value could possibly have been committed in any higher view* v'*, s.t.* $v_{\max} < v' < v$. In conclusion, from these two evidences, players are convinced of Equation (1). From there, the leader can responsively drive the view to the committing of $v_{\max}$, following the classical pattern of two-phases of votes.

Our contribution in this work is to enable the leader to efficiently prove the following remaining half of the predicate. It states that no value could have been committed in any view higher than $v_{\max}$, so is implied by the following.

$$\text{non-Supermajority}(v, v_{\max}) : \text{no supermajority of dec vote's, i.e., } 2t+1\text{, could have been cast in any higher view } v'\text{, s.t. } v_{\max} < v' < v\ . \tag{2}$$

In turn, it is straightforward to see that this non-Supermajority$(v, v_{\max})$ predicate is implied by the sufficient predicate that, *no set of* $t+1$ *honest players could have cast* dec vote*'s in any higher view* v'*, s.t.* $v_{\max} < v' < v$. This is this actually this last sufficient predicate which will be proven by what we call a PnS$(v, v_{\max})$. [A historical remark is that the previous straightforward implication was somewhat formalized in PBFT [CL99] (if not already in Paxos). Precisely, this implication can be seen as the contraposite of their statement on top of page 5 ($(t+1)$"committed-local(v', x)" $\Rightarrow$ "committed(v', x)") Indeed, since an honest player issues only one lock vote$\big[v', x\big]$,

it cannot be the case that a lock vote$[v', y]$ be formed for any conflicting value $y \neq x$, and hence no commit cert$[v', y]$ can ever be formed.]

3.2 Relationship to prior protocols. Nearly all prior two-phase BFTs with responsive view change [CL02; GAG+19; GKS+22; ACG+23] implicitly implemented a PnS, without actually defining it, as the concatenation of $2t+1$ signed testimonies. There, each of their issuers, say, P_i, testified that it does not have a higher lock certificate than the reported v_i. Since this concatenation is multicast by the leader, the communication cost is quadratic in n. Notice that the original PBFT [CL99] was even heavier, since the leader included, in the concatenation, all the reported lock certificates. This inefficiency was observed in the v2 of [YMR+19a], in §9.2, under the name "Vanilla" [and reappeared in [GKS+22], where this concatenation is called a TC]. In order to reduce the size of the signatures, it was proposed in [JNFG20] that the leader AGGREGATES$^{=}_{t}$ he signatures of the testimonies into one single BGLS aggregate signature [BGLS03]. Since verification of one aggregate signature over k messages costs $k+1$ pairings, this has the same order of magnitude as verifying k distinct BLS signatures. This is why we count this as a linear authenticator complexity per player, hence, a total *quadratic authenticator complexity*. We compare more concretely to their performance in Section 7.

3.3 Proofs of non-Supermajority (PnS). To enable the leader to efficiently build a short proof for the non-Supermajority predicate (Equation (2)), vouching for the lock certificate which it will propose, players and the leader use the following new threshold primitive. We formalize it in Definition 6, and call it a *Proof of non-Supermajority (PnS) system*. We now illustrate the definition by describing how it is used in a BFT. For each player P_i at the beginning of a new view v, denote as $v_i \in [0, v-1]$ its locked view number ($v_i = 0$ is for when P_i has no lock). In its new-view message to the leader L_v, P_i includes a somewhat signed testimony, stating that its locked view number at the beginning of v is equal to v_i. Somehow, it appends to this testimony extra *hints*, which will help L_v to efficiently aggregate many testimonies into a single proof. More formally, P_i creates the testimony and the hints *all-at-once*, in the form of what we call a report: $\text{report}_{i,v,v_i} \leftarrow \text{PnS.Report}_i(v, v_i)$. Upon receiving $2t+1$ valid reports for view v: $(\text{report}_{i,v,v_i})_{i \in I}$ indexed by some subset $I \subset [n]$, the leader somehow extracts from them a PnS$(v, v_{\max})$, where $v_{\max}$ is the max of the reported v_i's. Precisely, it obtains the PnS by applying, on the batch of $2t+1$ reports, the (publicly available) algorithm called PnS.Report. Finally, upon receiving from the leader a request to vote on some value $\widetilde{x}$, encapsulated in some lock certificate of rank $\widetilde{v}$, and vouched by a purported proof of non supermajority: π, each player applies the verification algorithm PnS.Verify (v,$v_{\max}$,π), and if this check passes, this means that π is a valid PnS$(v, v_{\max})$.

Two questions which may arize from the specification of PnS are (A) what prevent corrupt players from reporting very high inputs, e.g., $v_i = v-1$? Nothing. However, the way in which we *use* a PnS system in BFT is that a honest leader does not take into account reports unless they come appended with a lock certificate for the corresponding epoch. That way, when the leader outputs a PnS$(v, v_{\max})$, it is guaranteed to have a matching corresponding lock cert$[v_{\max}, x_{\max}]$. Also, (B) what prevents a corrupt leader from taking into account a report for a high input, e.g., $v_{\max}$, without checking existence of a lock cert$[v_{\max}, \bullet]$? Again, nothing. This will only *weaken* what it proves, since the higher the $v_{\max}$, the weaker the PnS$(v, v_{\max})$.

Definition 6 (PnS: definition as a threshold cryptographic primitive). Consider $n = 3t+1$ players, of which t are corrupt. All what follows takes as parameter any arbitrary bitstring, denoted v. It can be could be thought of as a unique instance identifier (in our context: the current view number v, implicitely appended with the consensus instance identifier). Each honest player P_i starts with a tuple (v, v_i), where $v_i \geqslant 1$ is a positive integer in some predetermined range (in our context: $v_i \in [0, v-1]$ is the *locked view number* of P_i at the beginning of view v). A *PnS system* is the data of a *setup*, and of a quadruple of locally computable algorithms, which are available after this setup: (PnS.Report, $(\text{PnS.Prove}_i)_{i \in [n]}$, PnS.ReportVerif, PnS.Verify). The outcome of the *setup* is the publication of some data on a publicly available bulletin board, e.g., (individual or threshold) verification key(s), and a private assignment to each player P_i, e.g., signature key(s).

- **PnS.Verify$(v, v_{\max}, \pi)$**. This publicly available algorithm takes as input an integer $v_{\max}$ and a bitstring π, and outputs either accept or reject. If it outputs accept, then we call π a *Proof of non-supermajority for* $v_{\max}$

in view v, shortened as a PnS $(v,v_{\max})$. We require that: existence of a PnS $(v,v_{\max})$ *implies* that no set of $t+1$ honest players have their inputs (v, v_i) such that $v_{\max} < v_i$.
[So this implies the *non-supermajority predicate* for $v_{\max}$ (Equation (2)). This requirement is dubbed as the *soundness condition.*]

- $\mathbf{report_{i,v,v_i}} \leftarrow \mathbf{PnS.Report_i(v, v_i)}$ for all $i \in [n]$. This algorithm is accessible only by player P_i, since it requires secret input(s) from P_i (left implicit), e.g., signature key(s). It outputs a data string called a report.
- $\mathbf{PnS.ReportVerif}\big(\boldsymbol{i, v, v_i, report_{i,v,v_i}}\big)$. It is a publicly available algorithm, it returns either accept or reject. If the outcome is accept, then we say that $\mathrm{report}_{i,v,v_i}$ is a *valid* report *of* P_i *on* (v, v_i). We require the condition, dubbed as *report-completeness*, that: $\forall i \in [n]$, $\mathrm{PnS.Report}_i$ always outputs valid reports.
- $\boldsymbol{\pi} \leftarrow \mathbf{PnS.Prove}\Big(\boldsymbol{v, I,} \big\{\boldsymbol{v_i}\,,\, \mathbf{report_{i,v,v_i}}\big\}_{\boldsymbol{i \in I}}\Big)$. This publicly available algorithm takes as input a set of $2t+1$ pairs of $\big\{$reported input, valid report$\big\}$, indexed by $I \subset [n]$ (all with respect to the same (current view number) v). Denote the highest reported input as $v_{\max} = \max_{i \in I} v_i$. We require the condition, dubbed as *prover-completeness*, that the output π is a $\mathsf{PnS}(v, v_{\max})$.

Remark (Generalized parameters). The specific thresholds in Definition 6: of t-out-of $n = 3t+1$ corruptions, and of the *non-supermajority predicate* stated for $t+1$ honest players, can seamlessly be generalized to any threshold. It would then give: consider any number n of players of which t are corrupt, and k a number such that $t + k \leqslant n$. Then $\mathrm{PnS.Prove}$ would take as input $k + t$ reports messages, and the supermajority predicate would then state that at least k honest players have their input lower than $v_{\max}$ (so no set of $n - t - k + 1$ honest players can have their input higher).

3.4 Introducing the toy two-phase BFT from any PnS. We now illustrate the use of PnS on a simple one-instance leader based consensus. This toy protocol is called *two-phase consensus* (2Pc) and formalized in Algorithm 2. Further formalism for the data structures is in Appendix A Fig. 5. We now convey the main ideas of 2Pc. It can be seen as *one* instance of SBFT [GAG+19] (no chain of instances), without a fast-track, and where the costly multicast of $2t+1$ reports was replaced by a black-box PnS system (see at the end of Section 3.1). Although simple, when 2Pc is instantiated with a PnS having constant authenticator complexity, then it yields *the first two-phase leader based consensus with responsive view change and linear communication complexity.* The steps of 2Pc are specified with respect to the current view number v. Players perform the steps numbered 0., 2. and 4., as soon as they can, in any order. Some additional steps, numbered as 1., 3. and 5., are taken only by the leader.

