GearBox: An Efficient UC Sharded Ledger Leveraging the Safety-Liveness Dichotomy

Bernardo David* [1], Bernardo Magri[2], Christian Matt[3], Jesper Buus Nielsen†[2], and Daniel Tschudi[3]

[1]ITU Copenhagen, bernardo@bmdavid.com
[2]Concordium Blockchain Research Center, Aarhus University, {magri, jbn}@cs.au.dk
[3]Concordium, Zürich, {cm, dt}@concordium.com

February 26, 2021

Abstract

Sharding is an emerging technique to overcome scalability issues on blockchain based public ledgers. Without sharding, every node in the network has to listen to and process all ledger protocol messages. The basic idea of sharding is to parallelize the ledger protocol: the nodes are divided into smaller subsets that each take care of a fraction of the original load by executing lighter instances of the ledger protocol, also called shards. The smaller the shards, the higher the efficiency, as by increasing parallelism there is less overhead in the shard consensus.

In this vein, we propose a novel approach that leverages the sharding safety-liveness dichotomy. We separate the liveness and safety in shard consensus, allowing us to dynamically tune shard parameters to achieve essentially optimal efficiency for the current corruption ratio of the system. We start by sampling a relatively small shard (possibly with a small honesty ratio), and we carefully trade-off safety for liveness in the consensus mechanism to tolerate small honesty without losing safety. However, for a shard to be live, a higher honesty ratio is required in the worst case. To detect liveness failures, we use a so-called control chain that is always live and safe. Shards that are detected to be not live are resampled with increased shard size and liveness tolerance until they are live, ensuring that all shards are always safe and run with optimal efficiency. As a concrete example, considering a population of 10K parties, 30% corruption and 60-bit security, our design permits shards of size 200 parties in contrast to 6K parties in previous designs.

Moreover, in this highly concurrent execution setting, it is paramount to guarantee that both the sharded ledger protocol and its sub protocols (e.g., the shards) are secure under composition. To prove the security of our approach, we present ideal functionalities capturing a sharded ledger as well as ideal functionalities capturing the control chain and individual shard consensus, which needs adjustable liveness. We further formalize our protocols and prove that they securely realize the sharded ledger functionality in the UC framework.

*This work was supported by a grant from Concordium Foundation and by Independent Research Fund Denmark grants number 9040-00399B (TrA^{2}C) and number 9131-00075B (PUMA).

†Partially funded by The Concordium Foundation; The Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE); The Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM).

Contents

1 Introduction **3**
- 1.1 Related Work . . . 5

2 Preliminaries **6**
- 2.1 Security Model . . . 6
- 2.2 Hash Functions . . . 7

3 Sharded Ledger Ideal Functionalities **7**
- 3.1 Timed Ledger (Control Chain) . . . 7
- 3.2 Shards . . . 9
- 3.3 Sharded Ledger . . . 11
- 3.4 Data Repository . . . 13

4 Realizing the Timed Ledger and the Shard Consensus **14**
- 4.1 Timed Ledger . . . 14
- 4.2 Shard Consensus . . . 15

5 Committee Selection **19**
- 5.1 Ideal Committee-Selection Functionality . . . 19
- 5.2 Randomness Beacon . . . 20
- 5.3 Realizing Committee Selection Using Randomness Beacon . . . 20
- 5.4 Determining the Committee Size . . . 21
- 5.5 Tight Analytic Bounds . . . 21
- 5.6 Alternative Ways to Realize Committee Selection . . . 24

6 Constructing a Sharded Ledger **24**
- 6.1 The Sharded Ledger Protocol $\Pi_{\text{BD-STL}}$ . . . 25
- 6.2 Extensions . . . 29
- 6.3 Inter-Shard Transactions and Communication . . . 29

7 Shard Safety-Liveness Dichotomies **31**
- 7.1 Synchronous, Unauthenticated SSLD . . . 31
- 7.2 Synchronous, Authenticated SSLD . . . 32
- 7.3 Partially Synchronous, Authenticated SSLD . . . 34

A Python Code for Computing Minimal Committee Sizes **38**

1 Introduction

Since the introduction of Bitcoin [25], there has been an explosion of interest in blockchains, both in research and practice. One of the biggest practical obstacles for the large-scale adoption of public blockchain systems that have been identified is the low throughput of transactions in systems such as Bitcoin. As a solution to overcome this limited scalability, a method called *sharding* has been proposed.

The basic idea of sharding is to parallelize the execution by dividing the network into smaller components, called shards. The smaller the shards are, the higher the efficiency is due to increasing parallelism and less overhead in the shard consensus. However, the security of the shards requires its size to be big, or analogously, small shards have lower security. For example, assuming at most 30% corruption[1] overall and a total population of 10K parties, the minimal shard size that guarantees at most 33% corruption with probability $1 - 2^{-60}$ in a randomly selected shard is over 6K, which hardly leads to a major improvement in efficiency. As we show later, the bounds are almost perfectly linear in the security parameter, so for 30-bit security the size would have to be about 3K. Thus, to get to a shard size in the hundreds, unsatisfying security would have to be adopted.

Existing sharding protocols suffer from this issue, because to get security in a partially synchronous network, at most 33% corruption can be tolerated. We overcome this apparent barrier by observing that security of a blockchain consists of two main properties: (1) Liveness says that the blockchain will eventually output new messages to all peers, and (2) safety says that the peers agree on the sequence of messages being output. The liveness threshold L is the level of corruption under which liveness is guaranteed. The safety threshold S is the level of corruption under which safety is guaranteed. Existing sharding solutions consider equal bounds $L = S$, which requires $L, S < 1/3$. In this work, we leverage what we call the *safety-liveness dichotomy* by considering different bounds for L and S.

Shard safety-liveness dichotomy. For partially synchronous protocols, the safety-liveness dichotomy says that $2L + S < 100\%$; whenever $S = L$ it must be that $L, S < 33\%$. For synchronous protocols the safety-liveness dichotomy says that $L + S < 100\%$; whenever $S = L$ it must be that $L, S < 50\%$. The shard dichotomies follow using standard arguments. We provide formal proofs in Section 7. Note that the synchronous dichotomy $L + S < 100\%$ seems to contradict the Dolev-Strong protocol from [14] which achieves synchronous broadcast for, e.g., $L = S = 99\%$. However, Dolev-Strong only achieves *internal* agreement among the n servers. External parties cannot verify the value agreed on. The crucial thing we use about a shard to prove the dichotomies is that external parties can post on it and read from it. If a reader can make a decision while a subset of (relative) size L is crashed, it means that a subset of size $1 - L$ can convince the reader. If $S = 1 - L$ this means it can be convinced after only talking to a set of potentially corrupt servers, which roughly gives the synchronous dichotomy. For the partially synchronous dichotomy we use the fact that if the unknown network delay is large enough, a reader cannot distinguish a corrupt subset of (relative) size L from a slow and honest one. And if a subset of size L is crashed, it must be definition be able to make a decision. Therefore a reader which makes a decision (say, on which message it saw on the shard first) could sometimes do this without having heard from a fraction L of the honest parties. This means that it only heard from a fraction $L + S$ of the parties and out of these a fraction L or fraction S might be

[1] In blockchains, the corruption bounds are typically weighted by some resource, e.g., computing power for proof-of-work systems, or stake in proof-of-stake systems. To simplify the presentation, we ignore this weighting in the introduction. We stress, however, that our results are not limited to this simplified setting and can indeed be used in a weighted setting. See Section 5.3 for a further discussion.

corrupted. Therefore reasoning similar to that behind the synchronous dichotomy on this "live" set of servers will give the lower bound.

Our contributions. We present a novel sharding approach that leverages the safety-liveness dichotomy to get the smallest possible shards, and therefore optimal efficiency, without sacrificing security. Our sharding design has security against $t < 1/3$ in the partially synchronous setting. The same technique can be used to get security against $t < 1/2$ corruption ratio in the synchronous network setting, but we focus on the partially synchronous case, which appear more desirable for a distributed sharding protocol.

We use a "two-layer" approach with a control-chain (CC) which is used to manage several shards. The shards will, in the optimistic model, run with a low liveness threshold and a high safety threshold, for instance $S = 89\%$ and $L = 5\%$. Being safe against 89% corrupt parties allows to sample a much smaller committee; however, being live only against 5% corruption makes it more likely for a shard to deadlock.

To mitigate this issue, the CC constantly monitors shards for liveness. This is done by letting the shards post "heart beats" on the CC. The CC can take down a deadlocked shard and spin up a new shard with a new random committee and a higher liveness threshold (and a lower safety threshold), leading to bigger shards. This can be iterated until a shard size is found which gives both safety and liveness. Crucially, at no point safety is compromised.

Our design has for each shard a "gearbox" of consensus protocols: shards at the top are slower but have robust liveness, shards at the bottom are fast but have a lower liveness. The CC changes gear upwards in the gearbox when deadlocks are detected, and can over time change gear downwards when there is no signs of an attack. At the top of the gearbox the gear cannot change upwards, so a deadlocked shard is restarted with the top consensus algorithm. The only requirement to get eventual liveness is that, when the top consensus algorithm is instantiated with a random committee, it happens with constant probability that the corruption threshold is low enough to get liveness. Our design allows to run with dramatically smaller shards when not under a large attack. For example, with the same 10K total parties and 30% overall corruption, a shard with only 200 parties already guarantees less than 60% corruption in the shard with probability $1 - 2^{-60}$. The above allows for a wide range of designs. One could even switch from the partially synchronous to the synchronous model within the gearbox and thus tolerate higher corruption thresholds. Next, we describe two different ways to instantiate our framework.

Partially synchronous instantiation. One can run with a partially synchronous CC and partially synchronous shards. At the bottom gear one could have $L = 5\%$ and $S = 89\%$. For a population of 10K peers and assuming corruption level at most 30% this would give a committee size of 51. At the top gear one could use a consensus protocol with $L = 30\%$ and $S = 39\%$. This would give a committee size of 1713 to not get more than 39% corruption (with 60-bit security). Note that we sample from a ground population with corruption at most 30% and need a committee with corruption at most 30% for liveness. It can be seen that we get less than 30% corruption with a constant probability, what gives eventual liveness.

This shows another advantages of our framework: It is possible to set the liveness threshold of the committees to be the same as of the ground population (30% in this example). This is impossible when requiring that liveness only fails with negligible probability. For our design, it is enough to have liveness with constant positive probability because shards that are not live simply get restarted.

Mixed partially synchronous and synchronous instantiation. Another instantiation is to run with a synchronous CC tolerating 49% corruption. At the bottom of the gearbox one

could again start with a partially synchronous shard with $L = 5\%$ and $S = 89\%$. One can run partially synchronous up until $L = 25\%$ and $S = 49\%$. After that one could then switch to a synchronous shard with $S = L = 49\%$ corruption. This allows an overall design tolerating 49% corruption, but running partially synchronous with small shards until 25% corruption. This is interesting because partially synchronous protocols can achieve higher throughput in good network conditions by avoiding to wait for the end of the round, as synchronous protocol do.

In the rest of the paper we focus on the partially synchronous setting, and therefore we are "stuck" with the $2L + S < 100\%$ dichotomy. Thus we need less than 33% corruption in the ground population to get safety and liveness of the CC. In most of our examples when we compute concrete committee sizes we will assume 30% corruption. Computing similar numbers for the case of a synchronous CC is straightforward.

UC formalization. To prove the security of our approach, we formalize an ideal functionality capturing a sharded ledger as well as functionalities capturing the consensus guarantees we require from the control chain and from the shards, which need to have adjustable liveness in our approach. We build on these functionalities to construct our sharded ledger protocol, which we prove to UC-realize the sharded ledger functionality. To the best of our knowledge, ours is the first sharded ledger protocol to achieve security under arbitrary composition, which is an extremely important property in settings where a number of protocols are executed in parallel (e.g., blockchains). Moreover, we introduce and model the concept of timed ledgers, which go beyond guaranteeing that messages recorded on the ledger remain ordered in a certain way, also allowing parties to obtain explicit timestamps for messages.

1.1 Related Work

In the last few years, many shard-based blockchain protocols have been proposed by the scientific community and by the industry in the form of whitepapers. Most of the proposals by the industry, despite many containing nice ideas and innovations, follow an heuristic approach, where no formal security guarantees are proposed or formally proven. Thus, in this section we only discuss a few of the most well-known (peer-reviewed) sharding protocol proposals, and we point out a common issue with all the proposals that hinders their practical usage.

Sharding protocols. To the best of our knowledge, Elastico [24] is the first sharding protocol proposed for public blockchains. The protocol is synchronous and runs in "epochs"; in every epoch each party solves a PoW puzzle based on randomness obtained from the previous epoch. The PoW's least-significant bits are used to form the committees that will run each shard and process the transactions. Even though the authors of [24] advocate for a small committee size per shard (around 100 parties), the probability of a shard being unsafe gets very high, close to 97%, after only six epochs, as shown in [21]. This renders the protocol completely insecure when used with small committees.

Building upon Elastico's ideas, and improving it in many ways, OmniLedger [21] is a sharding protocol that generates identities and assigns participants to shard committees using a synchronous PoW independent identity-blockchain. However, like Elastico, OmniLedger can only tolerate up to $t < n/4$ corruptions on the total number of parties in the system. During each epoch, new randomness is generated for a leader election lottery. The protocol can achieve low latency for the confirmation of transactions whenever $t < n/8$.

RapidChain [29] is a synchronous sharding protocol that tolerates up to $n/3$ corrupt parties out of the total number of participants. The protocol is bootstrapped by a committee election protocol that initially selects a reference committee of size $m = O(\log n)$. At the end of every

epoch, the reference committee is responsible for generating fresh randomness that will be used to select the committees for all the shards at the end of the *first* epoch, and to reconfigure the committees of existing shards in subsequent epochs.

Using an approach closely related to sharding, Monoxide [28] proposes a scale-out blockchain that contains many independent chains (called zones) running in parallel that divides the workload of the entire system; communication, computation and storage is shared among the different zones, making the burden of maintaining the entire system shared among the nodes running each zone. When a "cross-zone" transaction happens, an eventual atomicity technique is used in order to keep consistency among the different zones.

The work of Avarikioti et al. [1] proposes a framework with security properties tailored for sharded ledger protocols, building upon the Bitcoin backbone model of Garay et al. [15]. More specifically, the authors propose the novel notions of *consistency* and *scalability* for sharded ledgers that intuitively says that, cross-shard transactions must preserve safety and sharded systems must gain some speed-up in comparison to a non-sharded system, respectively. Moreover, the authors analyze many existing sharded ledger protocols in their model and prove if the protocol satisfy the proposed definition or not. Unfortunately, the model proposed in [1] is not composable, making it difficult to argue security of the sharded ledger protocol when combining it with a larger system. We refer the interested reader to the work of Wang et al. [27] that gives a nice overview of the state-of-the-art in sharding protocols.

Common issue. A common factor in all the previously described sharding protocols is that, for a robust security parameter, the size of the shard's committee needs to be large in order to guarantee the safety properties for each shard. In Section 5.4 we present some concrete numbers for the smallest size of committees needed to be sampled from a population of 10K parties with 30% corruption considering 60-bits of security; for honest majority (49% corruption) a committee of at least 462 parties is necessary, while for honest supermajority (33% corruption) a committee of around 6.3K parties is needed.

