GearBox: Optimal-size Shard Committees by Leveraging the Safety-Liveness Dichotomy

Bernardo David*[1], Bernardo Magri†[2], Christian Matt[3], Jesper Buus Nielsen‡[4], and Daniel Tschudi[3]

[1]ITU Copenhagen, bernardo@bmdavid.com
[2]The University of Manchester, UK, bernardo.magri@manchester.ac.uk
[3]Concordium, Zurich, {cm, dt}@concordium.com
[4]Concordium Blockchain Research Center, Aarhus University, jbn@cs.au.dk

July 5, 2022

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 with at most 30% corruption and 60-bit security, previous designs required over 5800 parties in each shard to guarantee security. Our design requires only 1713 parties in the worst case with maximal corruption, and in the optimistic case works with only 35 parties without compromising security.

Moreover, in this highly concurrent execution setting, it is paramount to guarantee that both the sharded ledger protocol and its sub protocols (i.e., 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).

†Work partially done while the author was at the Concordium Blockchain Research Center, Aarhus University.

‡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

Since the introduction of Bitcoin [Nak09], 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 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 with a total of 10K parties, the minimal shard size that guarantees at most 33% corruption with 60-bit security[2] in a randomly selected shard is 5886, which hardly leads to a major improvement in efficiency. As we show in Section C, the bounds are almost perfectly linear in the security parameter, so for 30-bit security the size would have to be about 3000. Thus, to get to a shard size in the hundreds, unsatisfying security levels would have to be adopted.

The security of a blockchain system 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 and the safety threshold S are the levels of corruption under which liveness and safety are guaranteed, respectively. Existing sharding solutions [LNZ+16, ZMR18, KJG+18] need to guarantee security of the shards (both liveness and safety) in the worst-case corruption scenario, thus forcing them to consider equal bounds for liveness and safety, what severely constrains the size of shards. In this work, we overcome this apparent barrier by leveraging the *shard safety-liveness dichotomy* and choosing different safety and liveness bounds. We devise a protocol where shards are always safe but can eventually lose liveness; when that happens the shards are respawned with adjusted liveness and safety parameters. This allows for significantly smaller shards.

The shard safety-liveness dichotomy defines the possible tuples (L, S) of liveness and safety thresholds for which a (shard) consensus protocol provides security. For partially synchronous protocols, the safety-liveness dichotomy says that $2L + S < 100\%$. Hence in the case of $S = L$ it must be that $L, S < 33\%$. For synchronous protocols the safety-liveness dichotomy says that $L + S < 100\%$, hence whenever $S = L$ it must be that $L, S < 50\%$. These dichotomies can be derived from the following observations. For the synchronous dichotomy, we use the crucial fact that external parties can post on and read from the shards. We observe that if a reader can be convinced about the state of a shard while a subset of (relative) size L is crashed, then it means that a subset of size $1 - L$ can convince the reader. Thus if $S = 1 - L$, this means that a reader can be convinced after only talking to a set of potentially corrupt servers. 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 subset. Moreover, if a subset of size L is crashed, a reader must by definition be able to make a decision. Hence, a reader that makes a decision (say, on which message it saw on the shard first) should be able do this without having heard from a fraction L of the honest parties. This means that from the fraction $L + S$ of the parties it heard, a fraction L or a fraction S might be corrupted. The shard dichotomies discussed here follow from standard arguments, and for completeness we provide formal proofs in Appendix A.

[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 4.1 for a further discussion.

[2] A security level of 60-bit means that security holds with probability at least $1 - 2^{-60}$.

1.1 Our Contributions

We present a novel sharding approach that leverages the safety-liveness dichotomy to get the smallest possible shard committees without sacrificing safety. Our sharding design has security against a fraction of $t < 1/3$ corrupt shard committee members in the partially synchronous setting.[3]

The shards will, in the optimistic model, start to run with a low liveness threshold and a high safety threshold, e.g., $S = 89\%$ and $L = 5\%$. Being safe against up to 89% corruption allows for sampling much smaller committees, however being live only against up to 5% corruption makes it a lot more likely for a shard to deadlock. For this, we use an approach where an independent ledger, that we call control-chain (CC), manages the shards by constantly monitoring them for liveness. This is done by letting the shards post "heart beat" transactions on the CC. The CC can then "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 is found which gives liveness (and safety). On the other hand, if no deadlocks are detected within a certain period, shards can be dynamically scaled down, leading to optimal shard sizes for the current corruption ratio. Crucially, at no point safety is compromised.

Our design has for each shard what we call a "gearbox" of consensus protocols: shards at the top are larger (therefore slower) but have robust liveness, while shards at the bottom are small (therefore faster) but have a lower liveness. The CC changes gear upwards in the gearbox when deadlocks are detected switching to a larger shard, 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 just restarted with the same parameters and a new random committee. 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. This approach allows us to select the best committee size given the *unknown* actual corruption threshold, albeit at the cost of resampling the committee until liveness is achieved.

Next, we describe two different ways to instantiate our framework.

Partially synchronous. One can run with a partially synchronous CC and partially synchronous shards. At the bottom gear one could have $L = 0\%$ and $S = 99\%$. For a population of 10K peers and assuming an overall corruption level of at most 30%, this would give a committee size of 35 guaranteeing that safety is not violated except with probability 2^{-60}. At the top gear one could use a consensus protocol with $L = 30\%$ and $S = 39\%$. This would give a committee size of 1713 guaranteeing not 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 in the top gear. It is easy to see that we get at most 30% corruption with a constant probability, which gives eventual liveness. This already gives a significant improvement over existing designs that require that liveness only fails with negligible probability. Moreover, 30% corruption is a worst-case assumption and in typical executions, there will be much less corruption. In the other extreme with 0 corruption, the shards will already be live in the lowest gear with 35 parties. Even for more realistic 20% actual corruption, 207 parties per shard are sufficient to obtain a live shard with constant probability, dramatically improving the state of the art. See Section 4.3 for more details on required committee sizes in different settings.