3.5 Steps common to every view v. The broad pattern of the following steps has been mainstream since at least [CL99], and is commonly known as *BFT with two phases of vote.* The first phase is initiated in step **1.** by the leader of the view, L_v, which proposes a valid value, x, to all players. Technically, it does so by multicasting a message of a specific format, called prepare , to be detailed. In step **2.**, upon receiving from L_v for the first time a prepare message for some value x each player accepts it if the prepare satisfies some conditions to be specified. If it accepts, then, it sends to the leader L_v a signed vote for the value x. It does so in the form of a signed message of a specific format, denoted as a lock vote$\big[v, x\big]$. In step **3.**, upon receiving $2t+1$ lock vote$\big[v, x\big]$ messages for the same value x, the leader AGGREGATES$^{=}_{2t+1}$ their signatures (shares) into what we call a *lock certificate attached to view number* v *for the value* x, and which we denote as lock cert$\big[v, x\big]$. Notice that the $t < n/3$ assumption implies that no two lock certificates attached to the same view number v can ever exist for two distinct values $x \neq x'$. This partly explains the name "lock". The leader L_v initiates the second phase of vote by multicasting the lock cert$\big[v, x\big]$. In step **4.**, upon receiving a lock cert$\big[v, x\big]$, each player sends to the leader L_v, a signed vote for the value x, in the form of a message called *decision vote*, and denoted as commit vote$\big[v, x\big]$. In step **5.**, upon receiving $2t+1$ commit vote$\big[v, x\big]$ messages for the same value x, the L_v AGGREGATES$^{=}_{2t+1}$ their signatures (shares) into what we call a decision certificate, denoted as commit cert$\big[v, x\big]$, then commits x. Any honest player which forms or receives a decision certificate for a value x must automatically commit x.

Remark. Our contribution is orthogonal to how decision certificate(s) are diffused once a value has been committed. Several well-known implementation options are possible. The most natural one is that L_v muticasts the decision

certificate to all. In the pipelined regime [YMR+19a; ACG+23], players send instead their commit vote$\big[v, x\big]$ to the next leader L_{v+1}, which then multicasts the decision certificate along with a new block proposal.

More accurately, following the terminology of [CL99]: we say that the predicate committed(v, x) holds for some value x and view number v, if there exists a set $\mathcal{D}$ of at least $t+1$ honest players which received a lock cert$\big[v, x\big]$ while they were in view v. Indeed, the view-change mechanism will guarantee that, from this point in the execution, only the value x can make its way into a decision certificate. Conversely, we will use in the proof of safety the easy aforementioned implication that, if some commit cert$\big[v, x\big]$ is formed, then committed(v, x) must hold.

3.6 View-change for higher views $v \geqslant 2$: locked values. We now detail the mechanism enabling the leader to choose the value that it proposes to players in step **1.**, and preventing two distinct values from being decided in two distinct views, thereby ensuring the "consistency" guarantee. This mechanism is known as *view-change* and shows up in steps **0.** to **2.**

No view-change however needs to be done in the first view, where the leader L_1 can simply send a blank prepare for any valid value x, e.g., its own input. To make the view-change operate, 2Pc requires that every player P_i locally stores and updates a lock certificate, denoted as its "high lock$_i$". To define it, we introduce the terminology that a lock certificate is "higher" than another one, if it is attached to a higher view number. Then, high lock$_i$ is simply defined as the highest lock certificate that P_i so far ever received or created. Notice that this mechanism is present in most BFTs since [DLS88]. For consistency of notations, we make the convention that a player P_i starting the BFT instance initializes high lock$_i$ $:= (v_i := 0, \widetilde{v}_{P_i})$, where $\widetilde{v}_{P_i}$ is any valid value of its choice, e.g., its input to the BFT instance. Let us just examplify, for concreteness. When 2Pc is used in the SMR regime of pipelined consecutive instances, then x_{P_i} can be any block of pending transactions with ancestor equal to a block of previous height and vouched by a quorum certificate, i.e., in our terminology: either a lock certificate or a decision certificate.

If, later, high lock$_i$ becomes equal to some lock cert$\big[v_i, \widetilde{x_i}\big]$, for some $v_i \geqslant 1$, then we say that P_i is "locked" on the value $\widetilde{x_i}$. Otherwise if v_i is still 0 then we say that P_i is not locked on any value. This is why we call lock cert$\big[v_i := 0, \widetilde{v}_{P_i}\big]$ a *non-locking* lock certificate (it can be seen as a mere wrapper of the value $\widetilde{v}_{P_i}$).

3.7 View-change for $v \geqslant 2$: choice of the value to prepare. At the beginning of a view v, in step **0.**, every player P_i sends its high lock$_i$ to the leader L_v, in a message of a specific format denoted as new-view. In addition to this high lock$_i$, a new-view message contains a report of player P_i (Equation (report) in Algorithm 2). Recall that this is, roughly, a signed testimony of P_i stating that it has locked view number equal to v_i at the beginning of v. The leader waits until it receives a set of $2t+1$ new-view messages:
$\mathrm{NV}_{\text{set}} = \Big\{\big(\text{"new-view"}, \mathrm{report}_{i,v,v_i}, \text{lock cert}\big[v_i, \widetilde{x}\big]\big) \;:\; i \in I\Big\}$ where $I \subset [n]$ the $2t+1$-sized subset of indices of the issuers. Denote as lock cert$\big[v_{\max}, x_{\max}\big]$ the high lock$_{L_v}$ of the leader L_v after reception of these messages. By definition, since the leader L_v must update it high lock$_{L_v}$ every time it receives a higher one, we have that $v_{\max} \geqslant \max_{i \in I} v_i$.

The leader L_v extracts, out of the reports contained in the messages, a *proof of non-supermajority for $v_{\max}$ in view v* (PnS$(v, v_{\max})$), denoted as π, as spelled-out in Algorithm 2 Equation (proof). Then L_v proposes to the players to vote for $v_{\max}$, in step **1.**. It does so by multicasting a message of a specific format: ("prepare", π, lock cert$\big[v_{\max}, x\big]$). Recall that if $v_{\max} = 0$, then we have that lock cert$\big[v_{\max}, x\big] = x$ is any externally-valid value x of L_v's choice, e.g., its input in the BFT instance.

One possible optimization in [**1.** Prepare] is that the leader L_v tentatively forms the PnS$(v, v_{\max})$ without verifying the reports. It verifies them a posteriori if the PnS is invalid. In this case, it publicly exposes the invalid report(s), thereby evidencing misbehavior of their issuer(s). Then it waits for further reports to replace the invalid one(s). This idea is further optimized in Section 4.1 in the context of the implementation called $\mathcal{T}$-PnS.

3.8 View-change for $v \geqslant 2$: accepting a prepare message. Upon receiving a ("prepare", π, lock cert$\big[\widetilde{v}, x\big]$) message, each player P_i in view v, in the case where lock cert$\big[\widetilde{v}, x\big]$ would be strictly higher than its own high lock$_i$ = lock cert$\big[v_i, \widetilde{x_i}\big]$, then by definition it immediately updates high lock$_i \leftarrow$ lock cert$\big[\widetilde{v}, x\big]$. In particular it becomes locked on x, if it was not already. [Notice that in case of equality $v_i = \widetilde{v} \geqslant 1$, since no two different values can

2Pc

The steps 0., 2., 4. are done by each player when it is in view v.
In addition the leader L_v performs the steps 1., 3., 5..
A player starts view $v = 1$ at the beginning of the protocol, then view $v = 2$ after 3Δ, then each next view after 4Δ.

0. Report If $v = 1$, skip and go directly to step **1**.
Else if $v \geq 2$: Every player P_i, denoting by $\mathsf{lock\,cert}\big[v_i, \widetilde{x}_i\big]$ its $\mathsf{high\,lock}_i$, forms a report:

(report) $$\mathrm{report}_{i,v,v_i} \leftarrow \mathrm{PnS.Report}_i(v, v_i).$$

Then it sends $\big(\text{“new-view”}, \mathrm{report}_{i,v,v_i}, \mathsf{lock\,cert}\big[v_i, \widetilde{x}\big]\big)$ to the leader L_v;

1. Prepare *If* $v = 1$, Leader L_1 multicasts $(\text{“prepare”}, \mathsf{lock\,cert}\big[0, x_{L_1}\big] = x_{L_1})$, where x_{L_1} is any valid value //e.g., its input to the BFT instance.
Else if $v \geq 2$: Leader L_v waits until it receives a set NV_{set} of $2t+1$ valid new-view messages, indexed by some $I \subset [n]$: $\mathrm{NV}_{\text{set}} = \Big\{\big(\text{“new-view”}, \mathrm{report}_{i,v,v_i}, \mathsf{lock\,cert}\big[v_i, \widetilde{x}\big]\big) \ : \ i \in I\Big\}$.
Denote by $\mathsf{lock\,cert}\big[v_{\max}, x_{\max}\big]$ the $\mathsf{high\,lock}_{L_v}$ of L_v //since it must update it every time it receives a higher one, we have in particular: $v_{\max} \geqslant \max_{i \in I} v_i$.
It extracts, out of NV_{set}, a *proof of non-supermajority for* $v_{\max}$ *in view* v:

(proof) $$\pi \leftarrow \mathrm{PnS.Prove}\big(\{\mathrm{report}_{i,v,v_i} \ : \ i \in I\}\big).$$

//since a $\mathsf{PnS}(v, \max_{i \in I} v_i)$ is *a fortiori* a $\mathsf{PnS}(v, v_{\max})$.
Then it multicasts $(\text{“prepare”}, \pi, \mathsf{lock\,cert}\big[v_{\max}, x_{\max}\big])$.

2 Lock Vote Every player P_i waits until it receives a prepare message for the first time from L_v:

$$\big(\text{“prepare”},\ \pi,\ \mathsf{lock\,cert}\big[\widetilde{v}, \widetilde{x}\big]\big)$$ //recall that, for the message to be valid, π must be a $\mathsf{PnS}(v, \widetilde{v})$.

Then P_i replies with a $\mathsf{lock\,vote}\big[v, \widetilde{x}\big]$.

3. Lock Certificate Upon receiving $\mathsf{lock\,vote}\big[v, x\big]$ from $2t+1$ distinct issuers, the leader L_v AGGREGATES$^{=}_{2t+1}$ their signatures to form a $\mathsf{lock\,cert}\big[v, x\big]$, which it multicasts //in the context of chained SMR, it may also multicast a new prepare for a block extending v.

4. Commit Vote Upon receiving a $\mathsf{lock\,cert}\big[v, x\big]$ from leader L_v, a player P_i replies with a signed $\mathsf{commit\,vote}\big[v, x\big]$ //In the pipelined regime, it sends it instead to the next leader.

5. Commit Certificate Leader L_v, upon receiving $2t+1$ $\mathsf{commit\,vote}\big[v, x\big]$ from distinct issuers, AGGREGATES$^{=}_{2t+1}$ the signatures into a $\mathsf{commit\,cert}\big[v, x\big]$. //In the pipelined regime then this is performed by the next leader. Upon forming or receiving a $\mathsf{commit\,cert}\big[v', x\big]$ for any $v' \in \mathbb{N}^*$, a player commits x (and continues the protocol).