2 Preliminaries

We denote by P a party in the party set $\mathcal{P}$. We denote by $\mathsf{Honest} \subseteq \mathcal{P}$ the set of honest parties during the protocol execution.

2.1 Security Model

Since our protocols make essential use of time, we need a notion of UC security for (partial) synchronous protocols. We thus need to assume that parties have access to a reliable network functionality with bounded delay Δ_{NET}, similar to the functionality $\mathcal{F}_{\text{N-MC}}^{\Delta_{\text{NET}}}$ in [3]. We further need a notion of time and access to clocks [18], and we assume an idealized signature functionality [6, 2]. To keep the presentation simple, we do not model all these functionalities and refer to the cited papers for UC-related details.

Time. The functionality $\mathcal{F}_{\text{CLOCK}}$ essentially amounts to assuming perfectly synchronised discrete clocks. We use *time slot* to denote the time period between two ticks of the clock $\mathcal{F}_{\text{CLOCK}}$. We use slot length to denote the length of time slots and we assume it to be fixed. In a slight abuse of notation, we also sometimes call a time slot a tick. By tick r we mean the time slot starting after the clock ticked r times. We assume time starts with a clock tick, so the first tick is tick 1. The execution proceeds in a way such that if honest party P_i is in tick r_i and honest party P_j is in tick r_j, then $|r_i - r_j| \leq 1$. We assume that each party has enough computational

power to complete arbitrary polynomial time computations in each time slot. Note that the previous simplification may not be a good model of reality, but by assuming a known upper bound on the clock drift in "real life", and designing our protocols such that a bounded number of computation is required in each round, setting the slot length long enough, and assuming an upper bound on the message delivery, then the model can clearly be realised. We stress that our goal is to communicate our sharding mechanism, which we believe is best done in a simple model. Implementing the protocol securely in practice is a highly non-trivial but largely orthogonal task.

Network. The network $\mathcal{F}_{\text{N-MC}}^{\Delta_{\text{NET}}}$ allows parties to multi-cast messages. The adversary determines when messages are output at honest parties, but the delay can be at most Δ_{NET} ticks. The bound Δ_{NET} is not known to honest parties making this a partially synchronous communication model.

Static vs adaptive adversaries. We consider the set of corrupted parties to be static. The reason is that our protocols rely on committees of parties, and therefore an adaptive adversary could easily identify the parties in the committee and corrupt them. Note that this is an inherent problem with protocols based on committees, and in particular all previous works that rely on committees face the same issue. However, *it is possible* to extend our protocols to allow for an adaptive adversary that can change corruptions *after a delay*. In that case, the protocols described in this paper would have to be adapted to resample the committees before the adversary is allowed to change the corrupted parties, see Section 6.2 for more details.

2.2 Hash Functions

We denote by $H\colon \{0,1\}^* \to \{0,1\}^\kappa$ a collision-resistant hash function. For a vector $\vec{m}$, we recursively define $\vec{H}$ as

$$\vec{H}(m_1) \coloneqq H(m_1),$$
$$\vec{H}(m_1, \ldots, m_{\ell+1}) \coloneqq H(\vec{H}(m_1, \ldots, m_\ell) \,||\, m_{\ell+1}).$$

3 Sharded Ledger Ideal Functionalities

In this section we describe all the ideal functionalities necessary for our sharded ledger model.

3.1 Timed Ledger (Control Chain)

We now formalize a simple ledger functionality which will act as the so-called control chain (CC) orchestrating the shards. We model the ledger as a persistent, timestamped, bounded-delay, totally-ordered broadcast channel. The ledger is a high-level abstraction that gets rid of many details that are irrelevant for our purposes. It is thus much simpler than, e.g., the ledger functionality in [3].

The timestamp property ensure the messages get timestamps on which the parties will agree on. We call these timestamps the *ledger arrival time*. These are important in defining precisely whether a message made it before a timeout. This is in particular important for a party that views messages on the ledger long after they were added. Note that a party has no other form to associate the order of the messages on the ledger with a physical time if the messages do not carry the timestamp information.

Moreover, the ledger arrival time may differ from the time the message was input (by an honest party) or the time the message is delivered locally to an honest party. However, the

bounded delay property ensures that those differences in time are bounded by Δ (similar to the delivery time in $\mathcal{F}_{\text{N-MC}}^{\Delta_{\text{NET}}}$). In other words, input messages are timestamped and delivered within some bounded time.

Functionality Bounded Delay, Timed Ledger $\mathcal{F}_{\text{BD-TL}}^{\Delta}$

Initialization

All $P_i \in \mathcal{P}$ set $\mathsf{TO}_i := ()$.

Interface for party $P_i \in \mathcal{P}$

Input: $(\textsc{Send}, m)$
Output $(\textsc{Send}, P_i, m)$ to the adversary.[a]

Input: $(\textsc{Get})$
Output $(\textsc{Get}, P_i)$ to the adversary.
Return TO_i.

Interface for adversary

Input: $(\textsc{Add}, P_i, m, t)$
Append (m, t) to TO_i. // Unless it violates the below restrictions

At any time, $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ automatically enforces these restrictions: [b]

Persistence: If P_i and P_j are honest, then TO_i and TO_j are order consistent, i.e., either TO_i is a prefix of TO_j or TO_j is a prefix of TO_i.

Liveness and bounded timestamps: For all messages m that are input at some time t (via $(\textsc{Send}, m)$) by an honest party for the first time, there is a $t' \leq t + \Delta$ such that by time $t' + \Delta$, we have $(m, t') \in \mathsf{TO}_i$ for all honest P_i.[c]

[a] Whenever we write that we output something for the adversary we mean that we leave the message in the ideal functionality and next time the adversary queries the ideal functionality it can pick up the message. This is to not turn over the activation to the adversary.

[b] The enforcement is by dropping ADD commands violating any restriction and by executing extra ADD commands when required to meet any restriction.

[c] We don't give any guarantees for subsequent inputs of the same message if it is input multiple times. Also note that messages could get timestamps before input by an honest party if dishonest parties input it before.

Implementation. Our ledger functionality can be implemented by a BFT consensus blockchain such as Algorand [9] or Tendermint [22, 4], or by a Nakamoto-style blockchain such as Bitcoin [25]. In case a Nakamoto-style blockchain is used, combining it with a finality layer, e.g., Casper the Friendly Finality Gadget [5] or Afgjort [13], is advisable to improve the finality time as discussed below. See Section 4.1 for more details on how to implement $\mathcal{F}_{\text{BD-TL}}^{\Delta}$.

Discussion. For the reader used to the formalization of a blockchain via a chain of blocks and properties like common prefix, chain quality and chain growth as in [16] the formalization via $\mathcal{F}_{\text{BD-TL}}$ might look overly simplified. That is meant as a feature. No matter which blockchain

is used to realize $\mathcal{F}_{\text{BD-TL}}$ most applications typically only depend on the messages in finalized blocks for safety and therefore relies mostly on the time it takes until finalization for liveness. We therefore conjecture that $\mathcal{F}_{\text{BD-TL}}$ allows to analyse the vast majority of secure applications of blockchain. Second, $\mathcal{F}_{\text{BD-TL}}$ captures the common API and guarantees of Nakamoto style blockchains, like Bitcoin and Ouroboros, BFT style blockchains like Algorand [9] and hybrid blockchains like Concordium [13]. Therefore a proof of an application using $\mathcal{F}_{\text{BD-TL}}$ allows it to be deployed on a wide range of blockchains. If one has an application which uses messages before they are final or need whitebox access to the blocks of the chain, then a formalization like [16] is more suited.

3.2 Shards

In essence, a shard is just a ledger. However, we need to be explicit about the fact that its properties may be violated if it is instantiated with a committee that is too small or with a corruption ratio that is too high. The functionality is parameterized by a set $\mathcal{C}$ of size n corresponding to a committee of parties that will be in charge of maintaining the shard, the sets $\mathcal{S}$ and $\mathcal{L}$ that represent adversary structures for safety and liveness respectively, and a delay Δ. The adversary structures $\mathcal{S}$ and $\mathcal{L}$ simply mean that the shard functionality must maintain its safety when the set of corrupted parties is in $\mathcal{S}$ and it must maintain its liveness when the set of corrupted parties is in $\mathcal{L}$; depending on whether the functionality is safe and/or live, some properties are guaranteed. The committee running the shard will be given the parties as a vector $\mathcal{C} = (P_1, \ldots, P_n)$ to allow to match the committee members to the adversary structure, but we often abuse notation and refer to $\mathcal{C}$ as a set.

The parties $P_i \in \mathcal{C}$ can interact with the shard functionality by sending transactions to the shard through the Send command, and retrieve the ledger through the Get command. The parties can also "close" the shard by issuing the Close command; looking ahead, this is useful when the sharded ledger protocol (Section 6.1) requests parties to shut down a shard in order to start a new shard with different parameters and parties. Moreover, the parties can request "finality proofs" to the functionality through the GetFinProof command. This proof can then be verified by *any* party (including external parties not in $\mathcal{C}$). Our functionality offers two ways to verify such proofs: Using VerifyFinProof, one can verify a proof relative to a message vector $\vec{m}$, i.e., it can verify that the messages in $\vec{m}$ are finalized in the ledger. Alternatively, external parties can use VerifyFinLength to verify a proof relative to an integer ℓ, i.e., to simply verify that *at least* ℓ messages have been finalized so far; this allows to check liveness by ensuring a growing ℓ without needing to know the actual messages.

We impose some restrictions on the adversary in the form of properties of the shard functionality.[2] The persistence property is the standard property that one expects from a ledger, i.e., intuitively all honest parties will maintain ledgers that are prefixes of each other. We formalize this by considering a global FTO and guaranteeing that the ledgers of all honest parties are prefixes of FTO. The liveness property is also standard and says that any message sent by an honest party will make it into the ledger of all honest parties after at most Δ time.[3]

The novel properties that we require for our shard functionality are called *censorship resilience*

[2]The properties just mean that the adversary is allowed to give inputs to the ideal functionality as it desires, but the ideal functionality ignores inputs that would violate a safety property. And, if the adversary at some point would have to give a particular command, like delivering a given message, to not break a liveness property, then the ideal functionality will enforce execution of this command to maintain the liveness property. The properties could, as is more usual, be expanded to more verbose pseudo-code for the ideal functionality doing these enforcements explicitly, but we prefer the more compact property-based specifications.

[3]Note that the delay Δ is a parameter of the functionality, but it may not be known to the honest parties. This is in particular the case when considering the partially synchronous model.

and *proof soundness*, and we require them to hold only when the ledger is safe, i.e., when the set of corrupted parties is in $\mathcal{S}$. Censorship resilience prevents an adversary to exclude specific messages from the ledger. We formalize this as the guarantee that when a party is sending a message for inclusion, it will be included at most two ledger updates later.[4] In other words, either the message gets included, or the ledger stops completely. This is useful because a complete halt can be detected from the outside. The proof soundness property intuitively says that finality proofs cannot be forged. That is, $(\textsc{VerifyFinProof}, \vec{m}, \pi)$ only returns 1 if $\vec{m}$ is a prefix of FTO, and $(\textsc{VerifyFinLength}, \ell, \pi)$ only returns 1, if $\left|\mathsf{FTO}\right| \geq \ell$. We formally define the shard functionality next and in Section 4.2 we show a protocol that UC-realizes this functionality.

Functionality $\mathcal{F}_{\textsc{Shard}}^{\mathcal{C},\mathcal{S},\mathcal{L},\Delta}$

Initialization

Set $\mathsf{FTO} = ()$ and $\mathsf{TO}_i := ()$ for all $P_i \in \mathcal{C}$.

Interface for party $P_i \in \mathcal{C}$

Input: $(\textsc{Send}, m)$
Send $(\textsc{Send}, P_i, m)$ to the adversary.

Input: $\textsc{Get}$
Send $(\textsc{Get}, P_i)$ to the adversary.
return TO_i

Input: $\textsc{Close}$
Send $(\textsc{Close}, P_i)$ to the adversary.

Input: $\textsc{GetFinProof}$
Send $(\textsc{GetFinProof}, P_i)$ to the adversary,
who immediately sends back a proof π such that no record $(\mathsf{TO}_i, \pi, 0)$ has been stored.
Store the record $(\mathsf{TO}_i, \pi, 1)$
return (TO_i, π)

Interface for adversary

Input: $(\textsc{Add}, m, i)$
Append m to TO_i.
Input: $(\textsc{AddFinal}, m)$
Append m to FTO.

Public interface // **Any (even "outside") party can use this interface.**

Input: $(\textsc{VerifyFinProof}, \vec{m}, \pi)$
if record $(\vec{m}, \pi, b)$ for some $b \in \{0, 1\}$ exists **then**
return b
else
Send $(\textsc{VerifyFinProof}, \vec{m}, \pi)$ to adversary, who immediately replies with $b \in \{0, 1\}$.
Store the record $(\vec{m}, \pi, b)$.
return b

[4] When the ledger is realized using blocks, this means the block after the next block must include this message. One could more generally also allow for larger delays than two blocks, but we here avoid the extra parameter.

end if

Input: $(\text{VERIFYFINLENGTH}, \ell, \pi)$ // Only verify length ℓ of message vector
if record $(\vec{m}, \pi, b)$ for some $\vec{m}$ with $|\vec{m}| = \ell$ and for some $b \in \{0,1\}$ exists **then**
return b
else
Send $(\text{VERIFYFINLENGTH}, \ell, \pi)$ to the adversary.
Adversary immediately replies with $b \in \{0,1\}$ and $\vec{m}$ with $|\vec{m}| = \ell$.
Store the record $(\vec{m}, \pi, b)$.
return b
end if

At any time, the functionality automatically enforces the following:

Let $A \subset \mathcal{C}$ be the corrupted parties in $\mathcal{C}$. We call the ledger *live* if $A \in \mathcal{L}$. If $A \in \mathcal{S}$ call the ledger *safe*. Call the ledger *weakly closed* if some honest party input CLOSE.

Persistence: (*If the ledger is safe*) For all $P_i \in \mathcal{C} \setminus A$, TO_i is a prefix of FTO.

Liveness: (*If the ledger is live and not weakly closed*) After a message m was input (via (SEND, m)) by an honest party for the first time,[a] we have $m \in \mathsf{TO}_i$ for all $P_i \in \mathcal{C} \setminus A$ at most Δ time later.

Censorship Resilience: (*If the ledger is safe*) After a message m was input (via (SEND, m)) by an honest party at time t and TO_i for a honest P_i was updated twice after time $t + \Delta$, we have $m \in \mathsf{TO}_i$.

Proof Soundness: (*If the ledger is safe*) If $(\text{VERIFYFINPROOF}, \vec{m}, \pi)$ returns 1, then $\vec{m}$ is a prefix of FTO. If $(\text{VERIFYFINLENGTH}, \ell, \pi)$ returns 1, then $|\mathsf{FTO}| \geq \ell$.

[a] We don't give any guarantees for subsequent inputs of the same message if it is input multiple times.