[3] Although our techniques can be used to get security against a fraction of $t < 1/2$ corrupt committee members in the synchronous setting, we focus on the more desirable (and challenging) partially synchronous case.

Mixed. One can run a synchronous CC tolerating 49% corruption, and at the bottom of the gearbox we again start with a partially synchronous shard with $L = 0\%$ and $S = 99\%$. We run partially synchronous up until $L = 25\%$ and $S = 49\%$. After that, we then switch to a synchronous shard with $S = L = 49\%$ corruption. This allows a design tolerating 49% overall corruption, but running partially synchronous small shards until 25% corruption. This is interesting since partially synchronous protocols can achieve higher throughput in good network conditions by avoiding waiting for the end of rounds, as synchronous protocols do.

In the rest of the paper we focus on the partially synchronous setting, and therefore we stick 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.

Static vs. adaptive security. Protocols that rely on the honest majority of long living committees for security clearly fail if an adaptive adversary can instantly corrupt at least half of the committee. This is also true for our sharding protocol of Section 5. We point out that this is not a weakness of our design but to the best of our knowledge applies to all sharding protocols based on randomly chosen committees. While Free2Shard [RKTV22] tolerates fully adaptive corruption, it assumes a different security model, and thus cannot directly be compared to our protocol.

If we assume the adversary can adaptively corrupt parties, but only after some delay [MNT22], one can obtain security by periodically resampling committees, see Section 5.4.

Cross-Shard communication. Cross-shard transactions have extensively been studied in the literature [ZABZ+21, AKW19, WSNH19]. For our sharding approach, special care needs to be taken since shards could lose liveness during a cross-shard transaction. We show in Section 5.5 on the example of Atomix [KJG+18], how existing protocols can be adapted to work in our setting. The basic idea is to leverage the control chain to finalize messages on the shards. This can be done without an additional overhead by including Merkle tree hashes of the relevant transactions in the heartbeats that are regularly posted on the control chain. This ensures that the amount of data posted on the control chain is independent of the number of cross-shard transactions, and short Merkle proofs can be used to prove finality of transactions on shards that holds even under restarting shards.

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.

1.2 Technical Overview

The main building blocks of our sharding protocol are a Control Chain and Shard ledgers, which we discuss below and describe in detail in Section 3.

Control chain. The control chain (CC) is used to orchestrate the sharding protocol, storing shard management metadata. The CC is modeled as a timed ledger functionality that orders and timestamps incoming messages. More formally, the control-chain is a totally-ordered broadcast with persistency and a timestamp guarantee. The CC is executed by all parties but it only stores metadata of size *independent* from the contents of the shards. We show in Section 6.1 and Appendix B how one can realize a control chain using different existing blockchain protocols.

Shards. Shards are again modelled as a ledger functionality parameterized by the size of the shard committee and an adversary structure for the liveness guarantee. This allows us to instantiate shards with committees of different sizes and with different liveness guarantees. A shard consensus protocol is only executed by a small shard committee. As we propose a general approach to sharding, our shard functionality can be readily instantiated by standard blockchain or permissioned consensus protocols. In Section 4.2, we show how to leverage specific properties of our approach to achieve a particularly efficient instantiation of the shard functionality using a simple BFT-style consensus protocol with a leader that proposes blocks. Our protocol does not guarantee liveness if the leader is corrupted as we can always recover using the control-chain. This comes at the price of possibly having to resample committees more often and is in contrast to existing protocols, that come with complex mechanisms to resolve an unresponsive leader. To minimize resamplings, one can also use existing protocols that include leader replacement, as discussed in Section 4.2.

The protocol. The goal of our sharding protocol in Section 5 is to achieve a sharded ledger. Essentially, the sharded ledger consists of multiple copies of the ledger type used as control-chain. Our protocol starts by initializing all shards with the smallest possible committees, i.e., using the smallest liveness threshold. The initialization (and later restarting) of shards is done by posting a command on the CC. Once the shard committee responsible for that shard sees the command on the CC, it starts executing that shard.

At the core of our sharding protocols are the heartbeats that allow to asses the liveness of a shard. They are implemented by asking the shards to periodically post hashes of blocks on the CC as a way to timestamp and "finalize" the blocks. A block is considered valid if it is accompanied by a proof that at least one honest party agrees with it (e.g., signatures on the block by a sufficiently large number of parties in the shard committee). Each time a valid block is posted, a timeout is implicitly set. If the next valid block does not arrive before the timeout, then the shard is considered deadlocked. The timestamp is used to uniquely determine whether a shard made the time out.

If a shard is considered deadlocked, it is restarted with a fresh larger committee guaranteeing a slightly larger liveness threshold, thus increasing the chance that it will have liveness. By proceeding this way, the protocol ensures that a committee of close to optimal size is eventually selected for each shard. In other words, the size of shards is dynamically adjusted to match the actual corruption ratio in the network.

Note that such a sharded ledger can be trivially achieved (in UC) by simply instantiating multiple copies of the control-chain. We therefore emphasize that our sharding protocol is designed for the sake of efficiency. In the trivial solution every party needs to participate in each ledger copy, while our solution allows to select small committees that maintain a shard each. The selection of committees is discussed in Section 4.1.