Algorithm 2: **2Pc**: generic two-phase consensus with responsive view change, instantiated from *any* black box PnS system.

have a lock certificate attached to the same view $\geqslant 1$, it must be the case that $x = \widetilde{x}_i$.] Then, P_i accepts the $\mathsf{prepare}\big(v, \mathsf{lock\,cert}\big[\widetilde{v}, x\big]\big)$ message *if and only if* π is a valid $\mathsf{PnS}(v, \widetilde{v})$. The rest of the view follows the classical pattern of PBFT, except that votes are collected then aggregated by the leader, in order to preserve linear communication complexity. The proof of safety is formalized in Section A.1.

3.9 Why the view-change is responsive (and where it was prevented). So in particular we see that, *even if* P_i's locked view number $v_i > \widetilde{v}$ was strictly higher, it *still* accepts the prepare message. This is the key to responsive view-change, as is the case in [CL99; GAG+19; GKS+22; ACG+23]. By contrast, in all existing Tendermint-like view-changes [DLS88; Buc16; BG17; YMR+19b; SWN+23; MN23], a player *does not* accept a prepare containing a

lock cert $\left[\widetilde{v}, x\right]$ if it is locked on a strictly higher view $v_i > \widetilde{v}$. In those view-changes, if the leader does not wait Δ to be sure that it receives their highest locks from all honest players, then its prepare message might not be accepted by a quorum of $2t+1$ players, preventing liveness. This limitation of those view-changes is coined as *Livelessness with two-phases* in [YMR+19a, §4.4], and dubbed a "hidden lock" in [ACG+23, §3.1]. In particular, this limitation shows up in [MN23] in their Figure 1: a player P_i votes in step (3) only if the lock certificate received in the prepare is "ranked no lower than" its own high lock$_i$. This is why their claim of being responsive shows up only in the good case.

4 $\mathcal{T}$-PnS: Lightweight, and from any NI-TSS

The following implementation of PnS, which we call $\mathcal{T}$-PnS, is very simple and general. For simplicity we present it with the following specific parameters: the inputs are of the form (v, v_i), where the tag v is a number and $v_i \in [0, v-1]$; and we consider the specific thresholds of Definition 6 (t out of n corruptions, and with $\mathrm{PnS.Prove}$ which aggregates $2t+1$-out-of-n valid reports into a PnS proof that no $t+1$-out-of-n honest players have their input higher than $v_{\max}$). All these parameters can be straightforwardly generalized, e.g., for the general thresholds discussed below Definition 6. $\mathcal{T}$-PnS operates from any $(2t+1)$-out-of-n threshold signature scheme: $\big($Sign, aggregate$^{=}_{2t+1}$, Verify$_{2t+1}^{=}\big)$.

PnS.Report$_i(v, v_i)$ Informally, the output consists of a signature (share) on each of the following testimonies for each $v' \in [v_i, \ldots, v-1]$:

(testi(v,v')) $\ll$ my input in view v is no higher than $v' \gg$

More formally, each such testimony may be just formatted as the pair (v, v'). So formally, the output is the collection of signature (shares): Sign$(\mathsf{sk}_i, (v, v'))_{v' \in [v_i, \ldots, v-1]}$.

PnS.ReportVerif$(i, v, v_i, \bullet)$ it takes as input (in place of $\bullet$): a tuple of signatures shares $(\sigma_{i,v'})_{v' \in [v_i, \ldots, v-1]}$ for some number v_i, and outputs accept if each σ_i is as previously, i.e., a valid signature from P_i on the testimony (v, v').

PnS.Prove$\big(v, I, (\sigma_{i,v'})_{v' \in [v_i, \ldots, v-1]}\big)$ Define $v_{\max}$ as the *lowest* value for which there exists a testimony: $(v, v_{\max})$ signed by the $2t+1$ signers, i.e.:

(testi(v,$v_{\max}$)) $\ll$ my input in view v is no higher than $v_{\max} \gg$

aggregate$^{=}_{2t+1}$ the $2t+1$ signatures (shares) $(\sigma_{i,v_{\max}})_{i \in I}$ on this testimony, into a threshold signature: π, which is the output.

PnS.Verify$(v, v_{\max}, \pi)$ Verify$_{2t+1}^{=}$ whether π is a valid threshold signature on the testimony $(v, v_{\max})$.

The proof of soundness of $\mathcal{T}$-PnS is just: consider π a valid threshold signature on the testimony $(v, v_{\max})$, then this implies that at least $t+1$ (honest) players have their inputs v_i no higher than $v_{\max}$, which is the non-supermajority predicate which was to be proven.

4.1 Optimisation for the leader: constant verification complexity (or one misbehaving player gets publicly exposed).

Upon receiving $2t+1$ reports $(\mathrm{report}_{i,v,v_i})_{i \in I}$ from a $2t+1$-set $I \subset [n]$ of issuers, the leader does not verify the signature shares. Instead, it optimistically computes the highest reported value: $v_{\max} = \max_{i \in I} v_i$, and tentatively applies $\mathrm{PnS.Prove}$, i.e., aggregates the signature shares for $v_{\max}$: $(\sigma_{i,v_{\max}})_{i \in I}$. It then verifies the threshold signature π obtained. If verification rejects, then it searches the signature share(s): $(\sigma_{i^*,v_{\max}})$ which was (were) invalid. Notice that even in this case, the leader needs only verifying $2t+1$ signature shares, not the whole content of the reports. So the leader removes the issuer i^* out of the set I, and waits to receive a report from some new player j. However when this bad event happens, the badly formed report of i^* constitutes a proof that it misbehaved, which the leader can publicly expose. So the price paid by i^* will be much higher than the consequence of its behavior, i.e., having the leader compute another threshold signature.

4.2 Performance. A report for input v_i has bitsize equal to $v - v_i$ authenticators. So it is particularly small for the common case where honest players obtained their high lock_i no more than from a few views behind v. As detailed above, the computation of the leader is *independent of* v, unless one misbehaving player is publicly exposed. In this latter case, the computation of the leader consists in verifying $2t+1$ signature shares. It may have to do such verification t times, but then this means that the leader caught all t corrupt players. Likewise, the PnS obtained has size equal to 1 authenticator (a threshold signature), so both the communication from the leader and verification at the players, are *independent of* v. In conclusion, 2Pc instantiated with $\mathcal{T}$-PnS has linear authenticator complexity: $O(nv)$ and keeps independent of v the load on the leader (unless all corrupt players are caught). Although this does not count in the authenticator complexity, let us further observe that the contents of messages *needs not* be appended to the threshold signature to enable its verification. So when instantiated with a constant-sized TSS, $\mathcal{T}$-PnS is the first PnS which has a *bit complexity* which is constant in n. In Section 6 we will provide further implementations of PnS which are useful in very bad scenarios where v grows large, since their sizes of reports have less worst-case dependency in v (either $+O(\log(v))$ or $\times O(\log(v))$).

5 No Proofs in Good Views, for Free

Technically, a *good view* is one in which no player P_i lately reports a lock which is strictly higher: $v_i > v_{\max}$, than the one of the leader (precisely: than its lock cert$\left[v_{\max}, x_{\max}\right]$ updated after reception of the $2t+1$ first new-view messages). In particular, it must be that $i \notin I$: such a late and isolated P_i is dubbed a "hidden lock" in [ACG+23]. A strict subset of *good views* are called "happy" in [JNFG20; GKS+22; SDZ22b]. A sufficient condition for a view to be "happy" is: GST happened at the beginning of the *previous* view and the leader of the *previous view* was *also* honest. In this section we present an optimization of $\mathsf{2Pc}^{\mathsf{0PnS}}$ which completely removes the overhead at the leader, due to PnS in step **1.**, in all the *good views*. Since the leader is in practice a bottleneck ([CDH+22]), removing the complexity at the leader is particularly desirable. Since this optimization applies to *any* black-box PnS system, it also applies to the naive one [CL02; GAG+19; GKS+22; ACG+23] consisting in forwarding $2t+1$ signed reports.

5.1 Illustration on the toy two-phase BFT. We now illustrate our optimization on 2Pc, in the form of the optimized variant which we call as $\mathsf{2Pc}^{\mathsf{0PnS}}$. We give the main ideas, $\mathsf{2Pc}^{\mathsf{0PnS}}$ is formalized in Section B Algorithm 6. The first ingredient is that the leader tentatively multicasts its prepare message without a PnS. Now, we add the following extra path for accepting such a prepare message. When a player P_i receives a prepare message containing a lock certificate lock cert$\left[\widetilde{v}, \widetilde{x}\right]$ at least as high as its own, i.e., $v_i \leqslant \widetilde{v}$, then it automatically replies with lock vote$\left[v, \widetilde{x}\right]$ (without even checking whether there was a PnS included or not). We dub this extra-path as the *Tendermint unlocking*, since this path was credited to Tendermint by [YMR+19b]. In good case executions, this extra-path is actually *the only one* followed. Indeed, all honest players have a lock as high as $v_i = v - 1$. So the leader is reported such a lock within the first $2t+1$ reports messages, so is able to convince all $2t+1$ honest players to cast a lock vote , yielding the desired lock certificate for the proposed value. In conclusion, both the communication from the leader and the verification by players, are as light as in all non-responsive two-phase BFTs [Buc16; BG17; YMR+19b; MN23] (and [DLS88]).

However, if we kept only this extra-path in 2Pc, then it would not be responsive anymore! As explained in Section 3.9, having only this path is what prevented responsiveness in BFTs such as [DLS88; Buc16; BG17; YMR+19b; MN23]. This is why in $\mathsf{2Pc}^{\mathsf{0PnS}}$ we also keep the path of 2Pc, *in parallel*, as a backup mechanism. Namely, when the leader receives a higher lock than its own (lock cert$\left[v_{\max}, x_{\max}\right]$) from a late P_i (so $i \notin I$), then it sends to P_i a $\mathsf{PnS}(v, v_{\max})$ to support its proposal (in the pseudo code we were underoptimal, for simplicity, and had the leader send a whole prepare message again to P_i). Then any such P_i accepts the prepare message as in 2Pc, namely, if the $\mathsf{PnS}(v, v_{\max})$ is valid.