3.3 Sharded Ledger

For simplicity, we describe a sharded ledger with a fixed set of shards; this will be implemented by starting and re-starting shards with varying liveness guarantees. The purpose of the sharded ledger is to hide these implementation details. Again for simplicity, we assume all shards running from the beginning of time and until the end of the execution. The sharded ledger can easily be modified to have commands for starting shards late and closing them early. However, this is easy to implement given the techniques for restarting deadlocked shards so we omit it here. In $\mathcal{F}^{\Delta}_{\text{BD-STL}}$, there are a number of shards uniquely identified by a value `sid`. Each party P_i can choose to add a shard identified by `sid` to its view, meaning it can read messages in this shard and see the total order of messages across all shards in its view.

Messages in shards will be ordered according to a total order given by the total order on the shard. These total orders are in particular strict parties orders. Messages across shards will be ordered according to a global strict partial order. This strict parties ordered is implemented by giving each message a Lamport clock. We describe it briefly here, and return in more detail when we discuss inter-shard communication in Section 6.3.

A shard Shard_i can contain a message $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ meaning m was sent to Shard_j. A shard Shard_j can contain a message $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ meaning m was received from Shard_i.

All messages will have Lamport clocks which are causally consistent. They increase within shards and $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ has a larger clock on the receiving shard than on the sending shard. We return to this in more detail later. This means that the local strict partial order of a Shard might have some out-going edges $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ where the endpoint is only visible on another shard.

For two such consistent strict partial orders $\mathsf{STO}_1 = \{(m_1^1, t_1^1), \ldots, (m_n^1, t_n^1)\}$ and $\mathsf{STO}_2 = \{(m_1^2, t_1^2), \ldots, (m_{n'}^2, t_{n'}^2)\}$ we let $\mathsf{STO}_1 \bowtie \mathsf{STO}_2$ be the zip of the two lists sorted in some canonical order based on the Lamport clocks with the following pruning. If one list contains a received message $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ and Shard_i is one of the shards being merged and $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ was not sent yet on Shard_i, then the shortest suffix of Shard_j is removed which removes the receive event $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$. Then the lists are merged according to their joint strict partial order. Next, we formally define the $\mathcal{F}_{\text{BD-STL}}^{\Delta}$ functionality.

Functionality Bounded Delay, Sharded Timed Ledger $\mathcal{F}_{\text{BD-STL}}^{\Delta}$

Initialization

for $P_i \in \mathcal{P}$ and all `sid` **do**
  $\mathsf{STO}_i^{\mathtt{sid}} \coloneqq ()$ // Shard Total Order of Shard `sid` for P_i
  $\mathsf{Sent}^{\mathtt{sid}} \coloneqq \varnothing$ // Messages sent to Shard `sid`
  $\mathsf{SHDS}_i \coloneqq \varnothing$ // Set of shards viewed by P_i
end for

Interface for party $P_i \in \mathcal{P}$

Input: $(\textsc{Send}, \mathtt{sid}, m)$ // Assume each $(\mathtt{sid}, m)$ is sent at most once per honest party.
  Output $(P_i, \mathtt{sid}, m)$ to the adversary. // Rushing.
  Add (P_i, m, t^{Now}) to $\mathsf{Sent}^{\mathtt{sid}}$.

Input: $(\textsc{Get})$.
  Output $(\textsc{Get}, P_i)$ to the adversary.
  Return $\bowtie_{\mathtt{sid} \in \mathsf{SHDS}_i} \mathsf{STO}_i^{\mathtt{sid}}$. // The strict partial order across shards in P_i's view.

Input: $(\textsc{AddShard}, \mathtt{sid})$.
  $\mathsf{SHDS}_i \coloneqq \mathsf{SHDS}_i \cup \{\mathtt{sid}\}$
  Wait until any element removed by pruning with the new shard in $\bowtie$ appears in $\bowtie_{\mathtt{sid} \in \mathsf{SHDS}_i} \mathsf{STO}_i^{\mathtt{sid}}$ again.

Input: $(\textsc{RemoveShard}, \mathtt{sid})$.
  $\mathsf{SHDS}_i \coloneqq \mathsf{SHDS}_i \setminus \{\mathtt{sid}\}$

Interface for adversary

Input: $(\textsc{Add}, \mathtt{sid}, m, t)$
  Append (m, t) to $\mathsf{STO}_i^{\mathtt{sid}}$. // Unless it violates the below restrictions

At any time, $\mathcal{F}_{\text{BD-STL}}^{\Delta}$ automatically enforces these restrictions: [a]

Persistence: If P_i and P_j are honest, then $\mathsf{STO}_i^{\mathtt{sid}}$ and $\mathsf{STO}_j^{\mathtt{sid}}$ are order consistent, i.e., either $\mathsf{STO}_i^{\mathtt{sid}}$ is a prefix of $\mathsf{STO}_j^{\mathtt{sid}}$ or $\mathsf{STO}_j^{\mathtt{sid}}$ is a prefix of $\mathsf{STO}_i^{\mathtt{sid}}$.

Liveness and bounded timestamps: For all messages $(\mathtt{sid}, m, t) \in \mathsf{Sent}^{\mathtt{sid}}$, there is a $t' \leq t + \Delta$ such that by time $t + \Delta$, we have $(m, t') \in \mathsf{STO}_i^{\mathtt{sid}}$ for all honest P_i.

[a]The enforcement is by dropping ADD commands violating any restriction, by executing extra ADD commands when required to meet any restriction.

The reason for the formulation of ADDSHARD and GET is to ensure the ledger time of any two ledgers consecutively gotten from $\mathcal{F}_{\text{BD-STL}}$ are not decreasing. This could have already outputted elements disappear again for a while and then popping up again.

3.4 Data Repository

We use an abstract notion of a public data repository called $\mathcal{F}_{\text{REPO}}$. Any party can store D under $h(D)$ and anyone can retrieve a stored D using $h(D)$. In real life some access control is needed to avoid denial of service attacks by flooding the storage. This is inconsequential to the main ideas that we want to present, so we do not clutter $\mathcal{F}_{\text{REPO}}$ with this. It has two adversarially defined delays, Δ_{STORE} is the time it takes to store D from when the first honest party tries and Δ_{GET} is the time it takes to get D given that it is stored.

Functionality Repository $\mathcal{F}_{\text{REPO}}$

Initialization

Initialize set of stored D to be empty.

Interface for party $P_i \in \mathcal{P}$

Input: (SEND, D)
Show D to the adversary Before time Δ_{STORE}, store D. Let the adversary specify the time.

Input: $(\text{GET}, h(D))$.
If D is stored such that $h = H(D)$ then output D. Let the adversary determine output time within delay Δ_{GET}.
If D is not stored then let the adversary determine whether a matching D should be outputted and when. There is no limit on the delivery time, but the adversary has to input D such that $h = H(D)$.

Interface for adversary

Can determine delivery times.

Here is a very inefficient example just for illustration. It uses n parties $P_1, \ldots, P_n$ for storing the files D. Assume that $t < n/2$ parties may be actively corrupted. To store D send it to all servers. The servers store all D and forward to the other server the first time it sees a new D. The parties only store one copy of each D. The storage is considered done when the last honest party got D. Here Δ_{STORE} is the network delay Δ_{NET}. The get D send h and have all servers holding a matching D send D. When receiving the first D matching h, output this D. Here Δ_{GET} is $2\Delta_{\text{NET}}$. Both the resilience and the communication can be optimised considerably.

4 Realizing the Timed Ledger and the Shard Consensus

In this section we give protocols to realize the timed ledger functionality $\mathcal{F}^{\Delta}_{\text{BD-TL}}$ and the shard functionality $\mathcal{F}_{\text{Shard}}$.

4.1 Timed Ledger

Following the analysis and the discussion in for instance [16] and in [11] it seems straight-forward that $\mathcal{F}^{\Delta}_{\text{BD-TL}}$ can be implemented using the Bitcoin protocol or the Ouroboros Praos blockchain under reasonable assumptions on *known* bounded network delay and traffic load, and assuming honest computational power respectively honest majority of stake.

Simply let the ledger arrival time be the time of the block the transaction appears in and output a message when it is in a block which is final. For Bitcoin finality is defined by the prefix property [11]. A message will appear in a blocks in reasonable time by chain quality plus chain growth and assuming that no more transactions are sent than can be put into blocks by the honest parties. In our setting, this should be the case as we will use $\mathcal{F}^{\Delta}_{\text{BD-TL}}$ as the Control Chain, where by design we can enforce that only control information from the sharding orchestration is posted, or that control information is given priority over normal payload. This allows to ensure by design the no more transactions are sent than can be posted.

If we want to achieve a truly partially synchronous $\mathcal{F}^{\Delta}_{\text{BD-TL}}$, i.e., where Δ is unknown to honest parties, one can use a BFT protocol like Algorand [9] which finalizes each block. This comes at the price that we need at least two thirds honesty overall. Alternatively, a hybrid blockchain like Concordium [13] with a Nakamoto style blockchain plus a BFT finality layer can be used. Here, a message is considered delivered if it is in a block marked as final.

In the following, we go into a bit more detail on how the Bitcoin protocol achieves $\mathcal{F}^{\Delta}_{\text{BD-TL}}$. According to the analysis in [16] Bitcoin satisfies the following properties:

Common-Prefix: There exists a $k \in \mathbb{N}$ (depending on the security parameter and the network delay) for any points in time $t_i \leq t_j$ and any pair of honest parties P_i, P_j with chains $\mathcal{C}_i$ resp. $\mathcal{C}_j$ they had a time t_i resp. t_j it holds that $\mathcal{C}_i$ with the last k blocks removed is a prefix of $\mathcal{C}_j$.

Chain-Growth: For any honest party the adapted chain grows over time.

Chain-Quality: There exists a $\ell \in \mathbb{N}$ and $0 < \mu \leq 1$ such that for any chain $\mathcal{C}$ of an honest party any ℓ consecutive blocks contain a μ fraction of honestly created blocks.

Given a bounded network delay, we can fix as part of the protocol a constant $k' \in \mathbb{N}$ such that for any honest party the k'-pruned chain (i.e., the chain with the last $k' \in \mathbb{N}$ blocks removed) is in the common-prefix. A message is considered *final* for party P if it is in this k'-pruned chain. We can also assume that parties add a timestamp to blocks they create where valid blocks have increasing time-stamps. The ledger arrival time of a message is then defined as the timestamp of the block that contains the message.

We can now argue that this achieves the ledger $\mathcal{F}^{\Delta}_{\text{BD-TL}}$ for some fixed Δ. First, we observe that *persistence* follows directly from the common-prefix property. For *liveness* and *bounded delay* we need to bound the time between message input and ledger arrival and between ledger arrival and message delivery. In the Bitcoin protocol input messages are flooded on the network. The bounded network delay ensures that exists an Δ_{net} such that an input arrives at all honest parties within Δ_{net} ticks. Once the message has been flooded it will be added latest to the next honest block (assuming not too many inputs or unlimited block size). Chain-growth and chain-quality ensure that an honest block is created within Δ_{add} ticks. Assuming that

parties do not accept blocks from the future (i.e., with a timestamp in the future), the time difference between input and ledger arrival is bounded by $\Delta_{net} + \Delta_{add}$. Bounded network delay, Chain-growth, and chain-quality ensure that within another Δ_{growth} ticks the message block is in the k'-pruned chain of every party. The difference between ledger arrival and message delivery is thus bounded by Δ_{growth}. For $\Delta = \max(\Delta_{net} + \Delta_{add}, \Delta_{growth})$ the claim follows.

Note that we are not making a formal security claim here. The models in [16, 11] are different from ours in the exact details and considerably work would have to be done to re-prove them secure in our model. However, the results in [16, 11] justify that $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ is a reasonable simple model of the finalized part of a blockchain for the sake of proving secure abstract protocol designs which use only the finalised part of a blockchain. Towards a secure real-life implementation of our sharding scheme it is an important step to ensure that $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ is securely implemented as a basis. This involves as much distributed systems engineering as it does cryptography.

Prevent delays for control messages. In order to achieve $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ with constant Δ it is important that at any time there is a bounded number of valid input messages that can be added to blocks. In our use-case, where $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ acts as a control chain for a bounded number of shard, this can be ensured. Moreover, these control messages should be given priority on the peer-to-peer layer, such that they propagate in some bounded time Δ_{net}. Finally, parties should priorties control messages when adding messages to blocks (in case the blocks are also used for different messages).

4.2 Shard Consensus

In this section we present a simple consensus algorithm that can be used to establish a total order of the transactions in a shard. Then, we show that it implements the shard functionality $\mathcal{F}_{\text{SHARD}}^{\mathcal{C},\mathcal{S},\mathcal{L},\Delta}$ described in Section 3.2.

Let $\mathcal{C}$ with $|\mathcal{C}| = n$ be the ordered committee of parties running the shard, and let $P^* \in \mathcal{C}$ be a special[5] party that we call the "sequencer". Note that the sequencer P^* additionally plays the role of a regular party in the protocol.

The protocol idea is simple. The sequencer periodically proposes a new block containing new messages, which is then signed by the other parties. A block is considered final if enough parties have signed it. For liveness detection a party can send old messages to the sequencer. The party will then refuse to sign the next block if it does not contain those messages.

Saftey and liveness guarantees. The protocol is safe, and thus persistence and proof soundness hold, whenever at most t_{S} parties are corrupted, and it is live whenever at most t_{L} parties are corrupted and P^* is honest. Hence, our protocol implements the ledger $\mathcal{F}_{\text{SHARD}}^{\mathcal{C},\mathcal{S},\mathcal{L},\Delta}$ with

$$\mathcal{S} = \{A \subseteq \mathcal{C} \mid |A| \leq t_{\text{S}}\},$$
$$\mathcal{L} = \{A \subseteq \mathcal{C} \mid |A| \leq t_{\text{L}} \ \wedge \ P^* \notin A\}.$$

The liveness threshold t_{L} can be chosen arbitrarily from $\{0, \ldots, \lfloor \frac{n-1}{2} \rfloor\}$. The safety threshold is then $t_{\text{S}} := n - 2t_{\text{L}} - 1$. For example, with $n = 100$, one can set $t_{\text{L}} = 33$ to obtain $t_{\text{S}} = 33$, which are the classical bounds for asynchronous Byzantine agreement. One can also set $t_{\text{L}} = 0$ to obtain $t_{\text{S}} = n - 1$, i.e., if full honesty is required for liveness, all but one party can be corrupted without breaking safety.

[5] Without loss of generality we choose P^* to be the first party in $\mathcal{C}$.

Blocks. In the protocol messages are added blockwise. A block $B = (c, \vec{m}_B, h, \ell, \vec{\sigma})$ consist of the block counter c, the tuple of new messages $\vec{m}_B$, hash h and length ℓ of the ledger after adding the messages in B, and a set of signatures on the previous block. To sign a block $B = (c, \vec{m}_B, h, \ell, \vec{\sigma})$ a party P_i actually signs the block header $(c, H(\vec{m}_B), h, \ell, H(\vec{\sigma}))$.