1.3 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 refer the interested reader to the survey of Wang et al. [WSNH19] that gives a nice overview of the state-of-the-art in sharding protocols. Finally, we point out a common issue with all the proposals that hinders their practical usage.

There exists other approaches, such as Prism [BKT+19] and OHIE [YNHS20], to overcome blockchain scalability problems. While these two are orthogonal approaches and limited to the proof-of-work setting, our work focuses on more generic sharding solutions. For a general overview of blockchain scalability solutions, see [SC21].

Sharding protocols. To the best of our knowledge, Elastico [LNZ+16] 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 [LNZ+16] 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 [KJG+18]. This renders the protocol completely insecure when used with small committees.

Building upon Elastico's ideas, and improving it in many ways, OmniLedger [KJG+18] 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 efficiently[4] 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 [ZMR18] 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 [WW19] proposes a scale-out blockchain that contains many independent chains (called zones) running in parallel that divides the work-load 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. [AKW19] proposes a framework with security properties tailored for sharded ledger protocols, building upon the Bitcoin backbone model of Garay et al. [GKL15]. 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 [AKW19] is

[4] It can handle up to 33%, but with bad performance. See Footnote 2 in [KJG+18].

not composable, making it difficult to argue security of a sharded ledger protocol when combining it with a larger system.

Common issues. 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 4.3 we present some concrete numbers for the smallest size of committees needed to guarantee different honesty levels considering 60-bits of security. For example, with a total population of 10000 parties and at most 30% overall corruption, an honest supermajority (33% corruption) can only be guaranteed with 5886 parties per committee. Considering only 2000 parties in the total population requires 1716 parties per committee. And even reducing the assumption on the overall corruption to only 20% with 2000 parties in the total population still requires 540 parties in every shard.

Moreover, none of these previous works consider (or prove) security of sharding protocols under composability. This is a major shortcoming since sharding protocols (and blockchains in general) are mostly used as building blocks of larger systems (e.g., cryptocurrencies and smart contracts), which requires them to retain their security under composability.

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. We denote by $H\colon \{0,1\}^* \to \{0,1\}^\kappa$ a collision-resistant hash function.

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 [BMTZ17]. We further need a notion of time and access to clocks [KMTZ13], and we assume an idealized signature functionality [Can04, BH04]. 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. As discussed in the introduction, we consider the set of corrupted parties to be static. We discuss possible extensions for adaptive corruptions in Section 5.4.

3 Ledger Functionalities

3.1 (Sharded) Timed ledger

Timed ledger. A timed ledger $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ allows all parties to input messages, which will then be totally ordered and provided with a timestamp. More formally, every party P_i can input messages to the ledger, and retrieve a list TO_i of already ordered messages together with their timestamps. The ledger provides the following guarantees:

Persistence: For the lists of messages TO_i and TO_j seen by honest parties P_i and P_j, respectively, TO_i is a prefix of TO_j or vice versa.

Liveness and bounded timestamps: For any message m input at time t by an honest party, there is a time $t' \leq t + \Delta$ such that by time $t' + \Delta$ the pair (m, t') is in TO_i for any honest P_i. The maximal delay Δ is fixed, but unknown to honest parties.

The timestamp property ensures that 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.

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 delay assumed in the underlying network). Formally, we define the functionality $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ as a special case of a sharded timed ledger with a single shard, i.e., $\mathcal{F}_{\text{BD-TL}}^{\Delta} \coloneqq \mathcal{F}_{\text{BD-STL}}^{\Delta,1}$. See below for the definition of $\mathcal{F}_{\text{BD-STL}}^{\Delta,\nu}$.

Realizing a timed ledger. The timed functionality can be implemented by a BFT consensus such as HotStuff [YMR+19], Algorand [CM19] or Tendermint [Kwo14, Buc16], or by a Nakamoto-style blockchain such as Bitcoin [Nak09]. In case a Nakamoto-style blockchain is used, combining it with a finality layer, e.g., Casper the Friendly Finality Gadget [BG17] or Afgjort [DYMM+20], is advisable to improve the finality time. See Section 6.1 and Appendix B for more details on possible implementations of $\mathcal{F}_{\text{BD-TL}}^{\Delta}$.

Sharded timed ledger. A sharded timed ledger is simply a collection of multiple timed ledgers running in parallel, where each individual ledger is refereed to as a shard. For simplicity, we assume the number of shards ν is fixed. In practice, it is desirable to have the ability to start new shards. This can be done easily with our techniques (similarly to how deadlocked shards are restarted), but we omit this to simplify the modelling.

Every shard is identified by a shard identifier $\texttt{sid} \in \{1, \ldots, \nu\}$. Parties can send message to different shards using the shard identifier. Furthermore, every party P_i can retrieve a totally ordered list of messages in a shard $\texttt{sid}$, denoted by $\mathsf{TO}_i^{\texttt{sid}}$. Within each shard, the same

guarantees as described above for the timed ledgers apply. Using the timestamps on each shard, parties can also merge the total orders of every shard to obtain a global total order. Inter-shard communication can also be implemented on top of a sharded timed ledger as discussed in Section 5.5.

Next, we formally define the $\mathcal{F}_{\text{BD-STL}}^{\Delta,\nu}$ functionality.

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

Initialization

1: **for** $P_i \in \mathcal{P}$ and all $\mathtt{sid} \in \{1, \ldots, \nu\}$ **do**
2: $\quad \mathsf{TO}_i^{\mathtt{sid}} := ()$ *// Shard Total Order of shard* $\mathtt{sid}$ *for* P_i
3: $\quad \mathsf{Sent}^{\mathtt{sid}} := \varnothing$ *// Messages sent to Shard* $\mathtt{sid}$

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

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

Input: $(\textsc{Get}, \mathtt{sid})$.
3: Output $(\textsc{Get}, P_i)$ to the adversary. *// Inform adversary.*
4: Return $\mathsf{TO}_i^{\mathtt{sid}}$.