Finally, the reason why allowing this extra-voting mechanism preserves safety is somewhat classical. Indeed, safety of the Tendermint mechanism is what supports [Buc16; BG17; YMR+19b; MN23], and it follows from a very old and classical induction proof (from [DLS88]). However in our case it may be more subtle because we allow *simultaneously* two paths to vote, i.e., the (PnS-based) one of PBFT and the (good-case) one of Tendermint. So in Section B.1 we give the details of the proof of safety.

We finally make some orthogonal optimization-related remarks related to aggregation of signature shares. In [GL22] they make the observation that, when NI-TSS is embodied by BLS multisignatures, then players can append their BLS signatures σ_i with a classical Chaum-Pedersen proof of equality of exponent (w.r.t. their public key). Then the leader needs only checking this proof to accept σ_i, so it needs not anymore performing the pairing-based BLS verification. In their Table 1 they report a $2\times$ speedup for 256 signatures. Actually the task of the leader can be further sped up. It can tentatively aggregate the batch of $2t+1$ signature shares, then test only validity of the multi-signature obtained. In the rare cases where a few invalid signatures are hidden in the batch, group-testing-based techniques [LM07] enable to find them all in a logarithmic number of pairings.

5.2 Relationship to prior optimizations. As commented in Table 1, a number of BFTs (Fast-Hotstuff [JNFG20], Jolteon [GKS+22], implemented in the late Diem [Tea21], Marlin [SDZ22b], [MN23]) enjoy better parameters in so-called "happy" views. A view is "happy" if the new leader is responsively reported a lock certificate of the *previous* view $v-1$. For instance, if the highest lock ever formed was formed in a view before $v-2$, then the view is not "happy", but it is still good if the leader is reported this lock in the first $2t+1$ reports. This shows that "happy" views are a strict subset of *good* views. In non-happy views, [SDZ22b] have an extra-phase due to a preliminary round-trip which they call "pre-prepare", as explained in [SDZ22b, §V. C]. Such an extra-round-trip is also paid in [GHA+21] in non-good views (as discussed in their §VII, D and in [SDZ22b]). Our new technique to avoid such an extra-phase in $2\mathsf{Pc}^{0\mathsf{PnS}}$ (Algorithm 6), despite our tentative Tendermint-like view-change, is that the leader detects hidden locks *straight from late* new-view *messages*. In the related works [GHA+21; ACG+23] it is not specified if players report their highest known lock certificate (high lock$_i$) to the leader in new-view messages. For our optimization to carry over these works, it would be needed that they incoporate this specification (as a by-product, this would implement the abstraction called "HighVoteReq" in [ACG+23]).

6 Two more PnS, for Bad Scenarios

We now introduce two more classes of implementations of PnS. The former has no dependency in v at all and the latter only in $O(\log(v))$. Hence, they may serve as a replacement for $\mathcal{T}$-PnS (Section 4) in bad scenarios where many consecutive views elapsed since the beginning of the instance. For simplicity we also present them with the same specific choice of parameters as in Section 4. The implementations obviously carry over general parameters.

6.1 From SNARKS of signed values in a range. This implementation is introduced in [AR23], so we just recall it here. We describe it when narrowed to our specific parameters. Consider a prover (a leader), which received $2t+1$ signed messages, signed by a subset $I \subset [n]$ of $2t+1$-out-of n public keys. Their contents come as pairs: $m_i = (v, v_i)$. The left entry v is *fixed public*, e.g., it encodes the current view and the instance number. Their right entries v_i are all in some range $[0, v_{\max}]$. Their tool is a succinct non-interactive argument of knowledge (a "SNARK") enabling the prover to prove knowledge of what we have just described. Namely: of signatures issued by a subset I of $2t+1$ out of n public keys, each on a message of the form (v, v_i) for a fixed public v, and such that all $v_i \in [0, v_{\max}]$ $\forall i \in I$. The latter range condition is what they denote as a *predicate* on the signed messages. So in our formalism, the $\mathrm{PnS.Report}_i$ consists in generating a signature on (v, v_i), and the $\mathrm{PnS.Prove}$ consists in generating a SNARK as just described. In conclusion, the advantage of this SNARKs-based instantiation of PnS is its short size, since the message contents, more precisely: the variable parts v_i's, *need not* be exhibited.

Remark 7. We make the observation that this gain of size is fully visible when the SNARK is applied to related contexts where the sizes of signed messages are larger. Such an example is given by the problem of *responsive* view-change of optimistically fast BFT. In this context, the leader multicasts $2t+1$ signed messages, and players check that no $(t+1)$-subset of them have equal content. In the v1 of this work it is detailed how . Non-responsive PoE's can be found in [RTA22].

6.2 From any aggregate signature scheme. This instantiation of PnS, which we denote as $\mathcal{A}$-PnS, is simply the one where players send a report of the form $m_i = (v, v_i)$ along with a signature on it. Then, the leader forms a PnS

consisting of: a set I of $2t+1$ senders indices, their reports, and an $\text{AGGREGATE}^{\neq}_{I}$ of their signatures. In [JNFG20] this approach is instantiated with the aggregate signature scheme of [BGLS03], which has verification complexity equal to $2t+1+1$ pairings.

7 Compiling Multi- into Aggregate- Signatures

We consider *any* multi-signature scheme (SIGN, VERIFY, $\text{AGGREGATE}^{=}$, $\text{VERIFY}^{=}$). The following very simple construction, called $\mathcal{M}\text{to}\mathcal{A}$, compiles it into a restricted-purpose aggregate signature scheme. Its first restrition is that it operates on sets of signed messages of which all issuers are *distinct*, i.e.: $(m_i)_{I \subset [n]}$. Then, without restricting generality, we assume that the messages input to $\text{AGGREGATE}^{=}$ are of the form $m_i = (v, v_i)$, where the *tag* v is the same for all $i \in I$, and where the *variable parts of the messages*, the v_i's, are within a predefined range. The second restriction is then that the signature must be used only once for each tag v. Said otherwise, in the unforgeability game, we do not allow queries on messages with the same prefix (v, v_i). Let us denote this range as $[0, 2^{\ell}-1]$ (so in our context of PnS: $\ell = \log(v)$). The setup is that each player P_i generates 2ℓ signature key pairs:

$$(\mathsf{sk}_{i,j,b}, \mathsf{pk}_{i,j,b})_{j \in [0,\ell[, b \in \{0,1\}}$$

of which it publishes the 2ℓ public keys (and proofs of possession if needed). So somehow, each player emulates 2ℓ signing machines, hence a total of $N := 2\ell.N$ signing machines. Let us fix the notation that the bit decomposition of a variable part of some input, v_i, is denoted as $[v_{i,0}, \ldots, v_{i,\ell-1}]$. To generate a signature on v_i, generate a signature *on the same message content:* v for each bit of this decomposition. The idea is that distinguishing which bit is encoded by the signature, and whether it is 0 or 1, is *not* achieved by modifying the content signed (always v). It is instead achieved by changing the *signing key* used. That way, the content (v) stays always the *same*, so signatures encoding different bits can still be *efficiently aggregated with a multi-signature*. The proof of unforgeability simply consists in considering a forger which creates a valid aggregate signature, σ, for a set of messages containing a (v, v_{i^*}) for one of the challenge keys: $\mathsf{pk}_{i^*} = [\mathsf{pk}_{i^*,j,b}]$, such that it did not query a signature before to pk_{i^*} for the tag v. Then, consider any bit of v_{i^*}, e.g., the first: $v_{i^*,1}$. We have that σ is a legitimate forgery against the underlying multisignature scheme (in the N-users setting) with respect to the target key $\mathsf{pk}_{1,j,v_{i^*,1}}$.

$\textbf{SIGN}\Big((\mathsf{sk}_{i,j,b})_{j \in [\ell-1], b \in \{0,1\}}, (v, v_i)\Big)$ Output the ℓ-uple of signatures on v:

$$\Big[\text{SIGN}(\mathsf{sk}_{i,j,v_{i,j}}, v) \ : \ j \in [0, \ell-1]\Big] \tag{3}$$

$\textbf{VERIFY}(i, (v, v_i), \bullet)$ it takes as input (in place of $\bullet$): an ℓ-uple of signatures $\big[\sigma_{i,j} \ : \ j \in [0, \ell-1]\big]$. Output accept if each $\sigma_{i,j}$ is a valid signature on v, under the key $\mathsf{pk}_{i,j,v_{i,j}}$.

$\textbf{AGGREGATE}^{\neq}\Big(I, \Big\{(v, v_i)\ ,\ \big[\sigma_{i,j} \ : \ j \in [0, \ell-1]\big]\Big\}_{i \in I}\Big)$ Let $\mathcal{I}$ be the $2\ell n$-sized binary array which encodes the subset of the $\ell|I|$-out-of-$2\ell n$ public keys, $\mathsf{pk}_{i,j,v_{i,j}}$, which signed the $\sigma_{i,j}$'s. Recall that these keys are indexed by the (multi)-indices $(i, j, v_{i,j})_{i \in I, j \in [\ell-1]}$.
Output $\sigma \leftarrow \text{AGGREGATE}^{=}\Big(\mathcal{I},\ v,\ \{\sigma_{i,j}\}_{i \in I, j \in [0,\ell-1]}\Big)$.

$\textbf{VERIFY}^{\neq}\big(I, (v_i)_{i \in I}, \sigma\big)\Big)$ From I and the binary decomposition of the v_i's, form the binary array $\mathcal{I}$ encoding the $\ell|I|$-sized subset of (multi)-indices $(i, j, v_{i,j})_{i \in I, j \in [\ell-1]}$.
Output $\text{VERIFY}^{=}(\mathcal{I}, v, \sigma)$.

7.1 Instantiation with PoP-based BLS multisignatures.

Let us first consider the application of $\mathcal{M}\text{to}\mathcal{A}$ to the BLS-multisignatures of [RY07]. Its verification costs 2 pairings, v.s. $|I|+1$ pairings for [BGLS03]. The public keys are of size 2ℓ group elements. So this poses no problem in our setting where ℓ, i.e., the number of bits by which messages *differ*, is small. Of course this would pose a problem with a large ℓ. We make the observation that this specific instantiation is exactly the pairing-based aggregate signature scheme called "Wendy" in [GHA+21] (and

which inspired $\mathcal{M}$to$\mathcal{A}$). This observation enables to replace their security proof, by a tight reduction to the security of [RY07]. We benchmark this instantiation in Section 8.1, against BGLS aggregate signatures. We conclude that it provides an efficient PnS, which can be used as a backup to the simple TSS-based one of Section 4, in the scenarios where the number v of consecutive bad views grows to dozens (which should be extremely rare, unless the system is misconfigured).