Let $\vec{m}$ be the current ledger state where B' denotes the latest block (if it exists). A block $B = (c, \vec{m}_B, h, \ell, \vec{\sigma})$ is a *valid* extension if

- $\ell = |\vec{m}||\vec{m}_B|$ and $h = \vec{H}(\vec{m}||\vec{m}_B)$,
- $c = 1$ or $\vec{\sigma}$ is a set of valid signatures on the previous block from at least $n - t_{\mathrm{L}}$ different parties in $\mathcal{C}$.

The protocol works as follows:

Protocol $\Pi_{\mathsf{seq}}^{n,t_{\mathrm{L}}}$

Initialization

At the beginning, all parties P_i initialize TO_i to be the empty list.
Each party P_i sets $\mathcal{M}_{i,1} = \varnothing$, $\mathsf{ctr}_{\mathsf{ledger},i} = 1$, and $\mathsf{ctr}_{\mathsf{sign},i} = 1$.
The sequencer additionally sets $\mathsf{ctr}_{\mathsf{seq}} = 1$.

Protocol for Sequencer P^*

- For $\mathsf{ctr}_{\mathsf{seq}} = 1$: For $\mathsf{ctr}_{\mathsf{seq}} = 1$:
  1: Collect all messages as sequence $\vec{m}_B$ ordered by arrival time.
  2: Compute $h = \vec{H}(\vec{m}_B)$ and set $\ell = |\vec{m}_B|$.
  3: Sign $B = (\mathsf{ctr}_{\mathsf{seq}}, \vec{m}_B, h, \ell, \perp)$ (via the signature functionality)
  4: Multicast signed B to parties in $\mathcal{C}$ and set $\mathsf{ctr}_{\mathsf{seq}} = \mathsf{ctr}_{\mathsf{seq}} + 1$.
- Once P^* has received $n - t_{\mathrm{L}}$ valid signatures $\vec{\sigma}$ on the previous block $B' = (\mathsf{ctr}_{\mathsf{seq}} - 1, \vec{m}'_{B'}, h', \ell', \vec{\sigma}')$:
  1: Collect all messages that have been sent with the $n - t_{\mathrm{L}}$ signatures or have been otherwise received by P^*, but which have not been included in the ledger (i.e., are not in $\vec{m}'$ where $\vec{H}(\vec{m}') = h'$). Denote by $\vec{m}_B$ the sequence of those messages ordered by arrival time.
  2: Compute $h = \vec{H}(h'||\vec{m}_B)$ and set $\ell = \ell' + |\vec{m}_B|$.
  3: Sign $B = (\mathsf{ctr}_{\mathsf{seq}}, \vec{m}_B, h, \ell, \vec{\sigma})$ (via the signature functionality).
  4: Multicast signed B to parties in $\mathcal{C}$ and set $\mathsf{ctr}_{\mathsf{seq}} = \mathsf{ctr}_{\mathsf{seq}} + 1$.

Protocol for $P_i \in \mathcal{C}$

- Once P_i has received signed $B = (c, \vec{m}_B, h, \ell, \vec{\sigma})$ from P^*:
  1: If the signature is invalid or B is invalid or $\mathsf{ctr}_{\mathsf{sign},i} \neq c$ abort.
  2: If a signature by P_i is in $\vec{\sigma}$, but $\mathcal{M}_{i,\mathsf{ctr}_{\mathsf{sign},i}}$ is not in $\vec{m}_B$ abort.
  3: Set $\mathcal{M}_{i,\mathsf{ctr}_{\mathsf{sign},i}+1}$ to the set of messages P_i has received but have not been included in blocks (i.e., message not in $\vec{m}$ with $\vec{H}(\vec{m}) = h$).
  4: Create signature σ_i on B (via the signature functionality).
  5: Multicast signed $(\sigma_i, \mathcal{M}_{i,\mathsf{ctr}_{\mathsf{sign},i}+1})$ to parties in $\mathcal{C}$ and set $\mathsf{ctr}_{\mathsf{sign},i} = \mathsf{ctr}_{\mathsf{sign},i} + 1$.
- Once P_i received $n - t_{\mathrm{L}}$ valid signatures $\vec{\sigma}$ on block $B = (c, \vec{m}_B, h, \ell, \vec{\sigma})$:
  1: If $\mathsf{ctr}_{\mathsf{ledger},i} < c$ store B in a buffer, repeat process once $\mathsf{ctr}_{\mathsf{ledger},i} = c$.

2: If $\mathsf{ctr}_{\mathsf{ledger},i} > c$ abort.
3: Add all messages $\vec{m}_B$ to TO_i, i.e., set $\mathsf{TO}_i = \mathsf{TO}_i || \vec{m}_B$.
4: Locally store B with the signatures and set $\mathsf{ctr}_{\mathsf{ledger},i} = \mathsf{ctr}_{\mathsf{ledger},i} + 1$.

Interface for $P_i \in \mathcal{C}$

- On input (SEND, m), party $P_i \in \mathcal{C}$ multicasts the message m to all parties in $\mathcal{C}$.
- On input GET, party P_i returns TO_i.
- On input CLOSE, party P_i stops participating in the protocol.
- On input GETFINPROOF, party P_i does the following:
  1: If TO_i is empty, set $\pi = \perp$ and return (TO_i, π).
  2: Otherwise let $B = (c, \vec{m}_B, h, \ell, \vec{\sigma})$ be the block of the last messages added to TO_i.
  3: Set $\pi = ((c, H(\vec{m}_B), h, \ell, H(\vec{\sigma})), \hat{\sigma})$ where $\hat{\sigma}$ is the set of collected signatures on B.
  4: Return (TO_i, π).

Public interface

- On input (VERIFYFINPROOF, $\vec{m}, \pi$), do the following:
  1: If $\pi = \perp$, return 1 if and only if $\vec{m}$ is empty.
  2: Otherwise, let $\pi = ((c, H(\vec{m}_B), h, \ell, H(\vec{\sigma})), \hat{\sigma})$.
  3: Check that $\hat{\sigma}$ contains at least $n - t_{\mathrm{L}}$ valid signatures on $(c, H(\vec{m}_B), h, \ell, H(\vec{\sigma}))$.
  4: Check that $\vec{H}(\vec{m}) = h$ and $\mathsf{length}(\vec{m}) = \ell$.
  5: If all checks pass, return 1, otherwise return 0.
- On input (VERIFYFINLENGTH, $\hat{\ell}, \pi$), do the following:
  1: If $\pi = \perp$, return 1 if and only if $\hat{\ell} = 0$.
  2: Otherwise, let $\pi = ((c, H(\vec{m}_B), h, \ell, H(\vec{\sigma})), \hat{\sigma})$.
  3: Check that $\hat{\sigma}$ contains at least $n - t_{\mathrm{L}}$ valid signatures on $(c, H(\vec{m}_B), h, \ell, H(\vec{\sigma}))$.
  4: Check that $\hat{\ell} = \ell$.
  5: If all checks pass, return 1, otherwise return 0.

Security analysis. We now argue that the protocol $\Pi_{\mathsf{seq}}^{n,t_{\mathrm{L}}}$ described above UC-realizes the functionality $\mathcal{F}_{\mathrm{SHARD}}$ described in Section 3.2.

Theorem 1. *The protocol $\Pi_{\mathsf{seq}}^{n,t_{\mathrm{L}}}$ UC-realizes the functionality $\mathcal{F}_{\mathrm{SHARD}}^{\mathcal{C},\mathcal{S},\mathcal{L},\Delta}$ for $\Delta \geq 5 \cdot \Delta_{NET}$ in the hybrid model with access to hybrids for a reliable network with maximal delay Δ_{NET}, a signature functionality, and a clock.*

Proof. The simulator runs a simulation of the protocol $\Pi_{\mathsf{seq}}^{n,t_{\mathrm{L}}}$ for the honest parties. That is, when $\mathcal{F}_{\mathrm{SHARD}}$ outputs (SEND, P_i, m) for an honest P_i to the simulator, the simulator simulates P_i distributing m to parties in $\mathcal{C}$. If P^* is honest, the simulator further simulates the block generation and sending of blocks from P^* as described in $\Pi_{\mathsf{seq}}^{n,t_{\mathrm{L}}}$. Furthermore, signing of blocks by honest parties is simulated as described in the protocol. Whenever an honest party P_i in simulated protocol $\Pi_{\mathsf{seq}}^{n,t_{\mathrm{L}}}$ adds a message m to TO_i, the simulator invokes the ideal functionality with (ADD, m, P_i). As soon as at least $n - t_{\mathrm{L}} - |A|$ honest parties signed a block in the simulation, where A is the set of corrupted parties, the simulator invokes (ADDFINAL, m) for all messages m in that block. When $\mathcal{F}_{\mathrm{SHARD}}$ outputs (GETFINPROOF, P_i) for honest P_i to the

simulator, the simulator returns π as computed by P_i in protocol $\Pi_{\mathsf{seq}}^{n,t_\mathrm{L}}$. When $\mathcal{F}_{\mathrm{SHARD}}$ outputs $(\textsc{VerifyFinProof}, \vec{m}, \pi)$ to the simulator, the simulator verifies the proof as described in $\Pi_{\mathsf{seq}}^{n,t_\mathrm{L}}$. When $\mathcal{F}_{\mathrm{SHARD}}$ outputs $(\textsc{VerifyFinLength}, \ell, \pi)$ to the simulator, the simulator verifies the proof as described in $\Pi_{\mathsf{seq}}^{n,t_\mathrm{L}}$.

As long as the simulator does not violate the constraints of persistence, liveness, censorship resilience, or proof soundness described in $\mathcal{F}_{\mathrm{SHARD}}$, the ideal world with the simulator and the real world with the protocol execution are identical. Hence, it suffices to prove that the protocol always respects these constraints.

Persistence: Let P_i be an honest party and assume TO_i is at some point not a prefix of FTO. Recall that messages are added to TO_i only after P_i has collected at least $n - t_\mathrm{L}$ signatures; at least $n - t_\mathrm{L} - |A|$ of these signatures must be from honest parties. Since after $n - t_\mathrm{L} - |A|$ honest signatures a message is also added to FTO in the simulation, then TO_i cannot contain messages that are not already in FTO. Therefore, TO_i not being a prefix of FTO implies that there is a position l_0 in which TO_i and FTO contain different messages. If these messages come from blocks with the same counter, two different blocks with that counter have been signed. Otherwise, there is a smaller counter for which blocks containing a different number of messages have been signed. Let c_0 be the smallest counter for which two different blocks have been added to TO_i and FTO. One of these blocks was signed by at least $n - t_\mathrm{L}$ parties (since it is in TO_i), and the other one by at least $n - t_\mathrm{L} - |A|$ *honest* parties (since it is in FTO). If the ledger is safe, we have $|A| \leq t_\mathrm{S} = n - 2t_\mathrm{L} - 1$. Hence, there are at least

$$n - t_\mathrm{L} - |A| \geq n - t_\mathrm{L} - (n - 2t_\mathrm{L} - 1) = t_\mathrm{L} + 1$$

honest parties who signed the block in FTO. Since $(n - t_\mathrm{L}) + (t_\mathrm{L} + 1) > n$, at least one of these honest parties also signed the block in TO_i. This is a contradiction because honest parties never sign two blocks with the same counter.

Censorship Resilience: After an honest party input $(\textsc{Send}, m)$ at time t, the message m arrives at all honest parties in $\mathcal{C}$ within Δ_{NET}. If an honest party P_i adds a block to TO_i after $t + 2\Delta_{\mathrm{NET}}$, the honest parties must have signed the block after time $t + \Delta_{\mathrm{NET}}$. If m was not added until now, all honest parties will request that the sequencer adds m to the next block. So TO_i must contain m after honest P_i adds the second block. The property follows for $\Delta \geq 2\Delta_{\mathrm{NET}}$.

Liveness: Assume P^* is honest and at least $n - t_\mathrm{L}$ parties are honest. After an honest party input $(\textsc{Send}, m)$, the protocol sends m to P^*, who receives it at most Δ_{NET} time later. As it takes at most $2\Delta_{\mathrm{NET}}$ to send out a block and receive signatures, P^* produces a new block containing m at most $2\Delta_{\mathrm{NET}}$ later and sends it to all honest parties. These honest parties receive that message at most Δ_{NET} time later, sign it, and send their signatures to all other honest parties. This again takes at most Δ_{NET} time. Since at least $n - t_\mathrm{L}$ parties are honest, all honest parties receive at least that many signatures on that block and thus add it to their TO_i. This in total takes at most $5\Delta_{\mathrm{NET}}$ time.

Proof soundness: First, consider the case $(\textsc{VerifyFinProof}, \vec{m}, \pi)$. Toward contradiction, assume that $(\textsc{VerifyFinProof}, \vec{m}, \pi)$ returns 1, but $\vec{m}$ is not a prefix of FTO. In that case, π contains a block header B and at least $n - t_\mathrm{L}$ valid signatures on B. This means that at least $n - t_\mathrm{L} - |A|$ honest parties have signed B. Furthermore, B contains a hash h such that $h = \vec{H}(\vec{m})$. By construction of the protocol, all these honest parties must have previously signed blocks with smaller counters containing messages with consistent hashes.

Since messages are added to FTO once $n - t_{\mathrm{L}} - |A|$ honest signatures exist, $\vec{m}$ can only not be a prefix of FTO if different messages yield the same hashes. A standard reduction to the collision-resistant property of the hash function H finally shows that any PPT adversary that violates the soundness of the finality proof can efficiently find collisions on H.

Next, consider $(\textsc{VerifyFinLength}, \ell, \pi)$. Towards a contradiction, we assume that $(\textsc{VerifyFinLength}, \ell, \pi)$ returns 1, but $|\mathsf{FTO}| < \ell$. In that case, π contains a block header B and at least $n - t_{\mathrm{L}}$ valid signatures on B. This means that at least $n - t_{\mathrm{L}} - |A|$ honest parties have signed B. Furthermore, B contains a hash h defining a messages vector $\vec{m}$ and length ℓ. By construction of the protocol, all these honest parties have checked that $|\vec{m}| = \ell$. By the above argument $\vec{m}$ must also be a prefix of FTO, which contradicts $|\mathsf{FTO}| < \ell$.

□

5 Committee Selection

In this section, we describe a committee-selection functionality, which is a core part of our sharding solution. We then discuss how to realize that functionality given a randomness beacon in permissionless blockchains. We further provide an analysis of the committee sizes needed for that approach.

5.1 Ideal Committee-Selection Functionality

We first describe the functionality $\mathcal{F}_{\textsc{ComSel}}^{\mathcal{P},\mathcal{U}}$, which is parameterized by the set $\mathcal{P}$ of parties executing the committee selection, and a set $\mathcal{U}$ of parties from which the committee gets selected. The functionality allows parties to request uniformly distributed sequences over $\mathcal{U}$ of a given length. As discussed in Section 5.3, the issue of mapping parties who control different fractions of a restricted resource (e.g., relative stake or computational power) into such a set $\mathcal{U}$ for selecting committees in a permissionless blockchain scenario has been extensively addressed in both Proof-of-Stake [20, 7, 12, 10, 17, 9] and Proof-of-Work [26] settings.