Interface for adversary

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

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

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

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 $(m, t') \in \mathsf{TO}_i^{\mathtt{sid}}$ for all honest P_i.

Notes about model and UC. 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 [GKL17], the formalization via $\mathcal{F}_{\text{BD-STL}}$ might look overly simplified. That is meant as a feature. We only formalize the features relevant for our sharding protocol and abstract away other details. This simplifies the presentation and allows $\mathcal{F}_{\text{BD-STL}}$ to be instantiated using many different consensus protocols, as discussed in Section 6.1 and Appendix B.

Our formalization is also closer to a property-based definition than a typical UC definition. We believe the functionality and its guarantees are easier to understand like this. Enforcing the restrictions guaranteeing persistence and liveness is done by dropping Add commands violating any restriction, and by executing extra Add commands when required to meet any restriction. We omit a description of how exactly these enforcements are handled by the functionality since it is irrelevant for our protocol.

Also note that whenever we write that the functionality outputs something to the adversary, this is meant to leak to the adversary that the corresponding action has occurred. It it not supposed to hand over the activation token to the adversary. Technically, this can be understood as leaving the message in the functionality for the adversary and next time the adversary is activated, it can query the functionality to fetch the message. We again omit the details from the definition to simplify the presentation and focus on our main ideas rather than technicalities.

3.2 Shards

In essence, a shard is just a ledger. For our sharding protocol, however, we need to be explicit about the fact that we want to instantiate shards of different sizes and that a shard may not be live if it is instantiated with a committee that has a too high corruption ratio. Our sharding protocol therefore does not use $\mathcal{F}_{\text{BD-TL}}$ for the shards, but the functionality $\mathcal{F}_{\text{SHARD}}^{s,\mathcal{L},\Delta}$, which we introduce next. The protocol then realizes $\mathcal{F}_{\text{BD-STL}}$ by restarting deadlocked shards internally, cf. Section 5.

The functionality $\mathcal{F}_{\text{SHARD}}^{s,\mathcal{L},\Delta}$ is, in addition to the upper bound Δ on the delay, parameterized by the size s of the committee of parties that will be in charge of maintaining the shard, and the set $\mathcal{L}$ representing the liveness adversary structure. Upon initialization, a committee $\mathcal{C}$ of size s is sampled uniformly at random from all parties, and the resulting list is sent to all parties. We represent the committee as a vector $\mathcal{C} = (P_1, \ldots, P_s)$ of parties, which allows giving special roles to specific parties, e.g., using the first committee member P_1 as a leader. We will sometimes abuse notation and refer to $\mathcal{C}$ as a set. The adversary structure $\mathcal{L}$ means that the shard functionality must maintain its liveness when the list $(i_1, \ldots, i_t)$ of the indices of corrupted parties $P_{i_1}, \ldots, P_{i_t}$ is in $\mathcal{L}$.

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 5.3) requests parties to shut down a shard in order to start a new shard with different parameters and parties. Moreover, we allow *all* parties (including external parties not in $\mathcal{C}$) to verify that $\mathcal{C}$ is indeed the correct committee. Additionally, all parties (including external ones) can request "finality proofs" from the functionality through the GetFinProof command. This proof can then be verified by *any* party. 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. The latter allows to check liveness by ensuring a growing ℓ without needing to know the actual messages.

As the timed ledger, we model the guarantees of the ledger in the shards by letting the adversary control how messages are added to the ledger and imposing some restrictions on the adversary in the form of properties of the shard functionality. 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 (finalized total order) 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.[5] What is special about liveness of shards is that the property only needs to hold if the corrupted parties are in $\mathcal{L}$.

The novel properties that we require for our shard functionality are called *proof soundness* and *censorship resilience*. Proof soundness intuitively says that valid finality proofs can only be produced for correct statements. Censorship resilience prevents an adversary from excluding specific messages from the ledger, while including others. Note that liveness already guarantees that *all* messages will be added to the ledger within time Δ. Censorship resilience is thus a guarantee in case the ledger is not live. 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.[6] In other

[5] 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.

[6] 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.

words, either the message gets included, or the ledger stops completely. This is useful for our sharding protocol, because we need to detect liveness failures and in that case restart the shard. Censorship resilience now ensures that either the ledger is live, or it stops completely, which can be detected from the outside (in contrast to some undetectable censorship).

We formally define the shard functionality next and in Section 4.2 we show a protocol that UC-realizes this functionality.

Functionality $\mathcal{F}_{\text{SHARD}}^{s,\mathcal{L},\Delta}$

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

Input: (InitShard, `sid`) *// Initialize shard with ID* `sid`

1: Output (InitShard, P_i, `sid`) to the adversary.
2: */* Select shard committee of size s */*
3: Upon receiving (InitShard, `sid`) (with the same `sid`) from all honest parties in $\mathcal{P}$, sample a sequence $\mathcal{C}$ uniformly among all sequences of length s from $\mathcal{P}$
4: Set $\mathsf{FTO} = ()$ and $\mathsf{TO}_i := ()$ for all $P_i \in \mathcal{C}$
5: Send (`sid`, $\mathcal{C}$) to all parties in $\mathcal{P}$

Interface for party $P_i \in \mathcal{C}$ *// After InitShard*

Input: (Send, m)

1: Send (Send, P_i, m) to the adversary.