7.2 Comparison and performance of $\mathcal{M}$to$\mathcal{A}$ in general. The construction can also be compared to the pairing-based one of [LOS+13]. They had a similar idea of one-key-per-bit, and also achieved verification in two pairings. But their scheme is agnostic of situations, as ours, in which messages differ by a few number ℓ of bits. Hence, the number of their keys is equal to the *whole* bitsize of messages, *vs* in our case, equal to only 2ℓ. They require sequential signing (one signer passes the intermediate aggregate to the next signer), so this does not match our non-interactive Definition 1 (and is impractical for BFT). Finally, a benefit of $\mathcal{M}$to$\mathcal{A}$ is its genericity, e.g., it can be instantiated with post-quantum multisignatures. Post-quantum ones with *non-interactivity* (Definition 1) were recently constructed [KCLM22; FSZ22].

8 Optimizations, Implementations and Comparisons

First, in Table 3 we provide a comparison of the complexities of all the implementations of PnS considered. Then in Section 8.1 and appendix C we describe implementations and optimizations related to pairing-based PnS. All our implementations were run on a laptop with Core i5-8265U (8 cores at 1.6GHz), 16GB of RAM, with the library gnark-crypto on Go [BPH+22]. They were run with curve BN254, for which the uncompressed size of a point in $\mathbb{G}_1$ is 512 bits and of a point in $\mathbb{G}_2$ is 1024 bits. They were also run with BLS12-377, for which the uncompressed size of a point in $\mathbb{G}_1$ is 768 bits and of a point in $\mathbb{G}_2$ is 1536 bits. Compressed points, i.e., their x-coordinate plus one bit, are twice smaller. Each number is the mean over 10 executions. The code is available at https://anonymous.4open.science/r/consensus-D7E0/ . In our implementations, we considered baseline BLS signatures implemented from type III pairings, which are the most used since the most efficient ones [AHO16; BCLS22]. In this type III setting, it is desirable for several reasons that the public key has also a component pk' in G_1, i.e., is of the form $(\mathsf{pk}', \mathsf{pk}) = (\mathsf{sk}.G_1, \mathsf{sk}.G_2)$. Such a double key is required in existing reductionist proofs of BGLS-aggregation [Dar10; Lac18]. Another benefit is that multi-scalar multiplications are $3\times$ faster in $\mathbb{G}_1$. We will leverage this in both Section 8.1 and appendix C. We finally mention that to evaluate related works we needed to compute large products of pairings, namely: for verifying one BGLS signature in Section 8.1, and for batching the verification of the PoP's of [RY07], in Appendix C. We used the optimized implementation of products of pairings in gnark-crypto, inherited from [GS06].

8.1 Pairing-based PnS for bad scenarios: BGLS vs $\mathcal{M}$to$\mathcal{A}$. In Table 4 we compare the verification times of two instantiations of aggregate-signature-based PnS, which are usable as a backup for $\mathcal{T}$-PnS in bad scenarios where v grows very large without a commit. The first is suggested in [JNFG20]. There, the aggregate signature is the one of [BGLS03; BDN18]. Its verification involves a product of at least as many pairings as distinct reported values. We considered $\min(2t+1, v)$ distinct ones, since nothing prevents the t corrupt players from reporting a distinct lock certificate each (in favorable executions, this would have given $\min(t+1, v)$).

The second is obtained by applying the compiler $\mathcal{M}$to$\mathcal{A}$ from any BLS multisignature scheme. By "BLS multisignature scheme" we mean any one such that the verification formula is as simple as in the original [Bol03, §4]. Namely, the verification of a purported multisignature σ on a message (v, in the context of $\mathcal{M}$to$\mathcal{A}$) with respect to a $(2t+1)$-sized subset $I \in [n]$ of issuers is as follows. In the context of $\mathcal{M}$to$\mathcal{A}$, these issuers used a total of $\ell(2t+1)$ keys to create the signatures, out of which the multisignature was produced. In the more efficient type-III pairing context, these public keys come as the pairs in $(\mathbb{G}_1, \mathbb{G}_2)$: $\big(X'_{i,j,b_{i,j}}, X_{i,j,b_{i,j}}\big)_{i \in I, j \in [0,\ell-1]}$. We denote as $\mathcal{I} = \big\{(i, j, b_{i,j}) \ : \ i \in I,\ j \in [0, \ell-1]\big\}$ the set of their $\ell(2t+1)$ (multi)-indices. Then, verification consists in verifying if $e\big(\sigma, G_2\big) = e\big(H(v), \sum_{i \in I, j \in [0,\ell-1]} X_{i,j,b_{i,j}}\big)$. There are three schemes which enable such a fast verification and which operate in the bulletin board PKI model. Their setups are benchmarked in Appendix C (of which a new one: $\mathcal{MSP}$-skoe, which we propose). We applied a further verification speedup for BLS multisignatures, which is a nice trick of [BCLS22]. The aggregator also gives to the verifier the aggregated key in $\mathbb{G}_2$, i.e.,

	report (auth. complexity)	PnS.Prove complexity	PnS (auth. complexity)	PQ
$\mathcal{T}$-PnS (Section 4)	generation cost: $v - v_i$ verification cost: 1	1 threshold sign. aggregation	1 threshold sign. verification	✓
$\mathcal{A}$-PnS using [BGLS03]	1	2×(addition of k points)	$k+1$ pairings	✗
$\mathcal{A}$-PnS using [BDN18] + Appendix D	1	2×(addition of k points)	$\min(v+1\,,\ k+1)$ pairings	✗
$\mathcal{A}$-PnS using $\mathcal{M}$to$\mathcal{A}$ (Section 7)	$\log(v)$	$(\log(v)+1).k$-multisignature aggregation	$(\log(v)+1).k$-multisignature verification	✓
…itself instantiated with BLS multisigs	$\log(v)$	addition of $(\log(v)+1).k$ points	2 pairings	✗
SNARKs of signed messages [AR23]	7 pairings	$O(n\log(n))$ point additions	6 pairings	✗

Table 3: Authenticator complexities of various instantiations of PnS, for a number of $k = 2t+1$ reports. The first column is for a single report. In $\mathcal{T}$-PnS we distinguished that the complexity for an honest player to create a report is as much signature shares as the gap between its high lock_i and the current view; whereas the one for the leader is only to verify 1 signature share (the one for $v_{\max}$). The complexities at the verifier ignore point-additions in pairing-based schemes. The last column PQ displays whether it can be instantiated with post-quantum schemes.

$X_{\mathcal{I}} := \sum_{i\in I, j\in[0,\ell-1]} X_{i,j,b_{i,j}}$. Instead of recomputing it directly, the verifier checks if $X_{\mathcal{I}}$ is well-formed by computing the aggregate key in $\mathbb{G}_1$: $X'_{\mathcal{I}} := \sum_{i\in I, j\in[0,\ell-1]} X'_{i,j,b_{i,j}}$, then checks equality of the exponents by testing if: $e(X'_{\mathcal{I}}, G_2) = e(G_1, X_{\mathcal{I}})$. The benefit is that, for $N \sim 1000$, then adding N points in $\mathbb{G}_1$ is twice as fast as in $\mathbb{G}_2$.

	BN254 curve	BLS12-377 curve
[BGLS03; BDN18]	39.8 ms	70.1 ms
$\mathcal{M}$to$\mathcal{A}$ from BLS multisig.	1.0 ms	1.8 ms

Table 4: Verification times of a BGLS aggregate signature from $2t+1$ players, on messages taking values in some range $[0, v-1 = 2^\ell - 1]$. Verification time of an $\mathcal{M}$to$\mathcal{A}$ aggregate signature from $2t+1$ players, when instanted with BLS multisignatures. Recall that an $\mathcal{M}$to$\mathcal{A}$ signature consists in multi-signature for $\ell.(2t+1)$ public keys. Our parameters are $v \in [0, 2^7\text{-}1]$, so $\ell = 7$, and $t = 64$ so $2t+1 = 129$. Thus the total number of public keys in the multisignature is 903.

References

[ACG+23] I. Abraham, N. Crooks, N. Giridharan, H. Howard, and F. Suri-Payer. "BeeGees: strengthened liveness in chained BFT". In: *PODC*. 2023.

[ACR21] T. Attema, R. Cramer, and M. Rambaud. "Compressed Sigma-Protocols for Bilinear Circuits and Applications to Logarithmic-Sized Transparent Threshold Signature Schemes". In: *ASIACRYPT*. 2021.

[AHO16] M. Abe, F. Hoshino, and M. Ohkubo. "Design in Type-I, Run in Type-III: Fast and Scalable Bilinear-Type Conversion Using Integer Programming". In: *CRYPTO*. 2016.

[AMS19] I. Abraham, D. Malkhi, and A. Spiegelman. "Asymptotically Optimal Validated Asynchronous Byzantine Agreement". In: *PODC*. 2019.

[AR23] *Succinct Proofs of Partial Knowledge of Signed Messages Satisfying a Public Predicate*. A preliminary version appears as the unpublished extra-Section 7 of "Compressed Sigma-Protocols for Bilinear Circuits and Applications to Logarithmic-Sized Transparent Threshold Signature Schemes", eprint 2020/1447.

[BBB+23] A. Bhat, A. Bandarupalli, S. Bagchi, A. Kate, and M. Reiter. *Unique Chain Rule and its Applications*. Financial Cryptography and Data Security. 2023.

[BCG+23] F. Baldimtsi, K. K. Chalkias, F. Garillot, J. Lindstrom, B. Riva, A. Roy, A. Sonnino, P. Waiwitlikhit, and J. Wang. *Subset-optimized BLS Multi-signature with Key Aggregation*. ePrint 2023/498. 2023.

[BCG22a] M. Bravo, G. V. Chockler, and A. Gotsman. "Liveness and Latency of Byzantine State-Machine Replication". In: *DISC*. 2022.

[BCG22b] M. Bravo, G. V. Chockler, and A. Gotsman. "Making Byzantine consensus live". In: *Distributed Comput.* (2022).