Functionality $\mathcal{F}_{\textsc{ComSel}}^{\mathcal{P},\mathcal{U}}$

Interface for party $P_i \in \mathcal{P}$

Input: $(\textsc{SelectCom}, \mathsf{cid}, s)$ // Select committee with ID cid of size s
Output $(\textsc{SelectCom}, P_i, \mathsf{cid}, s)$ to the adversary.

Upon receiving $(\textsc{SelectCom}, \mathsf{cid}, s)$ (with the same cid and s) from all honest parties in $\mathcal{P}$, sample a sequence $\mathcal{C}$ uniformly among all sequences of length s from $\mathcal{U}$, and send $(\mathsf{cid}, \mathcal{C})$ to all parties in $\mathcal{P}$.

Public interface // Any (even "outside") party can use this interface.

Input: $(\textsc{VerifyCommittee}, \mathsf{cid}, \mathcal{C})$
If $(\mathsf{cid}, \mathcal{C})$ has been sent to all parties in $\mathcal{P}$, return 1, otherwise, return 0.

5.2 Randomness Beacon

Our committee selection procedure can be realized from a randomness beacon. We model a randomness beacon as an ideal functionality $\mathcal{F}_{\mathsf{CT}}$ as described in [8], where it is also shown how this functionality can be efficiently UC-realized both based on the DDH assumption (with UC Zero Knowledge as setup) or on the CDH assumption (with a Global Random Oracle as setup). Moreover the protocols realizing $\mathcal{F}_{\mathsf{CT}}$ proposed in [8] require a public bulletin board that guarantees that posted messages become immutable and accessible to all honest parties. We observe that such a bulletin board can be realized by $\mathcal{F}_{\text{BD-TL}}$.

We describe the functionality $\mathcal{F}_{\mathsf{CT}}^{\mathcal{P},\mathcal{D}}$ parameterized by a set of parties $\mathcal{P}$ running the coin toss, and a distribution $\mathcal{D}$. It is defined as follows.

Functionality $\mathcal{F}_{\mathsf{CT}}^{\mathcal{P},\mathcal{D}}$

Interface for party $P_i \in \mathcal{P}$

Input: (Toss, sid)
Upon receiving (Toss, sid) from all honest parties in $\mathcal{P}$, uniformly sample $x \xleftarrow{\$} \mathcal{D}$ and send (Tossed, sid, x) to all parties in $\mathcal{P}$.

Public interface // Any (even "outside") party can use this interface.

Input: (Verify, sid, x)
If (Tossed, sid, x) has been sent to all parties in $\mathcal{P}$, return 1, otherwise, return 0.

5.3 Realizing Committee Selection Using Randomness Beacon

A straightforward way to realize $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ works as follows. Given a committee size s, use randomness from $\mathcal{F}_{\mathsf{CT}}^{\mathcal{P},\mathcal{D}}$ with $\mathcal{D}$ set to the uniform distribution, to sample a uniformly random sequence $\mathcal{C}$ from $\mathcal{U}$ with $|\mathcal{C}| = s$. It is easy to see that this realizes $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$: The liveness directly follows from liveness of the randomness beacon since parties compute the committees locally from that randomness. Since the beacons also provide the same randomness to all parties, the parties agree on the selected committees.

The simple protocol above departs from a set $\mathcal{U}$ of parties such that each party is selected as a committee member with equal probability. However, in permissionless blockchain protocols, corruption thresholds are expressed in terms of the amount of a restricted resource controlled by a party (e.g., the amount of relative stake in Proof-of-Stake based blockchains or the amount of computational power in Proof-of-Work based blockchains). Hence, it is necessary to map the parties executing the underlying blockchain protocol into (virtual) parties in such a set $\mathcal{U}$ according to the resources they control. In the setting of Proof-of-Stake based blockchains, such mapping can be achieved by the techniques commonly known as "follow-the-satoshi" [20, 7], "weighing by stake" [12] and "cryptographic sortition" [10, 17, 9]. In the setting of Proof-of-Work based blockchains, it has been shown that committee selection can be realized even without a randomness beacon [26]. Since this issue has been addressed multiple times in previous works, we focus instead on the novel aspects of our sharded ledger protocol, referring interested readers to the aforementioned results on committee selection on blockchains.

5.4 Determining the Committee Size

Our sharding protocol needs to find the smallest committee size s_{min} that guarantees that the ratio of corrupted parties in the selected committee is below some given threshold with overwhelming probability. Consider a scenario with a total population of n parties $\mathcal{P}$ such that at most t parties are corrupt, and a committee $\mathcal{C}$ with size s sampled uniformly at random from $\mathcal{P}$. We denote by $\mathsf{FAIL}_{t',s}^{t,n}$ the event where the committee $\mathcal{C}$ contains more than t' corrupt parties. The probability of the event $\mathsf{FAIL}_{t',s}^{t,n}$ happening can be expressed as the cumulative hypergeometric probability mass function:

$$\Pr\left[\mathsf{FAIL}_{t',s}^{t,n}\right] = \sum_{i=t'+1}^{i=s} \frac{\binom{t}{i}\binom{n-t}{s-i}}{\binom{n}{s}}. \tag{1}$$

Given a maximal corruption ratio $\frac{t'}{s}$ of the sampled committee $\mathcal{C}$ of size s one can find the smallest size s_{min} for which $\Pr\left[\mathsf{FAIL}_{t',s}^{t,n}\right] \leq 2^{-\kappa}$, for some security parameter κ. In figure 1 we show a graph relating the minimum committee size for a varying corruption ratio within the sampled committee. The blue line represents exact committee sizes sampled uniformly from a population of 10K parties with 30% corruption, computed using the code in Appendix A that uses the failing probability described in equation 1. The red line represents committee sizes derived from an infinite population size with 30% corruption by using the analytic bound described in Section 5.5. Both are for a security level of 60-bits. For convenience, Table 1 shows a few of the concrete values used for plotting the graphs in Figure 1.

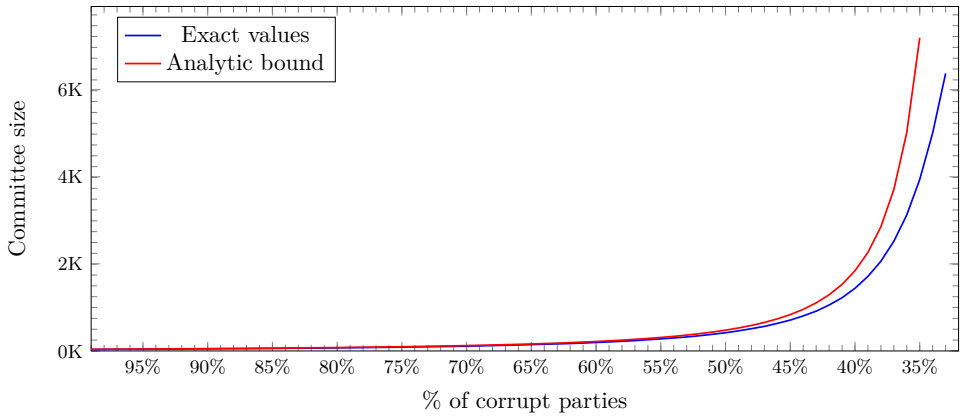

Figure 1: Relation between minimum committee sizes for varying corruption ratio within the sampled committee for a security parameter $\kappa = 60$. Exact values computed using the code in Appendix A for a population of 10K parties with 30% corruption. Analytic bounds (c.f. Section 5.5) derived from an infinite population size with 30% corruption.

5.5 Tight Analytic Bounds

We analyse the probability that when sampling a random committee of size n from a population with corruption fraction p we get a committee with corruption $\leq q$ for some $q > p$. We will calculate the probability of the bad event that there are $> q$ corruptions, and give a formula for picking n such that this bad event has small probability. We consider sampling without replacement. However, it is easy to see that we will get an upper bound by assuming sampling with replacement. We will do this. We can therefore ignore the size of the population. The analytic bound will be an upper bound. We propose using it for finding the approximately

	Maximal corruption ratio in committee														
	99%	94%	89%	84%	79%	74%	69%	64%	59%	54%	49%	44%	39%	34%	33%
Exact values	35	42	51	61	75	92	116	155	207	299	462	805	1713	5009	6376
Analytic bound	37	45	55	67	82	102	130	170	232	335	528	955	2264	11178	19761

Table 1: Table showing the comparison of minimum committee sizes for different corruption ratios within the sampled committee between exact committee sizes and bounds derived with the analytical bound of Section 5.5. Results for a total population size $n = 10000$ (for the exact values) with 30% corrupted parties and security parameter $k = 60$.

best committee size and then fine tune the selection with a more computationally heavy exact calculate around the approximate value.

We sample n elements with replacement. Let P_x be the probability that exactly x members are corrupt. Then,

$$P_x = \binom{n}{x} p^x (1-p)^{n-x} \ .$$

We are interested in upper bounding $\sum_{x=qn+1}^{n} P_x$. We have that

$$\frac{P_{x+1}}{P_x} = \frac{\binom{n}{x+1} p^{x+1}(1-p)^{n-x-1}}{\binom{n}{x} p^x (1-p)^{n-x}} = \frac{(n-x)p}{(x+1)(1-p)} \ .$$

When $x > qn$ then

$$\frac{(n-x)}{(x+1)} < \frac{1-q}{q+1/n} < \frac{1-q}{q} \ .$$

It follows that

$$P_{x+1} < P_x \frac{(1-q)p}{q(1-p)} \ .$$

If we do a geometric sum with start term $a = P_{qn}$ and ratio $\frac{(1-q)p}{q(1-p)}$, we get

$$\sum_{x=qn+1}^{n} P_x < \frac{P_{qn}}{1 - \frac{(1-q)p}{q(1-p)}} = P_{qn} \frac{q(1-p)}{q(1-p) - (1-q)p} = P_{qn} \frac{q(1-p)}{q-p} \ .$$

Note that $\frac{q(1-p)}{q-p}$ goes to ∞ as q goes to p, as expected. Assume now that we leave a constant gap α between the corruption we assume and the one we attempt to sample, i.e., $\alpha > 0$ and $q = p + \alpha$. Then

$$\frac{q(1-p)}{q-p} < \frac{q(1-p)}{p+\alpha-p} = q(1-p)\alpha^{-1} \ ,$$

which gives us

$$\sum_{x=qn+1}^{n} P_x < P_{qn} \frac{q(1-p)}{\alpha} \ .$$

We will now focus on P_{qn}. We have that

$$P_{qn} = \binom{n}{qn} p^{qn} (1-p)^{n(1-q)} \ .$$

Furthermore,

$$\binom{n}{qn} = \frac{n!}{(qn)!(n-qn)!} \ .$$

By Stirling's approximation we have

$$\sqrt{2\pi n} n^n e^{-n} < n! < e\sqrt{n} n^n e^{-n} \ .$$

This yields

$$\binom{n}{qn} < \frac{e\sqrt{n} n^n}{\sqrt{2\pi qn}(qn)^{qn}\sqrt{2\pi(n-qn)}(n-qn)^{n-qn}} = \frac{e\sqrt{n}}{\sqrt{2\pi qn}\sqrt{2\pi(n-qn)}} \frac{n^n}{(qn)^{qn}(n-qn)^{n-qn}} \ .$$

We have

$$\frac{e\sqrt{n}}{\sqrt{2\pi qn}\sqrt{2\pi(n-qn)}} = \frac{e}{2\pi\sqrt{n}\sqrt{q(1-q)}} \ ,$$

and

$$\frac{n^n}{(qn)^{qn}(n-qn)^{n-qn}} = \frac{n^n}{q^{qn}n^{qn}n^{n-qn}(1-q)^{n-qn}} = \frac{1}{q^{qn}(1-q)^{n-qn}} = \left(q^q(1-q)^{1-q}\right)^{-n} \ .$$

Putting these together, we conclude that

$$\binom{n}{qn} < \frac{e}{2\pi\sqrt{n}\sqrt{q(1-q)}} \left(q^q(1-q)^{1-q}\right)^{-n} \ .$$

Putting all the above together, we obtain

$$\sum_{x=qn+1}^{n} P_x < \frac{q(1-p)}{\alpha} \frac{e}{2\pi\sqrt{n}\sqrt{q(1-q)}} \left(q^q(1-q)^{1-q}\right)^{-n} p^{qn}(1-p)^{n(1-q)} \ .$$

Collecting terms a bit we can simplify to

$$\sum_{x=qn+1}^{n} P_x < \sqrt{n}^{-1}\beta \left(\left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q}\right)^n \ .$$

For

$$\beta = \frac{e}{2\pi\alpha} \frac{\sqrt{q}(1-p)}{\sqrt{1-q}} \ .$$

For reasonable values of p and q it is typically the case that $n \geq \beta^2$. For all such p and q it holds that

$$\sum_{x=qn+1}^{n} P_x < \left(\left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q}\right)^n \ .$$

Therefore, we get

$$\sum_{x=qn+1}^{n} P_x < 2^{-\kappa}$$

when additionally

$$n \geq \kappa / \log_2 \left(\left(\frac{q}{p}\right)^q \left(\frac{1-q}{1-p}\right)^{1-q}\right) \ .$$

This shows that for large enough κ, the committee size n has a straight forward linear dependence on κ with a constant which follows from p and q. If if κ is so small that one gets $n < \beta^2$, then one can either just β^2 as an upper bound, or solve the following inequality for n:

$$n \geq \frac{\kappa + \log_2(\sqrt{n}^{-1}\beta)}{\log_2 \left(\left(\frac{q}{p}\right)^q \left(\frac{1-q}{1-p}\right)^{1-q}\right)} \ .$$

5.6 Alternative Ways to Realize Committee Selection

In the setting of Proof-of-Work based blockchains, committee selection based on the Proof-of-Work mechanism itself (without randomness beacons) has been constructed in [26]. In previous works on Proof-of-Stake based blockchain consensus protocols [20, 7, 12, 10, 17, 9], a number of methods have been proposed for selecting committees in a publicly verifiable way using randomness beacons. These methods can be classified in two main categories according to the underlying randomness beacon: (1) uniformly random committee selection using randomness beacons based on coin tossing with guaranteed output delivery [20, 7]; and (2) biased committee selection using randomness beacons based on verifiable random functions [12, 10, 17, 9]. The protocol we have described in Section 5.3 falls into the first category. While the methods in category 2 allow an adversary to bias committee selection in a way that is not possible in category 1, they have higher concrete efficiency. To keep the presentation simple, our formalization and the derived bounds assume uniform committee selection. However, our results can be extended to also work with biased committees.

6 Constructing a Sharded Ledger

To realize $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$, we use a timed ledger $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ as a control chain. In order to keep the shards live and monitor their liveness, the parties in $\mathcal{P}$ will follow instructions based on messages posted on $\mathcal{F}_{\text{BD-TL}}^{\Delta}$. We use $\mathcal{F}_{\text{ComSel}}$ for selecting the new committees when we start or re-start a shard. Moreover, we use a repository functionality $\mathcal{F}_{\text{Repo}}$ to store previous shards' states and allow for parties to obtain them.