Input: Get

2: Send (Get, P_i) to the adversary.
3: **return** TO_i

Input: Close

4: Send (Close, P_i) to the adversary.

Input: GetFinProof

5: Send (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.
6: Store the record $(\mathsf{TO}_i, \pi, 1)$
7: **return** (TO_i, π)

Interface for adversary

Input: (Add, $\vec{m}, i$)

1: Append $\vec{m}$ to TO_i.

Input: (AddFinal, $\vec{m}$)

2: Append $\vec{m}$ to FTO.

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

Input: (VerifyCommittee, `sid`, $\mathcal{C}$)

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

Input: (VerifyFinProof, $\vec{m}, \pi$)

2: **if** record $(\vec{m}, \pi, b)$ for some $b \in \{0, 1\}$ exists **then**
3: | **return** b
4: **else**
5: | Send (VerifyFinProof, $\vec{m}, \pi$) to adversary, who immediately replies with $b \in \{0, 1\}$.
6: | Store the record $(\vec{m}, \pi, b)$.
7: | **return** b

Input: (VerifyFinLength, ℓ, π) *// Only verify length ℓ of message vector*

8: **if** record $(\vec{m}, \pi, b)$ for some $\vec{m}$ with $|\vec{m}| = \ell$ and $b \in \{0, 1\}$ exists **then**
9: **return** b

10: **else**
11: Send (VerifyFinLength, ℓ, π) to the adversary.
12: Adversary immediately replies with $b \in \{0,1\}$ and $\vec{m}$ with $|\vec{m}| = \ell$.
13: Store the record $(\vec{m}, \pi, b)$.
14: **return** b

At any time, the functionality automatically enforces the following:

Let $A \subseteq \{1, \ldots, s\}$ be the indices of corrupted parties in $\mathcal{C}$. We call the ledger *live* if $A \in \mathcal{L}$. Call the ledger *weakly closed* if some honest party input Close.

Persistence: 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,we have $m \in \mathsf{TO}_i$ for all $P_i \in \mathcal{C} \setminus A$ at most Δ time later.

Censorship Resilience: 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 (VerifyFinProof, $\vec{m}, \pi$) returns 1, then $\vec{m}$ is a prefix of FTO. If (VerifyFinLength, ℓ, π) returns 1, then $|\mathsf{FTO}| \geq \ell$.

4 Committee Selection and Shard Consensus

4.1 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.

Committee selection ideal functionality. We first describe the functionality $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$, 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. This corresponds to the committee selection step in the shard functionality, see Section 3.2. As discussed there, we choose sequences instead of subsets to be able to assign special roles to, e.g., the first committee member. The functionality is formally defined as follows:

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

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

Input: (SelectCom, cid, s) *// Select committee with ID* cid *of size* s
1: Output (SelectCom, P_i, cid, s) to the adversary.
2: Upon receiving (SelectCom, 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: (VerifyCommittee, $\mathsf{cid}, \mathcal{C}$)
1: If $(\mathsf{cid}, \mathcal{C})$ has been sent to all parties in $\mathcal{P}$, return 1, otherwise, return 0.

Selecting parties proportional to their resources. The functionality $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ selects parties from a set $\mathcal{U}$ such that each party is selected as a committee member with equal probability. However, in permissionless blockchain protocols, corruption thresholds are typically

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" [KRDO17, CD17], "weighing by stake" [DGKR18], and "cryptographic sortition" [DPS19, GHM+17, CM19]. In the setting of Proof-of-Work based blockchains, committee selection has also been studied [PS17]. We can thus assume that individual parties are mapped into users in the set $\mathcal{U}$ proportional to the relevant resources they control, and refer the interested readers to the aforementioned results on committee selection on blockchains.

Realizing $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ from a randomness beacon. Given access to a randomness beacon, it is straightforward to realize $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$. Such a randomness beacon gives all parties access to uniformly sampled randomness and allows parties to verify the randomness. A formal definition is provided in [CD20], 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 proposed in [CD20] require a public bulletin board that guarantees that posted messages become immutable and accessible to all honest parties. Note that such a bulletin board can easily be realized by a ledger $\mathcal{F}_{\text{BD-TL}}$. See Section 6.3 for more details on how to instantiate such a randomness beacon.

A straightforward way to realize $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ assuming a randomness beacon works as follows. Given a committee size s, use randomness from the beacon to sample a uniformly random sequence $\mathcal{C}$ from $\mathcal{U}$ with $|\mathcal{C}| = s$. Note that since the beacons also provide the same randomness to all parties, the parties agree on the selected committees. This also directly allows parties to verify selected committees. It is easy to see that this realizes $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$.

Remark on our committee selection. Looking ahead, we always instantiate the committee selection functionality $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ with the entire population $\mathcal{U}$, i.e., every time $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ is executed it returns a random party from the entire population $\mathcal{U}$. Crucially, this means that it is possible that the same party can be part of more than one shard at the same time. This greatly simplifies our analysis and allows us to focus on the techniques to reduce the size of sharding committees. We stress that for practical purposes a more complex committee selection should be used, minimizing the number of shards a single party can be assigned to.

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 [PS17]. In previous works on Proof-of-Stake based blockchain consensus protocols [KRDO17, CD17, DGKR18, DPS19, GHM+17, CM19], 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 [KRDO17, CD17]; and (2) biased committee selection using randomness beacons based on verifiable random functions [DGKR18, DPS19, GHM+17, CM19]. The simple protocol we have described above 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.

4.2 Shard Consensus

In this section, we present a simple protocol using the committee selection functionality from Section 4.1 to implement the shard functionality described in Section 3.2.