[BCK+22] M. Bellare, E. Crites, C. Komlo, M. Maller, S. Tessaro, and C. Zhu. "Better than Advertised Security for Non-interactive Threshold Signatures". In: *CRYPTO*. 2022.

[BCLS22] J. Burdges, O. Ciobotaru, S. Lavasani, and A. Stewart. *Efficient Aggregatable BLS Signatures with Chaum-Pedersen Proofs*. ePrint 2022/1611. 2022.

[BD21] M. Bellare and W. Dai. "Chain Reductions for Multi-Signatures and the HBMS Scheme". In: *ASIACRYPT*. 2021.

[BDN18] D. Boneh, M. Drijvers, and G. Neven. "Compact Multi-signatures for Smaller Blockchains". In: *ASIACRYPT*. 2018.

[Ben04] D. B. and Ben Lynn and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *J. Cryptology* (2004).

[Ber15] D. J. Bernstein. *Multi-user Schnorr security, revisited*. ePrint 2015/996. 2015.

[BG17] V. Buterin and V. Griffith. "Casper the Friendly Finality Gadget". In: *arxiv 1710.09437* (2017).

[BGLS03] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *EUROCRYPT*. 2003.

[BGR98] M. Bellare, J. A. Garay, and T. Rabin. "Fast batch verification for modular exponentiation and digital signatures". In: *EUROCRYPT*. 1998.

[BNN07] M. Bellare, C. Namprempre, and G. Neven. "Unrestricted Aggregate Signatures". In: *ICALP*. 2007.

[Bol03] A. Boldyreva. "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme". In: *PKC*. Latest long version at https://faculty.cc.gatech.edu / aboldyre/papers/b.pdf. 2003.

[BPH+22] G. Botrel, T. Piellard, Y. E. Housni, A. Tabaie, and I. Kubjas. *ConsenSys/gnark-crypto: v0.6.1*. 2022.

[BTA+19] S. Basu, A. Tomescu, I. Abraham, D. Malkhi, M. K. Reiter, and E. G. Sirer. "Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols". In: *CCS*. 2019.

[Buc16] E. Buchman. "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains". PhD thesis. University of Guelph, 2016.

[CDH+22] J. Camenisch, M. Drijvers, T. Hanke, Y.-A. Pignolet, V. Shoup, and D. Williams. "Internet Computer Consensus". In: *PODC*. 2022.

[CGK+22] S. Cohen, R. Gelashvili, E. Kokoris-Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman. "Be Aware of Your Leaders". In: *Financial Cryptography and Data Security*. 2022.

[CHM+20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: *EUROCRYPT*. 2020.

[CHP07] J. Camenisch, S. Hohenberger, and M. Ø. Pedersen. "Batch Verification of Short Signatures". In: *EUROCRYPT*. 2007.

[CL02] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Trans. Comput. Syst.* (2002).

[CL99] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI*. 1999.

[Dam92] I. Damgård. "Towards Practical Public Key Systems Secure Against Chosen Ciphertext attacks". In: *CRYPTO*. 1992.

[Dar10] S. C. and Darrel Hankerson and Edward Knapp and Alfred Menezes. "Comparing two pairing-based aggregate signature schemes". In: *Designs, Codes and Cryptography* (2010).

[DCX+23] S. Das, P. Camacho, Z. Xiang, J. Nieto, B. Bunz, and L. Ren. *Threshold Signatures from Inner Product Argument: Succinct, Weighted, and Multi-threshold*. ePrint 2023/598. 2023.

[DGNW20] M. Drijvers, S. Gorbunov, G. Neven, and H. Wee. "Pixel: Multi-signatures for Consensus". In: *USENIX*. 2020.

[DLS88] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* (1988).

[DN18] M. D. Dan Boneh and G. Neven. *Bls multi-signatures with public-key aggregation*. https://crypto.stanford.edu/ dabo/pubs/p 2018.

[DXKR23] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. Ren. "Practical Asynchronous High-threshold Distributed Key Generation and Distributed Polynomial Sampling". In: *Usenix Security symposium*. 2023.

[DZJR23] X. Dai, B. Zhang, H. Jin, and L. Ren. *ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path*. ePrint 2023/679. 2023.

[EEK+23] M. F. Esgin, O. Ersoy, V. Kuchta, J. Loss, A. Sakzad, R. Steinfeld, X. Yang, and R. K. Zhao. "A New Look at Blockchain Leader Election: Simple, Efficient, Sustainable and Post-Quantum". In: *AsiaCCS*. 2023.

[FPS20] G. Fuchsbauer, A. Plouviez, and Y. Seurin. "Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model". In: *EUROCRYPT*. 2020.

[FSZ22] N. Fleischhacker, M. Simkin, and Z. Zhang. "Squirrel: Efficient Synchronized Multi-Signatures from Lattices". In: *CCS*. 2022.

[GAG+19] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. "SBFT: A Scalable and Decentralized Trust Infrastructure". In: *DSN*. 2019.

[GHA+21] N. Giridharan, H. Howard, I. Abraham, N. Crooks, and A. Tomescu. *No-commit proofs: Defeating livelock in bft*. eprint 2021/1308. 2021.

[GJM+23] S. Garg, A. Jain, P. Mukherjee, R. Sinha, M. Wang, and Y. Zhang. *hinTS: Threshold Signatures with Silent Setup*. ePrint 2023/567. 2023.

[GKM+18] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. "Updatable and Universal Common Reference Strings with Applications to zk-SNARKs". In: *CRYPTO*. 2018.

[GKS+22] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang. "Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback". In: *Financial Cryptography and Data Security*. 2022.

[GL22] D. Galindo and J. Liu. "Robust Subgroup Multi-signatures for Consensus". In: *CT-RSA*. Ed. by S. D. Galbraith. 2022.

[Gro21] J. Groth. *Non-interactive distributed key generation and key resharing*. ePrint 2021/339. 2021.

[GS06] R. Granger and N. P. Smart. "On Computing Products of Pairings". In: *eprint 2006/172* (2006).

[GV10] R. Guerraoui and M. Vukolic. "Refined quorum systems". In: *Distributed Comput.* (2010). extends PODC'07.

[GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". In: *eprint 2019/953* (2019).

[JNFG20] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai. *Fast-HotStuff: A Fast and Resilient HotStuff Protocol*. arxiv. 2020.

[KCLM22] I. Khaburzaniya, K. Chalkias, K. Lewi, and H. Malvai. "Aggregating and Thresholdizing Hash-Based Signatures Using STARKs". In: *Asia CCS*. 2022.

[KS22] Y. Kondi and A. Shelat. *Improved Straight-Line Extraction in the Random Oracle Model With Applications to Signature Aggregation*. ePrint 2022/393. 2022.

[LA23] A. Lewis-Pye and I. Abraham. *Fever: Optimal Responsive View Synchronisation*. 2023.

[Lac18] M. Lacharité. "Security of BLS and BGLS signatures in a multi-user setting". In: *Cryptogr. Commun.* (2018).

[Lam06] L. Lamport. *Lower Bounds for Asynchronous Consensus*. https://lamport.azurewebsites.net/pubs/lower-bound.pdf. 2006.

[Lam11] L. Lamport. "Byzantizing Paxos by Refinement". In: *DISC*. 2011.

[Lam98] L. Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* (1998).

[LL22] C. Lenzen and J. Loss. "Optimal Clock Synchronization with Signatures". In: *PODC*. 2022.

[LLT22] Y. Lu, Z. Lu, and Q. Tang. "Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT". In: *CCS*. 2022.

[LM07] L. Law and B. J. Matt. “Finding Invalid Signatures in Pairing-Based Batches”. In: *Cryptography and Coding*. Ed. by S. D. Galbraith. 2007.
[LOS+13] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters. “Sequential Aggregate Signatures, Multisignatures, and Verifiably Encrypted Signatures Without Random Oracles”. In: *J. Cryptol.* (2013).
[LSP82] L. Lamport, R. E. Shostak, and M. C. Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* (1982).
[MBKM19] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. “Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings”. In: *CCS*. 2019.
[MN23] D. Malkhi and K. Nayak. *Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT*. ePrint 2023/397. 2023.
[MOR01] S. Micali, K. Ohta, and L. Reyzin. “Accountable-Subgroup Multisignatures: Extended Abstract”. In: *CCS*. 2001.
[NK20] O. Naor and I. Keidar. “Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR”. In: *DISC*. 2020.
[NRS21] J. Nick, T. Ruffing, and Y. Seurin. “MuSig2: Simple Two-Round Schnorr Multi-Signatures”. In: *CRYPTO*. 2021.
[PW23] J. Pan and B. Wagner. “Chopsticks: Fork-Free Two-Round Multi-Signatures from Non-Interactive Assumptions”. In: *EUROCRYPT*. 2023.
[Rob20] A. T. and Robert Chen and Yiming Zheng and Ittai Abraham and Benny Pinkas and Guy Golan-Gueta and Srinivas Devadas. “Towards Scalable Threshold Cryptosystems”. In: *IEEE S&P*. 2020.
[RTA22] M. Rambaud, A. Tonkikh, and M. Abspoel. “Linear View Change in Optimistically Fast BFT”. In: (2022).
[RY07] T. Ristenpart and S. Yilek. “The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks”. In: *EUROCRYPT*. 2007.
[SDPV22] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolic. “[Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks”. In: *J. Syst. Res.* (2022).
[SDZ22a] X. Sui, S. Duan, and H. Zhang. “BG: A Modular Treatment of BFT Consensus”. In: (2022).
[SDZ22b] X. Sui, S. Duan, and H. Zhang. “Marlin: Two-Phase BFT with Linearity”. In: *DSN*. 2022.
[SG02] V. Shoup and R. Gennaro. “Securing Threshold Cryptosystems against Chosen Ciphertext Attack”. In: *J. Cryptol.* (2002).
[SLL10] D. Schultz, B. Liskov, and M. Liskov. “MPSS: Mobile Proactive Secret Sharing”. In: *ACM Trans. Inf. Syst. Secur.* (2010).
[SWN+23] P. Sheng, G. Wang, K. Nayak, S. Kannan, and P. Viswanath. “Player-Replaceability and Forensic Support are Two Sides of the Same (Crypto) Coin”. In: *Financial Cryptography and Data Security*. 2023.
[Tea21] T. D. Team. *DiemBFT v4: State Machine Replication in the Diem Blockchain*. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf. 2021.
[Tea22] A. Team. *aptos*. https://aptos.dev/. 2022.
[TZ23] S. Tessaro and C. Zhu. “Threshold and Multi-Signature Schemes from Linear Hash Functions”. In: *EUROCRYPT*. 2023.
[VAFB22] R. Vassantlal, E. Alchieri, B. Ferreira, and A. Bessani. “COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services”. In: *IEEE S&P*. 2022.
[YMR+19a] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. “HotStuff: BFT Consensus with Linearity and Responsiveness”. In: *PODC*. 2019.
[YMR+19b] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. ““Two-phase HotStuff” in Sections 4.4 and 6 of HotStuff: BFT Consensus with Linearity and Responsiveness”. In: *PODC*. 2019.
[ZS92] Y. Zheng and J. Seberry. “Practical Approaches to Attaining Security Against Adaptively Chosen Ciphertext Attacks (Extended Abstract)”. In: *CRYPTO*. 1992.