Recall that for a fixed committee $\mathcal{C}$, our shard functionality is denoted by $\mathcal{F}_{\text{Shard}}^{\mathcal{C},\mathcal{S},\mathcal{L},\Delta}$. This ideal functionality describes the shard for a given committee $\mathcal{C}$. However, we need that the shard protocol works generically for any subset $\mathcal{C}$ of the parties of a given size, and with enough honest parties. To be concise, we use the notation $\mathcal{F}_{\text{Shard}}^{n,\mathcal{S},\mathcal{L},\Delta}$ to describe a sharding functionality which has not yet been instantiated with a committee, and which expects a committee of size n.[6] In order to instantiate shards, we use a "gearbox" of functionalities $\mathcal{F}_{\text{Shard}}^{n_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_1}, \ldots, \mathcal{F}_{\text{Shard}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$ to handle shard consensus. In principle each shard could use its own gearbox, but for the sake of presentation nothing is lost in using the same gearbox for all shards. There is a statistical security parameter κ, and a liveness guarantee $\gamma > 0$, e.g., $\gamma = \frac{1}{2}$. The gearbox should have the following properties.

Always safe: For a uniformly random $\mathcal{C} \subset \mathcal{P}$ of size n_i it holds for all i that $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_i,\mathcal{S}_i,\mathcal{L}_i,\Delta_i}$ is safe except with probability $2^{-\kappa}$.

Eventually live: For a uniformly random $\mathcal{C} \subset \mathcal{P}$ of size n_ℓ it holds that $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$ is live with probability at least γ.

We get safety and liveness at the same time by first running $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_1}$ for a random committee. If it loses liveness we switch gears to $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_2,\mathcal{S}_2,\mathcal{L}_2,\Delta_2}$ and so on. We start with $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_1,\mathcal{S}_1,\mathcal{L}_1}$ with very poor liveness but very small committees and high speed. On the other hand $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$ has strong liveness guarantees. The other $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_i,\mathcal{S}_i,\mathcal{L}_i,\Delta_i}$ act as intermediary points on the liveness versus efficiency scale, so some of them might be the same consensus mechanism repeated several times. It makes sense to try to sample committees several times (or to increase the size of

[6] In UC we can do this formally as follows. There is an ideal functionality $\mathcal{F}_{\text{Shard}}^{n,\mathcal{S},\mathcal{L},\Delta}$ with n associated to it. We consider $\mathcal{C}$ as part of the session identifier of this ideal functionality. That means that in UC when the parties $P_i \in \mathcal{C}$ call $\mathcal{F}_{\text{Shard}}^{n,\mathcal{S},\mathcal{L},\Delta}$, they identify the set $\mathcal{C}$ as part of the sid. If the set does not have size n the ideal functionality does nothing. If the set has size n, then the instance of $\mathcal{F}_{\text{Shard}}^{n,\mathcal{S},\mathcal{L},\Delta}$ thus created is what we call $\mathcal{F}_{\text{Shard}}^{\mathcal{C},\mathcal{S},\mathcal{L},\Delta}$.

	Actual total corruption ratio					
	5%	10%	15%	20%	25%	30%
Sufficient safety threshold	89%	79%	69%	59%	49%	39%
Committee size	51	75	116	207	462	1713

Table 2: Table showing for which safety thresholds shards will have a good probability to be live with different overall corruption ratios, using $S + 2L < 100\%$ and setting L to the overall corruption ratio. The second line shows the required shard sizes to guarantee safety with values taken from Table 1.

committees) to try to hit one with high honesty and hence liveness. However, we only proceed to the next level if the previous one loses liveness.

Optimistic committee sizes. As described above, the gearbox increases committee sizes until the shard is live. We will now discuss how large committees need to be to get liveness. For concreteness, we again consider a total population of $10K$ nodes with at most 30% corruption, as in Section 5. Even though we consider a worst-case corruption of up to 30% in the total population, in an optimistic scenario the system can have a lot less corruption. Note that the corruption ratio in the sampled shard is close to the overall corruption ratio with high probability. Thus, when the liveness threshold of the current gear is equal to the actual total corruption threshold, there is a good probability that the shard is live. Since the gearbox ensures that shards are always safe, one can think of shifting up as increasing the liveness threshold L and adjusting the safety threshold S such that $S + 2L < 100\%$. As soon as the liveness threshold matches the unknown actual corruption ratio, the shard is live with good probability and the gearbox stops switching up. This means the protocol automatically finds the optimal shard size without knowing the actual corruption ratio. Note that even in the worst case with 30% overall corruption, we only need $L = 30\%$ and $S = 39\%$. Hence, we can sample a committee with a guaranteed corruption of at most 39%, instead of the 33% required by solutions not leveraging the safety-liveness dichotomy.

Table 2 shows different committee sizes needed to get liveness for varying corruption ratios. For example, in the worst case with 30% overall corruption, we need a safety threshold of 39%, corresponding to 1713 committee members. With 20% corruption, we only need committees of size 207.

6.1 The Sharded Ledger Protocol $\Pi_{\text{BD-STL}}$

In our protocol, parties in $\mathcal{P}$ continuously perform a number of maintenance actions in order to detect the potential loss of liveness in shards and ensure that a new committee is chosen to take over the operation of a shard that has lost liveness. These actions can be divided as follows: (1) Shard Management, which are actions performed by all parties in $\mathcal{P}$; (2) Shard Operation, which are actions performed by the parties in the committee responsible for operating a given shard. Moreover, when receiving inputs, the parties execute instructions that realize the interfaces in $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$. The protocol works as follows:

Protocol $\Pi_{\text{BD-STL}}$

Parties in $\mathcal{P}$ interact with each other and with functionalities $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, $\mathcal{F}_{\text{Repo}}$, $\mathcal{F}_{\text{ComSel}}$ and the gearbox of functionalities $\mathcal{F}_{\text{Shard}}^{n_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_1}, \ldots, \mathcal{F}_{\text{Shard}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$. All parties in $\mathcal{P}$ post management instructions to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ in order to ensure shards are live, selecting a new shard committee with the help of $\mathcal{F}_{\text{ComSel}}$ if a loss of liveness is detected. Each shard `sid` is associated to a functionality $\mathcal{F}_{\text{Shard}}^{n_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ in the gearbox executed by a committee $\mathcal{C}_h$. Shards have parameters h, indicating which committee $\mathcal{C}_h$ and functionality $\mathcal{F}_{\text{Shard}}^{n_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ is responsible for that shard, start time t, indicating when the shard execution with $\mathcal{F}_{\text{Shard}}^{n_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ and $\mathcal{C}_h$ started in terms of $\mathcal{F}_{\text{BD-TL}}^{\Delta}$'s ledger time, and finalization time out t_{Timeout}, representing the maximum delay before a shard is considered to have lost liveness. When describing a shard with a given parameter h, we denote the associated functionality $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ as a short hand for executing $\mathcal{F}_{\text{Shard}}^{n_h,\mathcal{S}_h,\mathcal{L}_h}$ with committee $\mathcal{C}_h$.

Shard Management.

Init: Initially start all shards with parameters $h = 1, t = 1, t_{\text{Timeout}} = \Delta_{init}$ for all `sid`, where $\Delta_{init} \in \mathbb{Z}, \Delta_{init} > 0$. Initially set $\mathsf{SHDS}_i := \varnothing$ for all $P_i \in \mathcal{P}$.

Start: In order to start a shard identified by `sid` with parameters h, t, t_{Timeout}, all parties $P_i \in \mathcal{P}$ proceed as follows;

1: Send $(\text{SelectCom}, \mathsf{cid}, n_h)$ to $\mathcal{F}_{\text{ComSel}}$ with a new cid, getting $(\mathsf{cid}, \mathcal{C}_h)$.
2: Send $(\text{Send}, (\text{Start}, \texttt{sid}, \mathsf{cid}, \mathcal{C}_h, t, t_{\text{Timeout}}))$ to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, i.e., a command to start shard `sid` with $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ at ledger time t w.r.t. $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ with finalization timeout t_{Timeout}.
3: Set a finalization timeout counter to $c_{Timeout}^{\texttt{sid}} = t + t_{\text{Timeout}}$.

Stop: Parties $P_i \in \mathcal{P}$ send$(\text{Send}, (\text{Stop}, \texttt{sid}))$ to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, instructing parties to stop executing shard `sid` associated to $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$.

FinalizeCheck: Parties $P_i \in \mathcal{P}$ keep counters $L_{last}^{\texttt{sid}}$ initially set to 0 for each shard identified by `sid`. Parties $P_i \in \mathcal{P}$ continuously send (Get) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i. For every shard identified by `sid`, all $P_i \in \mathcal{P}$ performs the following steps to check that a shard has liveness:

1: For every message $((\text{Finalize}, \texttt{sid}, H, \pi, L), t) \in \mathsf{TO}_i$ check that that $t < c_{Timeout}^{\texttt{sid}}$ and $L > L_{last}^{\texttt{sid}}$, send $(\text{VerifyFinLength}, L, \pi)$ to $\mathcal{F}_{\text{Shard}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$, obtaining b, and check $b = 1$.
2: Let $((\text{Finalize}, \texttt{sid}, H, \pi, L_{max}), t) \in \mathsf{TO}_i$ be the message with the largest L for which the checks of Step 1 succeeded. Set $c_{Timeout}^{\texttt{sid}} = t + t_{\text{Timeout}}$ and set $L_{last} = L_{max}$.
3: If the checks did not succeed for any message $((\text{Finalize}, \texttt{sid}, H, \pi, L), t) \in \mathsf{TO}_i$ (i.e., the shard has timed out), stop the shard and start it again with incremented parameters $h++, t', t_{\text{Timeout}}++$, where t' is a future ledger time w.r.t. $\mathcal{F}_{\text{BD-TL}}^{\Delta}$. If the top of the gearbox $\mathcal{F}_{\text{Shard}}^{n_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_h}, \ldots, \mathcal{F}_{\text{Shard}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$ for the shard has been reached, it is restarted with the same parameters $h = \ell$ but with incremented $t_{\text{Timeout}}++$ and a freshly selected committee for $\mathcal{F}_{\text{Shard}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$.

Shard Operation. For every shard identified by `sid`, parties $P_i \in \mathcal{P}$ continuously send (Get) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i. Parties $P_i \in \mathcal{P}$ keep executing Shard Operation instructions as follows according to the messages in TO_i and the ledger time:

Start Shard: When parties $P_i \in \mathcal{C}_h$ see $((\text{START}, \mathtt{sid}, \mathsf{cid}, \mathcal{C}_h, t, t_{\text{TIMEOUT}}), t') \in \mathsf{TO}_i$, P_i checks that the message is valid by verifying the $\mathcal{F}_{\text{COMSEL}}$ returns $\mathcal{C}_h$ when queried with $(\text{SELECTCOM}, \mathsf{cid}, n_h)$. If it is valid, they set $c^{\mathtt{sid}}_{Timeout} = t + t_{\text{TIMEOUT}}$ and start executing $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ at ledger time t.

Finalize: At ledger time $c^{\mathtt{sid}}_{Timeout} - \Delta'$ (where we choose a higher Δ' every time the shard is restarted in order to ensure that finalization messages are received before the timeout), parties $P_i \in \mathcal{C}_h$ for shard $\mathtt{sid}$ proceed as follows:

1: Send GET to $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ corresponding to shard $\mathtt{sid}$, obtaining $\mathsf{STO}_i^{\mathtt{sid}}$.
2: Send GETFINPROOF to $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$, obtaining the corresponding finalization proof π.
3: Set $D = \mathsf{STO}_i|\pi$ and send (STORE, D) to $\mathcal{F}_{\text{REPO}}$. Notice that if a prefix of STO_i has already been stored in $\mathcal{F}_{\text{REPO}}$, only the new messages in STO_i w.r.t. this prefix need to be sent to $\mathcal{F}_{\text{REPO}}$.
4: Set $L = |\mathsf{STO}_i|$ and send $(\text{SEND}, (\text{FINALIZE}, \mathtt{sid}, H(\mathsf{STO}_i), \pi, L))$ to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$.

Stop Shard: When parties $P_i \in \mathcal{C}_h$ see a message $((\text{STOP}, \mathtt{sid}), t) \in \mathsf{TO}_i$, they send CLOSE to $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ and stop executing further shard operation instructions.

Interfaces from $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ for Parties $P_i \in \mathcal{P}$. Parties $P_i \in \mathcal{P}$ execute the following instructions upon receiving inputs:

On input $(\text{Send}, \mathtt{sid}, m)$: P_i proceeds as follows:

1: Send (GET) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i.
2: Find the latest valid message $((\text{START}, \mathtt{sid}, \mathsf{cid}, \mathcal{C}_h, t, t_{\text{TIMEOUT}}), t) \in \mathsf{TO}_i$ and determines the instance $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ responsible for shard $\mathtt{sid}$.
3: If $P_i \in \mathcal{C}_h$, when receiving m (as input or from another party), P_i sends (SEND, m) to $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ and ignore the next steps. Otherwise, proceed to the next step.
4: Send m to all parties in $\mathcal{C}_h$ and continuously sends (GET) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i and checking that there is a message $((\text{FINALIZE}, \mathtt{sid}, H, \pi, L), t) \in \mathsf{TO}_i$ such that $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ associated to shard $\mathtt{sid}$ returns 1 when queried with $(\text{VERIFYFINPROOF}, \vec{m}, \pi)$, where $\vec{m}$ is obtained by sending (GET, H) to $\mathcal{F}_{\text{REPO}}$. If a message $((\text{STOP}, \mathtt{sid}), t)$ appears before these checks succeed, P_i goes to Step 1 and waits for a new message $((\text{START}, \mathtt{sid}, \mathsf{cid}, \mathcal{C}_{h+1}, t, t_{\text{TIMEOUT}}), t) \in \mathsf{TO}_i$.

On input (Get): P_i sends (GET) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i. For each $\mathtt{sid} \in \mathsf{SHDS}_i$, P_i sets $\mathsf{STO}_i^{\mathtt{sid}} = \varnothing$ and determines the final $\mathsf{STO}_i^{\mathtt{sid}}$ by executing the following instructions starting from the largest value of L_j for each $((\text{FINALIZE}, \mathtt{sid}, H_j, \pi_j, L_j), t_j) \in \mathsf{TO}_i$:

1: Send $(\text{VERIFYFINLENGTH}, L_j, \pi_J)$ to $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ (where this instance is determined by the last valid message $((\text{START}, \mathtt{sid}, \mathsf{cid}, \mathcal{C}_h, t, t_{\text{TIMEOUT}}), t') \in \mathsf{TO}_i$), obtaining b. If $b = 0$, ignore the next steps and proceed to the next message $((\text{FINALIZE}, \mathtt{sid}, H_{j+1}, \pi_{j+1}, L_{j+1}), t_{j+1}) \in \mathsf{TO}_i$ with $L_{j+1} < L_j$.
2: Send (GET, H_j) to $\mathcal{F}_{\text{REPO}}$, receiving D_j, and send $(\text{VERIFYFINPROOF}, D_j, \pi)$ to $\mathcal{F}_{\text{SHARD}}^{\mathcal{C}_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$, obtaining b. If $b = 0$, ignore the next step and proceed to the next message $((\text{FINALIZE}, \mathtt{sid}, H_{j+1}, \pi_{j+1}, L_{j+1}), t_{j+1}) \in \mathsf{TO}_i$ with $L_{j+1} < L_j$.
3: Let D' be the new messages in D_j that are not contained in D_{j-1}. Append (D', t_j) to $\mathsf{STO}_i^{\mathtt{sid}}$.