Our protocol is parameterized by the committee size s and the maximal number t_{L} of corrupted parties in the committee that can be tolerated without losing liveness. The parameter t_{L} can be any number in $\{0, \ldots, \lfloor \frac{s-1}{2} \rfloor\}$. Based on the safety-liveness dichotomy, the protocol then sets $t_{\mathrm{S}} := s - 2t_{\mathrm{L}} - 1$, where t_{S} is the maximal number of corrupted committee members that can be tolerated without breaking safety. For example, with $s = 100$, one can set $t_{\mathrm{L}} = 33$ to obtain $t_{\mathrm{S}} = 33$, which are the classical bounds for partially synchronous Byzantine agreement. One can also set $t_{\mathrm{L}} = 0$ to obtain $t_{\mathrm{S}} = n - 1$, i.e., if full honesty is required for liveness, all but one party can be corrupted without breaking safety. Consequently, the protocol can only be instantiated securely with committee sizes s that guarantee at most t_{S} corruptions except with negligible probability. See Section 4.3 for how to compute these minimal committee sizes.

The protocol idea is simple. Upon Initialization, the functionality $\mathcal{F}_{\mathsf{CT}}^{\mathcal{P},\mathcal{D}}$ is invoked to obtain a uniformly chosen committee of size s. The first member of the committee is designated a special leader role we call "sequencer". The sequencer periodically proposes a new block containing new messages, which is then signed by the other parties. A block is considered final if at least $s - t_{\mathrm{L}}$ parties have signed it. This is the same basic idea underlying many BFT consensus algorithms [CL99, YMR+19]. One difference is that our protocol also considers values of t_{L} smaller than the maximal $\lfloor \frac{s-1}{2} \rfloor$. Furthermore, typical BFT consensus protocols include (usually involved) mechanisms for replacing a corrupted leader [CL99, YMR+19]. Since our sharding protocol already has an external mechanism (using the control chain) to replace committees, we do not need a leader replacement mechanism in our shard consensus, greatly simplifying it. This means our shard will be live if at most t_{L} parties are corrupted and the leader is honest. That is, our protocol realizes $\mathcal{F}_{\mathrm{SHARD}}^{s,\mathcal{L},\Delta}$ for

$$\mathcal{L} = \{A \subseteq \{1, \ldots, s\} \mid |A| \leq t_{\mathrm{L}} \ \wedge \ 1 \notin A\}.$$

The "special features" of $\mathcal{F}_{\mathrm{SHARD}}$ are straightforward to achieve: Committee verification is provided by the committee selection functionality. Finalization proofs are simply the signatures from the committee members, which can be publicly verified. To guarantee censorship resilience, every party sends a list of "old" messages that have not yet been included in the ledger together with the signature on the proposed block. If the sequencer does not include these messages in the next block, the party refuses to sign that block. Thus, if the sequencer tries to censor a message the honest committee members want to have included, the shard will lose liveness, and in the overall sharding protocol, the committee (including the malicious sequencer) will be replaced.

Formal description of shard consensus. Let $H \colon \{0,1\}^* \to \{0,1\}^\kappa$ be a collision-resistant hash function. For a vector $\vec{m}$, we recursively define $\vec{H}$ as

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

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 $s - t_{\mathrm{L}}$ different parties in $\mathcal{C}$.

The protocol works as follows:

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

Initialization

1: On input (InitShard, `sid`), forward (SelectCom, `sid`, s) to $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$
2: Let $\mathcal{C}$ be the committee returned from $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$
3: Send $(\texttt{sid}, \mathcal{C})$ to all parties $P \in P$
4: Let the sequencer P^* be the first party in $\mathcal{C}$
5: All parties $P_i \in \mathcal{C}$ initialize TO_i to be the empty list.
6: Each party $P_i \in \mathcal{C}$ sets $\mathcal{M}_{i,1} = \varnothing$, $\mathsf{ctr}_{\mathsf{ledger},i} = 1$, and $\mathsf{ctr}_{\mathsf{sign},i} = 1$.
7: The sequencer additionally sets $\mathsf{ctr}_{\mathsf{seq}} = 1$.

Protocol for Sequencer P^*

- 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 $s - t_{\mathrm{L}}$ 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 $s - t_{\mathrm{L}}$ 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 $s - t_{\mathrm{L}}$ 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 (VerifyCommittee, sid, $\mathcal{C}$), forward the request to $\mathcal{F}_{\text{ComSel}}^{\mathcal{P},\mathcal{U}}$ and output the returned bit.
- 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 $s - t_{\text{L}}$ 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 $s - t_{\text{L}}$ 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 prove that the protocol $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$ described above UC-realizes the functionality $\mathcal{F}_{\text{Shard}}$ described in Section 3.2 for appropriate choices of s and t_{L}.