A Further Details for the Toy Two-Phase Consensus

The data structures of 2Pc are in Fig. 5, the protocol in Algorithm 2 and the proof in Algorithm 2) The steps of 2Pc are specified with respect to the current view number v. Some steps are specified to be taken only in the first view

($v = 1$), they are in addition highlighted in gray for extra-clarity. Players perform the steps numbered 0., 2. and 4., as soon as they can, in any order. Some additional steps, numbered as 1., 3. and 5., are taken only by the leader.

Utilities

- A **lock vote**$\big[\boldsymbol{v}, \boldsymbol{x}\big]$ / **commit vote**$\big[\boldsymbol{v}, \boldsymbol{x}\big]$ is a triple $(v \in \mathbb{N}^*, x \in \mathcal{X}, \sigma_i)$, where σ_i is a signature (share) of any player, e.g., P_i, on the triple $\big[\{\text{“lock vote” / “commit vote”}\}, v, x\big]$. P_i is then called the *issuer* of this lock vote / commit vote. The σ_i is left-out of the notation, for brevity.
- A **lock certificate** is a triple $(\widetilde{v} \in \mathbb{N}, \widetilde{x} \in \mathcal{X}, \sigma)$, where $\widetilde{v}$ is called the *view number attached*, $\widetilde{x}$ a valid value, dubbed as the *locked value* and:
  - *if* $\widetilde{v} \geq 1$, then σ is the $\text{AGGREGATE}_{2t+1}^{=}$ of $2t+1$ signatures over identical commit vote$\big[\widetilde{v}, \widetilde{x}\big]$ from distinct issuers.
  - *else*, i.e., if $\widetilde{v} = 0$, then $\sigma = \perp$. We say that such lock certificate is *non-locking*, and identify it with $\widetilde{x}$.

  We say that a lock certificate is (strictly) *higher* than another one if the view number attached is (strictly) higher. For brevity, we denote it as **lock cert**$\big[\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{x}}\big]$, omitting σ.
- **high lock**$_{\boldsymbol{i}}$ is a lock certificate locally stored by every player P_i. Upon starting the BFT instance, P_i initializes high lock$_i$ as the (non-locking) lock cert$\big[0, x_{P_i}\big]$, where x_{P_i} is P_i's input to the BFT instance. Upon receiving or creating a lock certificate, say, lock cert$\big[v_i', \widetilde{x_i}'\big]$, which is strictly higher than its high lock$_i$, then P_i sets high lock$_i \leftarrow$ lock cert$\big[v_i', \widetilde{x_i}'\big]$.
- A **new-view** message is a tuple of the form $\big(\text{“new-view”}, \text{report}_{i,v,v_i}, \text{lock cert}\big[v_i, \widetilde{x}\big]\big)$, where $v_i \in [0, v-1]$ and report_{i,v,v_i} is a valid report of P_i for (v, v_i).
- A **prepare** message is a tuple of the form $\big(\text{“prepare”}, \pi, \text{lock cert}\big[v_{\max}, x_{\max}\big]\big)$, where $v_{\max} \in [0, v-1]$ and π is a *Proof of non-supermajority for* $v_{\max}$ *in view* v ($\mathsf{PnS}(v, v_{\max})$). If $v_{\max} = v - 1$, then we have simply $\pi = \perp$ because the non-supermajority predicate is automatically satisfied (no commit vote could have been cast in view v at the beginning of v).
- A decision certificate **commit cert**$\big[\boldsymbol{v} \in \mathbb{N}^*, \boldsymbol{x} \in \boldsymbol{\mathcal{X}}\big]$ is the $\text{AGGREGATE}_{2t+1}^{=}$ of $2t+1$ signatures over identical commit vote$\big[v, x\big]$'s from distinct issuers.

Figure 5: Data structures for **2Pc** consensus (Algorithm 2).

A.1 Reminder of the proof of liveness and safety of 2Pc. Liveness is straightforward: from any $2t+1$ valid reports, the leader can form a PnS for the highest reported locked view number: $\mathsf{PnS}(v, \max_{i \in I} v_i)$, which is a fortiori a $\mathsf{PnS}(v, v_{\max})$. From there, it can obtained all the required votes for the proposed value $x_{\max}$.

As for safety, the argument is classical since PBFT [CL99], and actually the conceptualization of PnS makes it even easier to follow! Let us make the induction assumption that for all view numbers $v' \in [0, v-1]$ included, any value y for which a lock certificate was formed in view v': lock cert$\big[v', y\big]$, must verify the safety invariant coined as safe(v',y) in Equation (1). Recall that this means that no other value than, possibly, y, could have been committed in any view up to v'. We now show that the induction assumption holds for view v. Assume by contradiction that a lock cert$\big[v, x\big]$ for some value is formed, and that a conflicting value y was committed in a previous view $v' < v$ (the supermajority needed to make a lock certificate rules out that y could have been committed in v). But for the lock cert$\big[v, x\big]$ to be formed, it must be that at least one (and even $t+1$) honest players checked existence of a lock cert$\big[v_1, x\big]$ and of a $\mathsf{PnS}(v, v_1)$, for some $v_1 < v$. We now show that this could actually not happen in any case, which concludes the proof:

either $\boldsymbol{v' \leqslant v_1}$: then this would violate the induction assumption for v_1, since $v_1 < v$.
or $\boldsymbol{v_1 < v'}$: then this violates the supermajority predicate proven by $\mathsf{PnS}(v, v_1)$, i.e., *no value could have been committed in any view in* $]v_1, v]$.

B Further Details for the No-Proofs-in-Good-Views

B.1 Proof of liveness and safety of 2Pc$^{\mathsf{0PnS}}$ Liveness follows from the one of 2Pc, since we *added* an extra-path for voting, without removing the previous path.

2Pc$^{\mathsf{0PnS}}$: no PnS sent in good views.

0. Report Unchanged

1. Prepare - Tendermint unlocking *If* $v = 1$: unchanged
Else if $v \geq 2$: Leader L_v waits until it receives a set $\mathrm{NV}_{\mathrm{set}}$ of valid $2t+1$ new-view messages, indexed by some $I \subset [n]$: $\mathrm{NV}_{\mathrm{set}} = \left\{ (\text{“new-view”}, \mathrm{report}_{i,v,v_i}, \mathsf{lock\ cert}[v_i, \widetilde{x}]) \ : \ i \in I \right\}$. Denote by $\mathsf{lock\ cert}[v_{\max}, x_{\max}]$ the high lock_{L_v} of L_v.
//One optimization is that instead of verifying validity of the $2t+1$ reports, the leader L_v can tentatively form a $\mathsf{PnS}(v, v_{\max})$ out of them. If the PnS obtained is invalid, L_v publicly exposes the invalid report(s), thereby evidencing misbehavior of their issuer(s). Then it waits for further reports to replace the invalid one(s).
Then it muticasts a prepare message *without a PnS*: (“prepare”, $\mathsf{lock\ cert}[v_{\max}, x_{\max}]$).

1. Prepare - non-good case: PnS-unlocking the hidden locks *If* the following non-good event ever happens, then the leader opens the following non-good thread *in parallel* of the rest (in particular, of **3.**). The non-good event is reception of a (late) new-view message, issued by some P_i, reporting a strictly higher lock certificate than L_v's: $v_i > v_{\max}$.
The non-good thread is as follows (the leader stops it upon succeeding in forming a $\mathsf{lock\ cert}[v, x_{\max}]$, in **3.**):
Leader L_v forms, out of $\mathrm{NV}_{\mathrm{set}}$, a *proof of non-supermajority* π *for* $v_{\max}$ *in view* v as in Equation (proof) of Fig. 5 (it needs to do so only once).
Then it sends to every such P_i a prepare message as in 2Pc: (“prepare”, π, $\mathsf{lock\ cert}[v_{\max}, x_{\max}]$).

2 Lock Vote Every player P_i waits until it accepts a prepare message for the first time from L_v. There are two possibilities for a prepare message to be accepted:

(a) Tendermint unlocking if the prepare message contains a lock certificate at least as high: $\mathsf{lock\ cert}[\widetilde{v}, \widetilde{x}]$ than then one of P_i: $v_i \leqslant \widetilde{v}$, then P_i accepts it.

(b) bad case: PnS unlocking the hidden locks (Unchanged: otherwise, if the prepare message is as in 2Pc, i.e., of the form $(\text{“prepare”}, \pi, \mathsf{lock\ cert}[\widetilde{v}, \widetilde{x}])$ with π a $\mathsf{PnS}(v, \widetilde{v})$, then P_i accepts it.)

Then P_i replies with $\mathsf{lock\ vote}[v, \widetilde{x}]$.

Remaining steps 3., 4.& 5. unchanged

Algorithm 6: Optimization for 2Pc in good views. The unchanged steps compared to 2Pc are shaded-out.

As for safety, let us make the same induction assumption as in Section A.1, namely, that for all view numbers $v' \in [0, v-1]$ included, any value y for which a lock certificate was formed in view v': $\mathsf{lock\ cert}[v', y]$, must verify the safety invariant coined in Equation (1) as safe(v',y). Recall that this means that no other value than, possibly, y, could have been committed in any view up to v'. We now show that the induction assumption holds for view v. Assume by contradiction that a $\mathsf{lock\ cert}[v, x]$ for some value is formed, and that a conflicting value y was committed in a previous view $v' < v$ (the supermajority needed to make a lock certificate rules out that y could have been committed in v). But for the $\mathsf{lock\ cert}[v, x]$ to be formed, one of the following two (non-exclusive) situations must have happened. Consider the set of the honest players which cast a $\mathsf{lock\ vote}[v, x]$ (so there are at least $t+1$ of them). Denote as $\mathsf{lock\ cert}[v_1, x]$ the lock certificate included in the prepare message of the leader L_v.