Notice that if a previous version of $\mathsf{STO}_i^{\mathtt{sid}}$ has already been computed, P_i only needs to perform these steps for new messages $((\textsc{Finalize}, \mathtt{sid}, H_j, \pi_j, L_j), t_j) \in \mathsf{TO}_i$ such that $t_j > \hat{t}$, where $\hat{t}$ is the highest ledger time registered in the previous version of $\mathsf{STO}_i^{\mathtt{sid}}$. Finally, P_i outputs $\bowtie_{\mathtt{sid} \in \mathsf{SHDS}_i} \mathsf{STO}_i^{\mathtt{sid}}$.

On input $(\textbf{AddShard}, \mathtt{sid})$**:** Party P_i sets $\mathsf{SHDS}_i := \mathsf{SHDS}_i \cup \{\mathtt{sid}\}$.

On input $(\textbf{RemoveShard}, \mathtt{sid})$**:** Party P_i sets $\mathsf{SHDS}_i := \mathsf{SHDS}_i \setminus \{\mathtt{sid}\}$.

Theorem 2. *Protocol $\Pi_{\text{BD-STL}}$ described above UC-realizes functionality $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ in the $(\mathcal{F}_{\text{BD-TL}}^{\Delta}, \mathcal{F}_{\text{REPO}}, \mathcal{F}_{\text{COMSEL}}, \mathcal{F}_{\text{SHARD}}^{n_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_1}, \ldots, \mathcal{F}_{\text{SHARD}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell})$-hybrid model in the partially synchronous model (i.e., where $\Delta_1, \ldots, \Delta_\ell, \Delta'$ are unknown but finite) with security against active static adversaries.*

Proof. In order to prove this theorem, we construct a simulator $\mathcal{S}$ such that no PPT environment $\mathcal{Z}$ can distinguish an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ from a real execution of $\Pi_{\text{BD-STL}}$ with any adversary $\mathcal{A}$. $\mathcal{S}$ executes $\Pi_{\text{BD-STL}}$ with an internal copy of the adversary $\mathcal{A}$, forwarding all inputs from $\mathcal{Z}$ to $\mathcal{A}$. $\mathcal{S}$ simulates functionalities $\mathcal{F}_{\text{BD-TL}}^{\Delta}, \mathcal{F}_{\text{REPO}}, \mathcal{F}_{\text{COMSEL}}, \mathcal{F}_{\text{SHARD}}^{n_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_1}, \ldots, \mathcal{F}_{\text{SHARD}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$ towards $\mathcal{A}$ by following the exact instructions of these functionalities unless explicitly stated otherwise.

$\mathcal{S}$ executes the Shard Management and Shard Operations steps of $\Pi_{\text{BD-STL}}$ with $\mathcal{A}$ exactly as an honest party would. In particular, $\mathcal{S}$ follows the exact instructions of $\mathcal{F}_{\text{REPO}}$ and $\mathcal{F}_{\text{COMSEL}}$ when simulating these functionalities. In the case of $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ and the functionalities $\mathcal{F}_{\text{SHARD}}^{n_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ corresponding to each shard, $\mathcal{S}$ simulates adversarial commands $(\textsc{Add}, \cdot)$ according to $\mathcal{A}$'s behavior. This ensures that shards are (re-)started and finalized w.r.t. to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ in the same way as in a real execution of $\Pi_{\text{BD-STL}}$.

Upon receiving a message $(P_i, \mathtt{sid}, m)$ from $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ indicating that an honest party has sent a message m to shard $\mathtt{sid}$, $\mathcal{S}$ simulates the corresponding honest party sending m to shard $\mathtt{sid}$ by following the steps of an honest party in $\Pi_{\text{BD-STL}}$. When m appears in the simulated $\mathcal{F}_{\text{REPO}}$ along a valid finalization message in the simulated $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ (i.e., with a valid finalization proof w.r.t. to the simulated $\mathcal{F}_{\text{SHARD}}^{n_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ corresponding to shard $\mathtt{sid}$), $\mathcal{S}$ sends $(\textsc{Add}, \mathtt{sid}, m, t)$ to $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$, where t is the time when the finalization proof corresponding to m appeared in the simulated $\mathcal{F}_{\text{BD-TL}}^{\Delta}$.

As the execution with $\mathcal{A}$ progresses, $\mathcal{S}$ checks whether new messages are finalized in the simulated $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$. If these messages were not sent by $\mathcal{S}$ on behalf of a simulated honest party, $\mathcal{S}$ adds them to $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$. When m appears in the simulated $\mathcal{F}_{\text{REPO}}$ along a valid finalization message in the simulated $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ (i.e., with a valid finalization proof w.r.t. to the simulated $\mathcal{F}_{\text{SHARD}}^{n_h,\mathcal{S}_h,\mathcal{L}_h,\Delta_h}$ corresponding to shard $\mathtt{sid}$) such that m was not input by $\mathcal{S}$ (in which case $\mathcal{S}$ took care of this message by following the previous steps), $\mathcal{S}$ sends $(\textsc{Add}, \mathtt{sid}, m, t)$ to $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$, where t is the time when the finalization proof corresponding to m appeared in the simulated $\mathcal{F}_{\text{BD-TL}}^{\Delta}$.

Notice that the execution with $\mathcal{S}$ is exactly the same as in the real execution. Following these steps, $\mathcal{S}$ ensures that messages finalized for each shard in the simulated execution of $\Pi_{\text{BD-STL}}$ match the same messages in $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$. However, messages are only added to $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ after they are finalized in the simulated execution of $\Pi_{\text{BD-STL}}$. Hence, the delay Δ' must be such that finalizing a message m sent to a shard $\mathtt{sid}$ of $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ in the simulated execution of $\Pi_{\text{BD-STL}}$ takes at most Δ' clock ticks. We remark that Δ' is guaranteed to be finite since it is equal to the delay Δ from the simulated $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ plus the worst case delay to find a live instance in the gearbox $\mathcal{F}_{\text{SHARD}}^{n_1,\mathcal{S}_1,\mathcal{L}_1,\Delta_1}, \ldots, \mathcal{F}_{\text{SHARD}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell,\Delta_\ell}$), which is guaranteed to be finite as per the analysis in

the beginning of this section. Hence, no PPT environment $\mathcal{Z}$ can distinguish an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\text{BD-STL}}^{\Delta'}$ from a real execution of $\Pi_{\text{BD-STL}}$ with any adversary $\mathcal{A}$. □

6.2 Extensions

While Protocol $\Pi_{\text{BD-STL}}$ realizes a sharded ledger $\mathcal{F}_{\text{BD-STL}}$, it can have its efficiency and security improved by extending the way it switches gears over functionalities $\mathcal{F}_{\text{SHARD}}^{n_1,\mathcal{S}_1,\mathcal{L}_1}, \ldots, \mathcal{F}_{\text{SHARD}}^{n_\ell,\mathcal{S}_\ell,\mathcal{L}_\ell}$ for each shard. Here we describe some of these extensions informally.

6.2.1 Damping

So far we only showed how to move up the gearbox of consensus algorithms. That is enough to prove eventual liveness. In practice one also wants to have a way to regain efficiency if the loss of liveness was due to some temporary event such as a burst error in the network. This can be achieved using some heuristic. The timestamps on the control chain can be used to determine the uptime of the shards, and if the uptime exceeds some heuristic threshold, one tries to move down the gearbox again. This will tend to find the optimal position in the gearbox producing only some acceptable downtime. Since safety is never violated, any heuristic can be used that works well in practice.

6.2.2 Cycling Committees

Our analysis so far has assumed static corruptions. To tolerate slowly adaptive corruptions, one can resample committees repeatedly. This can be combined with the damping described above, i.e., one can close and reopen shards after some timeouts and possibly change gears when resampling. This is secure as long as the time it takes to corrupt a party is longer than the resampling timeout.

6.3 Inter-Shard Transactions and Communication

While our protocol allows for parties to read and write from any shard, doing so would nullify the benefits of a sharded ledger. We finally describe how to add causal communication and payments across shards while only requiring parties to watch one shard. The same mechanism can be used for smart contracts on one shard to send signals to smart contracts on other shards. For this to be meaningful, we would need a notion of transactions and accounts (or `UTXO` [25]), which we have not included in the above formalism for clarity. Doing a full formal treatment is out of the scope of this paper, where we want to focus on how to work around the safety-liveness dichotomy. However, we sketch the chosen solution for completeness, as there are some special considerations for a sharding scheme with shards which might "crash".

We need a notion of some shard "deciding" to send a message to another shard. This might just be a message from a smart contract to another. It might also be a transfer of stake from one shard to another. Finally, it might be the transfer of a UTXO from one shard to another: it gets burned on one shard and appears on another one. We solve all these cases using a uniform notion of inter-shard signals, where the receiving shard can be sure the signal came from the sending shard. For this purpose we introduce on each shard an efficiently computable function Π_i which interprets the sequence of messages on the shard. It could for instance keep track of how much stake is on each account, and interpret the messages on the shard as commands moving the stake or UTXO around. For our purposes, however, we only need that it can sometimes inject into the sequence of messages, a special message of the form $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ meaning that m has to be sent from Shard_i to Shard_j. If we can move such messages, the function Π_j can

then interpret m as adding money to some receiving account if needed. Here we focus only on moving the signals. In a bit more detail, for each sequence σ of messages on a shard, Π_i creates a super-sequence $\Pi_i(\sigma)$, i.e., σ is a sub-sequence of $\Pi_i(\sigma)$. The extra messages are interpreted as the injected ones. We assume that messages are only injected "at the end". Formally, if σ is a prefix of σ', then $\Pi(\sigma)$ is a prefix of $\Pi(\sigma')$. We are only interested in injected messages of the form $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$. The main properties we want to guarantee are:

Delivery: If the message $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ appears on Shard_i, then eventually $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ appears on Shard_j.

Validity: If the message $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ appears on Shard_j, then eventually $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ appears on Shard_i.

Causality: Following [23], we say that M $\mathsf{happens\ before}$ M' if M appears before M' on the same shard, or $M = (\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ appears on Shard_i and $M' = (\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ appears on Shard_j. Take the transitive closure of this relation. Then this must be a strict partial order; in particular, there are no loops.

These three requirements together ensure that the global sharded ledger with the injected messages forms a causal system.

There are three main approaches for inter-shard communication in the literature (for an overview of different solutions, see [30, 1, 27]): (1) In control-chain (CC) driven approaches, the CC is used to pass the messages, and the order on the CC is used for causality. (2) In client-driven approaches it is the client that moves the signal. Client-driven approaches put an unwanted responsibility on the clients; e.g., a shop using a blockchain for payments would have to take responsibility in running it, what is undesirable in practice. Moreover, it is also hard to give any proven guarantees in such a setting. (3) Finally, in shard-driven approaches the signal is moved by the nodes running the shard.

In a setting where shards can be taken down and brought up dynamically, a shard driven approach is met with many of the same challenges as getting *exactly-once delivery* in a system where both the sender and the receiver can crash. We therefore opted for a CC driven mechanism. To not overload the CC, we only post hashes on the CC and store the actual signals in a repository where the receiving committee can retrieve them. We sketch a protocol next.

Inter-shard communication protocol. When a shard Shard_i computes the finality proof which is posted to the CC as a heart beat, instead of reporting the sequence σ it reports $\Pi_i(\sigma)$. The committee of the receiving shard Shard_j will retrieve $\Pi_i(\sigma)$ from the repository, authenticate it against the heart beat on the CC, and then run through $\Pi(\sigma_i)$ to find all $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ transactions in the finalized sequence. The inter-shard transactions will be treated as normal transactions and will be added to the shard (if not already included). When all transactions of the heart beat have been added to Shard_j, then Shard_j will report this as meta data in its own next heart beat. More formally, the shards use Π_j to inject the message in $\Pi_j(\sigma_j)$ such that it automatically becomes part of the next heart beat.

In a bit more detail, each shard Shard_j keeps a vector Clock_j, where $\mathsf{Clock}_j(\mathsf{Shard}_i)$ is the newest heart beat of Shard_i on CC which has been fully processed by Shard_j. The value $\mathsf{Clock}_j(\mathsf{Shard}_j)$ is the set of new messages to send. When $\mathsf{Clock}_j(\mathsf{Shard}_i)$ changes, it is then injected into $\Pi_j(\sigma_j)$. When a shard runs (possibly after a "reboot") it can find its own newest Clock_j and then it inspects the CC; for each Shard_i which has a heart beat on CC newer than $\mathsf{Clock}_j(\mathsf{Shard}_i)$ it retrieves the corresponding inter-shard signals from the repository and starts adding the missing ones to its own shard. When done it updates $\mathsf{Clock}_j(\mathsf{Shard}_i)$.

Each shard Shard_i will maintain a Lamport clock [23] c_i and use it to timestamp each message with an increasing timestamp. When an inter-shard message $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ is sent, its Lamport timestamp c on the sending shard is included. When $(\mathsf{Shard}_i, \mathsf{Shard}_j, m)$ is included on Shard_j it gets timestamp $\max(c_j, c)$ and c_j is updated to $\max(c_j, c) + 1$. This implements a causal, global strict partial order on all messages.

We note that there are obvious possible optimizations. The authenticated data structure used for the heart beat should allow to efficiently retrieve the vector clocks and the set of messages intended for a particular shard. It is also possible to add an optimistic mode which allows shared-based intershard communication of final parts of a shard that was not yet committed to the CC. However, these optimisations are out of scope for this paper.

Inter-shard signals can be used to move stake and UTXO. We take UTXO as an example. The sending signal burns the UTXO on the sending shard and only adds the UTXO to the signal for the receiving shard if the UTXO was unspent at burn time. The receiving signal recreates the UTXO on the receiving chain. This ensure a UTXO only appears as unburned on one shard at a time.

7 Shard Safety-Liveness Dichotomies

In this section, we present the shard safety-liveness dichotomies (SSLD). They follow from basic quorum based proof techniques from Byzantine agreement theory, but we present them explicitly here for the shard setting for completeness. We want to prove the if S is the fraction of corruptions that can be tolerated without breaking safety and L is the fraction of corruptions that can be tolerated without breaking liveness, then $L + S < 100\%$ in the synchronous case and $2L + S < 100\%$ in the partially synchronous case. Note that the Dolev-Strong broadcast protocol [14] achieves synchronous broadcast for $L = s = 99\%$, so it seemingly violates or bound $L + S < 100\%$. However, Dolev-Strong only achieves internal agreement among the n servers. External parties cannot verify the value agreed on. The crucial thing about a shard is that external parties can post on it and read from it. A shard is not run by all parties. It is therefore not enough for committee members to be able to agree among themselves on the ledger. They must be able to convince a third party about the value of the ledger.

To exploit this in the lower bound we need to model readers and writer. We will go for a minimal model with a single writer W and several readers R. Besides this there will be n committee members $\mathcal{C} = \{\mathsf{C}_1, \ldots, \mathsf{C}_n\}$. We assume they are fixed for the lower bound.