Theorem 1. *Let $n = |P|$ be the total number of parties and let t be an upper bound on the number of corrupted parties in P. Further let $t_S \coloneqq s - 2t_L - 1$ and let $\mathsf{FAIL}_{t_S,s}^{t,n}$ be the event that a uniformly chosen committee of size s contains more than t_S corrupt parties. Assume that $\Pr[\mathsf{FAIL}_{t',s}^{t,n}]$ is negligible (cf. Section 4.3). Then, the protocol $\Pi_{\mathsf{seq}}^{s,t_L}$ UC-realizes the functionality $\mathcal{F}_{Shard}^{s,\mathcal{L},\Delta}$ for $\mathcal{L} = \{A \subseteq \{1,\dots,s\} \mid |A| \leq t_L \ \wedge \ 1 \notin A\}$ and $\Delta \geq 5 \cdot \Delta_{NET}$ 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 $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$ for the honest parties. That is, when $\mathcal{F}_{\text{Shard}}$ 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 $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$. Furthermore, signing of blocks by honest parties is simulated as described in the protocol. Whenever an honest party P_i in simulated protocol $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$ adds a message m to TO_i, the simulator invokes the ideal functionality with (Add, m, P_i). As soon as at least $s - t_{\text{L}} - |A|$ 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 $\mathcal{F}_{\text{Shard}}$ outputs (GetFinProof, P_i) for honest P_i to the simulator, the simulator returns π as computed by P_i in protocol $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$. When $\mathcal{F}_{\text{Shard}}$ outputs (VerifyFinProof, $\vec{m}, \pi$) to the simulator, the simulator verifies the proof as described in $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$. When $\mathcal{F}_{\text{Shard}}$ outputs (VerifyFinLength, ℓ, π) to the simulator, the simulator verifies the proof as described in $\Pi_{\mathsf{seq}}^{s,t_{\text{L}}}$.

As long as the simulator does not violate the constraints of persistence, liveness, censorship resilience, or proof soundness described in $\mathcal{F}_{\text{Shard}}$, 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.

Let A be the set of indices of corrupted parties in $\mathcal{C}$. In the following, we assume $|A| \leq t_{\text{S}}$. Note that by the assumption in the theorem statement, this is the case with overwhelming probability.

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 $s - t_{\text{L}}$ signatures; at least $s - t_{\text{L}} - |A|$ of these signatures must be from honest parties. Since after $s - t_{\text{L}} - |A|$ 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 $s - t_\mathrm{L}$ parties (since it is in TO_i), and the other one by at least $s - t_\mathrm{L} - |A|$ *honest* parties (because it is in FTO). Since $|A| \leq t_\mathrm{S} = s - 2t_\mathrm{L} - 1$, there are at least

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

honest parties who signed the block in FTO. Since $(s - t_\mathrm{L}) + (t_\mathrm{L} + 1) > n$, 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.

Liveness: Assume P^* is honest and at least $s - t_\mathrm{L}$ 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 $s - t_\mathrm{L}$ 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.

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. Since there are at most $t_\mathrm{S} = s - 2t_\mathrm{L} - 1$ corrupted parties and $s - t_\mathrm{L} > t_\mathrm{S}$ signatures are required to finalize a block, no block without m can be finalized anymore. Hence, TO_i must contain m after honest P_i adds the second block. The property follows for $\Delta \geq 2\Delta_\mathrm{NET}$.

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 $s - t_\mathrm{L}$ valid signatures on B. This means that at least $s - t_\mathrm{L} - |A|$ 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 $s - t_\mathrm{L} - |A|$ 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 $s - t_\mathrm{L}$ valid signatures on B. This means that at least $s - t_\mathrm{L} - |A|$ 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 $|\mathsf{FTO}| < \ell$. □

Leader replacement within the shard consensus. We have described a particularly simple protocol without leader replacement since our sharding protocol detects corrupted leaders and consequently resamples the committee. The advantage of this is that the shard consensus is extremely simple and efficient with honest leaders. The downside is that in case of a corrupted leader, the whole committee gets resampled, which may be less efficient than an optimized leader-replacement mechanism of state-of-the-art BFT consensus protocols such as HotStuff [YMR+19].

Alternatively, one can also use a BFT consensus with builtin leader replacement in the shards. In that case, one can realize $\mathcal{F}_{\text{SHARD}}^{s,\mathcal{L},\Delta}$ for

$$\mathcal{L} = \{A \subseteq \{1, \ldots, s\} \mid |A| \leq t_{\text{L}}\}.$$

That is, the additional restriction of an honest leader is not needed and fewer committee resamplings in the shard protocol may suffice to reach liveness.

These protocols are typically only specified for the special case $t_{\text{L}} = t_{\text{S}} = \lfloor \frac{s-1}{3} \rfloor$. It is straightforward to generalize, e.g., HotStuff [YMR+19] to also work with smaller t_{L} and $t_{\text{S}} = s - 2t_{\text{L}} - 1$ by only accepting blocks that have been signed by at least $s - t_{\text{L}}$ committee members. The "special features" of $\mathcal{F}_{\text{SHARD}}$ can be achieved similarly as in our protocol described above and liveness and safety follow can be proven analogously.

4.3 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:

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

Given a maximal admissible corruption ratio $\frac{t'}{s}$ of the sampled committee $\mathcal{C}$ of size s one can find the smallest size s_{min} for which $\Pr[\mathsf{FAIL}_{t',s}^{t,n}] \leq 2^{-\kappa}$, for some security parameter κ. In Appendix C, we derive an analytical bound on the required committee size. In Appendix D, we provide Python code that computes the minimal committee size precisely.

Figure 1 shows the relation between the minimum committee sizes and the level of required guaranteed honesty in the committees for different settings. As the graphs show, the required committee size grows exponentially with the required honesty level. Hence, requiring a smaller guaranteed honesty level than the usual 1/3 significantly improves performance.

Recall that our shard consensus from Section 4.2 can be instantiated with different liveness and safety thresholds L and S, respectively, as long as $S + 2L < 100\%$. Since we want our shards to always be safe, we need to sample committees such that the corruption ratio in the committee is at most S with probability at least $1 - 2^{-\kappa}$ for security parameter κ. Table 1 shows minimal committee sizes for different safety thresholds in different settings.