Either at least one of them was unlocked by a PnS. Then we are back in the situation of the proof of Section A.1, so this cannot happen (Recall that this player must have received the $\mathsf{lock\ cert}[v_1, x]$ and a valid $\mathsf{PnS}(v, v_1)$, but we showed that the simultaneous existence of those two objects was impossible under our contradictory assumption).

Or all of them voted following the Tendermint unlocking.
This means that, at the beginning of v, at least $t+1$ honest players had a locked view number no higher than v_1. This implies that *no value could have been* committed *in a view at least as high as* v_1 (because less than t honest

players could have cast a commit vote in such a higher view). So we must have that $v' < v_1$, so this violates the induction assumption for v_1.

C Faster proofs of possession (in the AGM)

We finally address simultaneously the two apparent issues mentionned in the (4) at the end of the Introduction. In [BCK+22], Fig. 11, they formalize the following *schnorr-knowledge-of-exponent* (schnorr-koe) assumption. Consider an abelian group $\mathbb{G}$ of some prime order q, with a public generator G. This assumption states existence of an extractor Ext such that if an adversary is able to produce a public key $\mathsf{pk} \in \mathbb{G}$ and a Schnorr signature on pk, then except with negligible probability $\mathsf{negl}(\kappa)$, Ext recovers the exponent: $\mathsf{sk} \in \mathbb{Z}/q\mathbb{Z}$ such that $\mathsf{pk} = \mathsf{sk}.G$, in straight-line from the random oracle queries of the adversary. It is proven, independently in both [FPS20, Thm 1] and [BCK+22, Thm 8], that schnorr-koe holds (tightly) under the AGM and the DL assumptions for $\mathbb{G}$. Hence, our fix for both issues in (4) is as follows. Instead of requiring the proofs-of-possession (PoP) of [RY07], we require PoP's consisting of Schnorr signatures on their key(s) by players. We call as $\mathcal{MSP}$-skoe the multisignature scheme obtained. We believe it to be practical, since Schnorr signatures are now standard (as EdDSA in TLS 1.3, and in Bitcoin since 2022). Even when $t \sim N/3$ machines are controled by the adversary, then Ext recovers all their secret keys except with negligible $t.\mathsf{negl}$ probability. Without both assumptions that Ext is straight-line and with negligible probability of failure, then no better reduction is known than with an exponential loss in N ([SG02] and [MOR01, Problems 4]). Given the keys extracted from the adversary, then the same trivial argument as in [Bol03, Thm 4.2] provides a tight reduction: from the security of BLS multisignatures in the $(N-t)$-users setting, to the one of standalone BLS signatures in the $(N-t)$ users setting. This concludes our $(N-t)$-users security claim, since the security of the latter reduces to the single-user setting with at most $(N-t)$ overhead. This is proven in [Ber15; Lac18]). As for the performance issue, Schnorr signatures have the advantage not to require pairings. Concretely, to verify one Schnorr signature $\sigma = (R, z)$ on pk, the verification consists in computing the hash $c = H(\mathsf{pk}, \mathsf{pk}, R)$ then testing if $z.G = R + c.\mathsf{pk}$. Then, testing N such equalities can be done all at once, by testing their linear combination with powers of a random number in $\mathbb{Z}/q\mathbb{Z}$ ([BGR98]).The cost of such a N-sized linear combination, known as a *multi-scalar multiplication*, is highly amortized in modern libraries such as the one we used.We also leveraged existence of a component in $\mathbb{G}_1$, pk' of each key. This allowed to make the Schnorr PoP instead in $\mathbb{G}_1$, and thus verified under pk'. Since multiscalar multiplications are 3x faster in $\mathbb{G}_1$ for thousands of keys, it is not surprising that this trick brought a further 2x speedup of the total time for the batch verification of the setup of $\mathcal{MSP}$-skoe.

We measured the computation done by each Verifier, at the setup, in three BLS-based multi-signatures: [BCG+23], [RY07] and the new $\mathcal{MSP}$-skoe. We report on them in Table 7. In all three, we incorporated the time taken by the verifier to check equalities of exponents in all published pairs of keys. This check for one key is $e(\mathsf{pk}', G_2) = e(G_1, \mathsf{pk})$, so we batched its verification in a total of 2 pairings. The verification of the PoP [RY07] was optimized as essentially a single product of N pairings with random exponents, using the batch verification of BLS signatures of [CHP07, §5.1]. In the table we also evaluated the marginal computation of the Verifier for every set of $2.\ell = 14$ keys published. This corresponds to the use-case of $\mathcal{M}\mathsf{to}\mathcal{A}$ for $v = 2^7$, when a new player registers a set of $2.\ell$ keys. For the scheme of [BCG+23], most of the computation of the Verifier is the same when one new key registers, as the one for the setup of N keys. More generally, each time the set of published keys is modified, the Verifier of [BCG+23] must replace, in its head, each published key pk_k by its multiplication by $H(\mathsf{pk}_k \| [\mathsf{pk}_1, \ldots, \mathsf{pk}_N])$. Of course, [BCG+23] have the size advantage not to require that keys come appended with a PoP (512 bits for our Schnorr one, and 256 for [RY07]). The latter are proven secure under the RMSS and AGM assumptions. Without these assumptions, then it is still possible to skip PoP's by using the previous scheme [BDN18]. This previous scheme comes at the slight overhead that the aggregation of k signatures and keys, cost two k-points multiplication and k hashes. Instead, this cost is just two k-points additions in [BCG+23] and PoP-based schemes ([RY07] and our $\mathcal{MSP}$-skoe).

Before we report on the performance, we briefly discuss the assumption. The schnorr-koe is folklore since [ZS92; Dam92]. *Avoiding* such kind of assumptions, i.e., that the adversary gives the exponent to the reduction (known as KOSK [Bol03]), is precisely what motivated the PoP of [RY07]. It turns out that the AGM is used in popular SNARK systems (Marlin [CHM+20], PlonK [GWC19] and Sonic [GKM+18; MBKM19]), as well as in DKG ([BCK+22]) and multisignatures ([NRS21; BD21; BCG+23]). Still, such assumptions are non-falsifiable. So we believe that the Schnorr PoP which we put forth (as $\mathcal{MSP}$-skoe) should be further composed with the Fischlin transform. Precisely, this

transform compiles a Schnorr proof into one with straight-line extractability, *without any overhead for the verifier*. Plus, its cost is now highly optimized [KS22].

BN254 curve	[BCG+23]	[RY07]	$\mathcal{MSP}$-skoe
Batch N keys	495.9 ms	582.1 ms	14.9 ms
Each $2.\ell$ keys	319.2 ms	5.5 ms	0.7 ms

BLS12-377 curve	[BCG+23]	[RY07]	$\mathcal{MSP}$-skoe
Batch N keys	1136.5 ms	1023.1 ms	24.1 ms
Each $2.\ell$ keys	860.0 ms	10.3 ms	1.7 ms

Table 7: One-shot computation time of the Verifier for each new list of N public keys, in three BLS multi-signature schemes. First line: for a complete new set of N keys. Second line: when $2.\ell$ new keys are added (in [BCG+23]: for any change in the set of keys). Times for $N = 4608$ public keys [this corresponds to the following choice of parameters in $\mathcal{M}\mathsf{to}\mathcal{A}$ (Section 7), where $N = 2\ell.n$: $\ell = 7$ bits of variable parts of messages, and $n = 193$ players ($= 3.64 + 1$)]. We neglected the time taken by the hashes, they will be incorporated in a future update of the code.

D An optimized version of BGLS.

Of independent interest, we introduce a class of non interactive aggregate signatures, in which the number of pairings is brought down to the number of distinct message contents. Roughly, it consists of the ones called AMSP and ASMP-pop in [BDN18, §3.3, §6.1], but in which signatures are *not* prefixed with sub-aggregate keys apk_i. More precisely:

- the signature is the vanilla BLS [Ben04; Dar10], in particular not prefixed with one's key.
- the aggregator, on input signed messages from a subset $I \subset [n]$ of issuers, gathers them by sub-subsets $\mathcal{J}_i$ of identical contents m_i. For each distinct content m_i, it applies the AGGREGATE$^=$ algorithm desribed in their blogpost [DN18] (multi-scalar multiplication), resp. of [RY07] (multi-point addition). The former aggregation is a multi-scalar multiplication of the signatures $(\sigma_j)_{j \in \mathcal{J}_i}$ by the hashes $\left\{ H(\mathsf{pk}_j \| (\mathsf{pk}'_j)_{j \in \mathcal{J}_i}) \ : \ \forall j \in \mathcal{J}_i \right\}$. Notice that in the proceedings version [BDN18, §3.1], these multiplications are instead done by the signers in $\mathcal{J}_i$. But we cannot require this, since this is not compatible with our syntax of NI-TSS (members of a quorum of signers are not aware of each other). So the aggregator obtains a multisignature on each content m_i, with respect to a (sub-)aggregate key apk_i.
- Finally, it applies the aggregation of [BGLS03][BDN18, §3.3] (add the multi signatures).
- The verification algorithm consists in verfying the all m_i's are distinct, then verifying the multisignatures on them, then apply the BGLS verification formula w.r.t. the sub-aggregate keys apk_i, as in [BDN18, §3.3].

Hence, the verification of each subset of signatures over the same message content, collapses to one pairing. This is why in Table 3, we count that verification of their aggregate signature requires $\min(k+1, v+1)$ pairings, where v is the number of possible distinct message contents. Notice that in the schemes presented as AMSP and ASMP-pop in [BDN18, §3.3, §6.1], the messages were prefixed with sub-aggregate keys. But as noticed by the authors in their §3.3, this prefix serves only to guarantee that the messages are distinct. Indeed, their reduction to the security of the baseline multisignature scheme, holds as soon as the messages are distinct. This reduction is a direct adaptation of the one of [BNN07], of which a variant for Type III pairings can be found in [Lac18, Thm 2]. We further observe that the same construction applies when using instead the multi-signature scheme of [BCG+23] as baseline (and also the one with our Schnorr PoP, in Appendix C).