7.1 Synchronous, Unauthenticated SSLD

As a warm-up we first look at a very minimal model where the writer can chose to send a bit on the shard and the reader can read it, if it was posted.

A writer node posts to the shard by sending the same bit to all C_i. The writer and reader do not speak to each other, they only speak to $\mathcal{C}$. We can assume that first W sends a single bit to each C_i and then leaves. Letting W take interactive part in the protocol would *de facto* make it another server and we would get other bounds. We assume that the reader node reads by getting a message from each C_i and then R maybe outputs a bit. Letting R take interactive part in the protocol would *de facto* make it another server and we would get other bounds. The committee members will talk between themselves.

We are interested in when we can get liveness and safety. Liveness means R outputs something. Safety means that if W is honest and sends the same bit to all servers then this is the bit that R will output if it outputs something. It is clear this is not enough to have a blockchain, but having low expectations makes the lower bound stronger.

We assume a monotone liveness structure $\mathcal{L}$, a set of subsets of $\mathcal{C}$. By monotone we mean that if $L \in \mathcal{L}$ and $L' \subset L$ then $L' \in \mathcal{L}$. We also assume a monotone safety structure $\mathcal{S}$. Let C be the set of corrupted parties. We only want liveness if we have safety, so we assume $\mathcal{L} \subset \mathcal{S}$. With this we can make the following minimal requirements.

Liveness If W sends the same bit b to all C_i and $C \in \mathcal{L}$ then eventually R outputs a bit.

Safety If W sends the same b to all C_i and R outputs some c and $C \in \mathcal{S}$, then $c = b$.

For a synchronous protocol we can prove that it cannot be the case that there exist $S \in \mathcal{S}$ and $L \in \mathcal{L}$ such that $S \cup L = \mathcal{C}$.[7] To see this consider two disjoint sets $S \in \mathcal{S}$ and $L \in \mathcal{L}$ such that $S \cup L = \mathcal{C}$. If they are not disjoint we can make them smaller with monotonicity until they are disjoint and still in $\mathcal{C}$ and $\mathcal{L}$.

Consider two experiments.

Experiment 1: Let W send 0 to all servers. Corrupt L and let all parties in L run with input 1. Since we corrupted from $\mathcal{L}$ the reader R will give an output. Call it b_1. Since $\mathcal{L} \subset \mathcal{S}$ the output will be $b_1 = 0$.

Experiment 2: Let W send 1 to all servers. Corrupt S and let all parties in S run with input instead 0. Since we corrupted from $\mathcal{S}$ the reader R will only output 1. From the point of view of the reader the experiment 2 is identical to experiment 1. So we know that R *does* give an output. So it outputs $b_2 = 1$.

We conclude that $0 = b_1 = b_2 = 1$, a contradiction.

7.2 Synchronous, Authenticated SSLD

The above proof did not take care of the fact that W might sign its bit to prevent corrupt servers from changing its input. We now cover this case too. To get a lower bound in this case we will need to also require agreement on the order of messages. To prove the bound we will then let a corrupt W sends signed 0's to some servers and signed 1's to other servers and show that the receivers cannot agree on which bit appeared on the shard *first*.

We assume the committee knows the public key of W for a signature scheme and that W has the secret key.

As before W only writes to the shard and R only reads. The writer W posts to the shard by sending the same signed bit to all C_i. Then the committee members will talk between themselves defining a sequence of signed bits having been posted. For our proof it is enough to consider ledgers of length at most 1. So the ledger is empty or has a single message. In other words, we prove that it is even impossible to agree on just the first element of the ledger. We assume that R at some point reads from each committee member and possibly computes a single output message m. The readers do not communicate. If they did, they would *de facto* become servers and the bounds would change. We allow that R does not give an output. Think of it as the ledger currently being empty.

We again have monotone liveness structure $\mathcal{L}$ and monotone safety structure $\mathcal{S}$ and require that $\mathcal{L} \subset \mathcal{S}$. Let C be the set of corrupted parties. We require the following.

Liveness If W sends the same signed m to all C_i and $C \in \mathcal{L}$, then R_i can eventually read $m_i = m$.

[7] Notice that if we let $\mathcal{S}$ be all sets of servers of size s and let $\mathcal{L}$ be all sets of servers size ℓ, then $S \cup L \neq \mathcal{C}$ for disjoint sets translates into $s + \ell < n$. Dividing by n we get the simplified dichotomy $s/n + \ell/n < 100\%$ of the introduction.

Validity If W does not send a signed m to any server C_i and $C \in \mathcal{S}$ and R_i outputs m_i, then $m_i \neq m$.

Agreement If R_1 outputs m_1 and R_2 outputs m_2 and $C \in \mathcal{S}$, then $m_1 = m_2$.

Note that the way we phrase liveness it implies safety. Normally you would formulate liveness as a pure liveness property, but we assume $\mathcal{L} \subset \mathcal{S}$ and the above makes the proof simpler. Validity just means that the ledger cannot post anything that the writer did not sign. This is an essential requirement for the ledger to be meaningful. Agreement says that two different honest readers agree on what is the first message on the ledger if they both consider the ledger non-empty. This is again essential.

For a synchronous protocols we assume the parties have access to round-based communication, where if a party does not send a message in a round, then instead NoMsg is delivered.

For a synchronous protocol we can prove that it cannot be the case that there exist $S \in \mathcal{S}$ and $L \in \mathcal{L}$ such that $S \cup L = \mathcal{C}$. To see this consider two disjoint sets $S \in \mathcal{S}$ and $L \in \mathcal{L}$ such that $S \cup L = \mathcal{C}$.

Let W sign 0 and 1. When we say that b is given as input we mean that the signature is given along. Let the output of a reader R_i be the first bit b_i it sees appearing on the ledger.

Experiment 1: Let W send 0 to all servers. Run with R_0. Corrupt L and drop all messages between L and S. I.e., L sends NoMsg and will act as if S did the same. Since we corrupted from $\mathcal{L} \subset \mathcal{S}$ the reader R_0 will get output $b_1 = 0$ by liveness and validity.

Experiment 2: Let W send 0 to all servers and also send 1 to L. Run with R_2. Corrupt L and drop all messages between L and S. Since we corrupted from $\mathcal{L}$ the reader R_2 will get some output b_2 by liveness.

Assume for the sake of contradiction that $b_2 = 1$ with constant positive probability p. Then we can break safety as follows. Let W send 0 to all servers and also send 1 to L. Corrupt L and make in run two copies of L. One running with input 0 and one with input 1, call them L_0 and L_1. Drop all messages between L_0 to S. Drop all messages between L_1 to S. Clearly L can run both copies as they do not interact with S, the view of S is the same with both of them. Show L_0 to R_0 and show L_1 to R_1. Now we break agreement with probability p. So we can assume $b_2 = 0$.

Experiment 3: Let W send 1 to all servers. Run with R_3. Corrupt L and drop all messages between L and S. Since we corrupted from $\mathcal{L}$ the reader R_3 will get output $b_3 = 1$ by liveness and validity.

Experiment 4: Let W send 1 to all servers and also send 0 to L. Run with R_4. Corrupt L and drop all messages between L and S. Since we corrupted from $\mathcal{L}$ the reader R_1 will get some output b_4 by liveness. We can conclude that $b_4 = 1$ as above.

Experiment 5: Let W send 0 and 1 to all servers. Run with R_5. Corrupt S and drop all messages between L and S. Let S ignore the 0 input. This experiment is identical to experiment 4 so the output of R_5 is $b_5 = b_4 = 1$.

Experiment 6: Let W send 0 and 1 to all servers. Run with R_6. Corrupt S and drop all messages between L and S. Let S ignore the 1 input. This experiment is identical to experiment 2 so the output of R_6 is $b_6 = b_2 = 0$.

We are now again ready to break agreement. The difference between experiment 5 and 6 is whether S drops 0 or 1. Since it does not talk to L it can run both experiments in the head and show experiment 5 to R_5 and show experiment 6 to R_6.

7.3 Partially Synchronous, Authenticated SSLD

We finally look at the partially synchronous SSLD. For a partially synchronous protocol all messages are delivered within some unknown delay Δ_{NET}. We use the same liveness and safety properties as for the synchronous, authenticated SSLD. Assume we have a partially synchronous shard with $\mathcal{L}$-liveness and $\mathcal{S}$-safety. We can prove that it cannot be the case that there exist disjoint sets $S \in \mathcal{S}$ and $L_0, L_1 \in \mathcal{L}$ such that $L_0 \cup L_1 \cup S = \mathcal{C}$.[8] Assume for the sake of contradiction that we have such sets.

Consider the following experiment. Give L_0 input 0. Give L_1 input 1. Delay all messages between L_0 and L_1 for time $\Delta_{\text{NET}} = \infty$. This is not allowed in the partially synchronous model, but we will lower Δ_{NET} to a large enough finite value below.

Let a corrupt S run as follows. Towards L_0 it runs an honest copy of S with input 0. Call it S_0. Towards L_1 it runs an honest copy of S with input 1. Call it S_1. We consider two readers R_0 and R_1. When R_0 reads delay message from L_1 for time $\Delta_{\text{NET}} = \infty$ and let S reply as S_0 would. When R_1 reads delay message from L_0 for time $\Delta_{\text{NET}} = \infty$ and let S reply as S_1 would.

Consider the view of R_0. It is interacting with honest S_0 and L_0 and with L_1 being infinitely later. This corresponds to a corruption of $L_1 \in \mathcal{L}$ where we let L_1 send no messages, so eventually R_0 will output 0. Say this happens within time E_0. Now set $\Delta_{\text{NET}} = E_0 + 1$, run L_1 honestly but instead delay all messages from L_1 for time Δ_{NET} as allowed in a partially synchronous run. In this run L_0, S_0, R_0 all have the same view of L_1 as when L_1 is corrupted and sends no messages, so R_0 still outputs $m_0 = 0$.

Consider then the view of R_1. It is interacting with honest S_1 and L_1 and with L_0 being infinitely later. This corresponds to a corruption of $L_0 \in \mathcal{L}$ so R_1 will output 1. Say this happens within time E_1. Now set $\Delta_{\text{NET}} = \max(E_0, E_1) + 1$, run L_1 honestly but delay all messages from L_1 for time Δ_{NET}. In this run L_1, S_1, R_1 all have the same view of L_0, so R_1 still outputs $m_1 = 1$.

Since we now consider a valid partially synchronous run with $\Delta_{\text{NET}} = \max(E_0, E_1) + 1$ and corruption $S \in \mathcal{S}$ it follows that if R_0 eventually outputs m_0 and R_1 eventually outputs m_1, then $m_0 = m_1$. We conclude that $0 = m_0 = m_1 = 1$, a contradiction.

References

[1] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. Divide and scale: Formalization of distributed ledger sharding protocols. *CoRR*, abs/1910.10434, 2019.

[2] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In Kan Zhang and Yuliang Zheng, editors, *ISC 2004*, volume 3225 of *LNCS*, pages 61–72. Springer, Heidelberg, September 2004.

[3] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Katz and Shacham [19], pages 324–356.

[4] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, The University of Guelph, Guelph, Ontario, Canada, 6 2016.

[5] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[8] Notice that if we let $\mathcal{S}$ be all sets of servers of size s and let $\mathcal{L}$ be all sets of servers size ℓ, then $S \cup L_0 \cup L_1 \neq \mathcal{C}$ for disjoint sets translates into $s + 2\ell < n$. Dividing by n we get the simplified dichotomy $s/n + 2(\ell/n) < 100\%$ of the introduction.

[6] Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

[7] Ignacio Cascudo and Bernardo David. SCRAPE: Scalable randomness attested by public entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 537–556. Springer, Heidelberg, July 2017.

[8] Ignacio Cascudo and Bernardo David. Albatross: publicly attestable batched randomness based on secret sharing. Cryptology ePrint Archive, Report 2020/644, 2020. To Appear at Aisacrypt 2020.

[9] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

[10] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. https://eprint.iacr.org/2016/919, To Appear in the Proceedings of Financial Crypto 2019.

[11] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.

[12] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.

[13] Thomas Dinsdale-Young, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort: A partially synchronous finality layer for blockchains. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 24–44. Springer, 2020.

[14] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 401–407. ACM, 1982.

[15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

[16] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323. Springer, 2017.

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

[18] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[19] Jonathan Katz and Hovav Shacham, editors. *CRYPTO 2017, Part I*, volume 10401 of *LNCS*. Springer, Heidelberg, August 2017.

[20] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Katz and Shacham [19], pages 357–388.

[21] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy*, pages 583–598. IEEE Computer Society Press, May 2018.

[22] Jae Kwon. Tendermint: Consensus without mining. manuscript, 2014. https://tendermint.com/static/docs/tendermint.pdf.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[24] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 17–30. ACM Press, October 2016.

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. manuscript, 2009. http://www.bitcoin.org/bitcoin.pdf.

[26] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 39:1–39:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[27] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, pages 41–61. ACM, 2019.

[28] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*, pages 95–112. USENIX Association, 2019.

[29] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 931–948. ACM Press, October 2018.

[30] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. SoK: Communication across distributed ledgers. Cryptology ePrint Archive, Report 2019/1128, 2019. https://eprint.iacr.org/2019/1128.

Appendix

A Python Code for Computing Minimal Committee Sizes

```
import math
import scipy.special
import numpy
from fractions import Fraction

# Compute probability of having too many corrupted parties in committee with
# n = total population
# t = number of corruptions in total population
# s = committee size
# h = minimum number of honest parties required in committee
# pMax = maximal value for which pFail returns correct value. Output is 1 if pFail > pMax.
def pFail(n, t, s, h, pMax) :
  p = 0
  denom = scipy.special.comb(n, s, exact=True) # compute n choose s as exact integer
  for i in range(s - h + 1, s+1) :
    p += Fraction(scipy.special.comb(t, i, exact=True) * scipy.special.comb(n-t, s-i, exact=True), denom)
    if p > pMax :
      return 1
  return p

# Find minimum committee size with corruption ration cr such that pFail <= 2^{-k}
def minCSize(n, t, cr, k) :
  pMax = Fraction(1, 2**k)
  for s in range(1, n+1) :
    h = math.ceil((1 - cr) * s) # we want at least h honest parties to not violate corruption threshold
    if pFail(n, t, s, h, pMax) <= pMax :
      return s

# Compute analytical upper bound
def analyticBound(n, t, cr, k) :
  p = Fraction(t, n)
  q = cr
  alpha = q - p
  beta = (math.e * math.sqrt(q) * (1-p)) / (2 * math.pi * alpha * math.sqrt(1-q))
  bound = math.ceil(k/math.log((q/p)**q * ((1-q)/(1-p))**(1-q), 2))
  betaBound = math.ceil(beta*beta)
  return max(bound, betaBound)
```

```
38
39 # Compute values for 10000 total parties, 30% corruption, and 60
       bit security
40 n = 10000
41 t = 3000
42 k = 60
43
44 # corruption threshold percentages we are interested in
45 crps = range(99, 32, -1)
46
47 print("cr\t min\t bound")
48 for crp in crps:
49   cr = Fraction(crp, 100) # convert percentage to fraction
50   print(float(cr), "\t", minCSize(n, t, cr, k), "\t",
         analyticBound(n, t, cr, k))
```