Note that if the actual corruption ratio in the total population is close to the liveness threshold, there is a good probability for the shard being live. That means if we assume an overall corruption ratio of at most 30%, we never need to use liveness thresholds above 30%, corresponding to safety thresholds of 39%. This already gives a significant performance improvement over prior work, which needed a threshold of 1/3. In the optimistic case, where the actual corruption ratio is below the worst case assumption of 30%, we can run with even smaller committees.

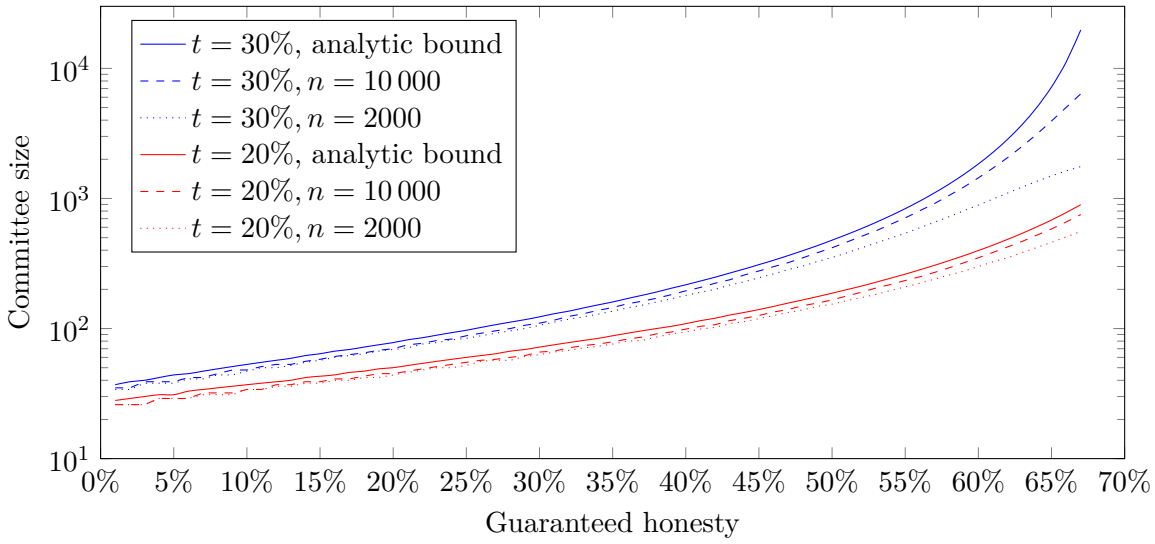

Figure 1: Required committee sizes (on a log scale) to guarantee different honesty levels in the committee with probability $1 - 2^{-60}$, assuming a total population of $n \in \{2000, 10\,000\}$ parties with $t \in \{20\%, 30\%\}$ corruption ratio.

Table 1: Minimum committee sizes for different liveness and safety thresholds such that $S + 2L < 100\%$ with total populations $n = 2000$ and $n = 10\,000$ with 20% and 30% corruption. Minima are computed to guarantee safety except with probability 2^{-60}. Values for $n = \infty$ are the analytical bounds from Lemma 1.

Liveness threshold:		0%	5%	10%	15%	20%	25%	30%	33.3333%
Safety threshold:		99%	89%	79%	69%	59%	49%	39%	33.3333%
n	t	**Minimal committee size**							
∞	30%	37	55	82	130	232	528	2264	16 037
10 000	30%	35	51	75	116	207	462	1713	5886
2000	30%	34	50	71	112	190	382	990	1716
∞	20%	28	38	52	75	115	199	438	854
10 000	20%	26	34	47	67	104	178	385	717
2000	20%	26	34	46	66	99	164	326	540
		our work							state of the art

5 Constructing a Sharded Ledger

5.1 Overview

To realize $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$, 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 $\mathcal{F}_{\text{BD-TL}}^{\Delta}$. For the sake of simplicity, we model $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ as a local functionality. However, our proof does not crucially rely on this modeling choice as it is not necessary for the simulator to program $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ or interfere in its behavior in any way. Moreover, we use a repository functionality $\mathcal{F}_{\text{REPO}}$ to store finalized parts of a shard and previous shards' states, which parties can later obtain these data when reading from a shard or joining a shard committee. This use of $\mathcal{F}_{\text{REPO}}$ captures the fact that it is necessary for at least one party to always store the state of each shard. See Section 5.2 for more details on $\mathcal{F}_{\text{REPO}}$.

In order to execute shards, we use a "gearbox" of shard functionalities $\mathcal{F}_{\text{SHARD}}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ with increasing committee sizes $s_1 \leq s_2 \leq \cdots \leq s_\ell$ to handle shard consensus. In principle, each shard could use its own gearbox, with a different progression of shard functionalities for each shard (e.g., mixing different shard consensus protocols). Moreover, each shard could operate with a different liveness structure $\mathcal{L}$ and a different delay Δ. However, for the sake of presentation nothing is lost in using a gearbox with the same progression of shard functionalities with fixed liveness structures and delay parameters for all shards. There is a statistical security parameter κ, and a liveness guarantee $\gamma > 0$, e.g., $\gamma = \frac{1}{2}$. The gearbox has the following properties.

Always safe: For any committee size s_i, $\mathcal{F}_{\text{SHARD}}^{s_i,\mathcal{L},\Delta}$ is safe except with probability $2^{-\kappa}$.

Eventually live: $\mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ is live with probability at least γ.

We achieve safety and liveness at the same time by first running $\mathcal{F}_{\text{SHARD}}^{s_1,\mathcal{L},\Delta}$ for a random committee. If it loses liveness we switch gears to $\mathcal{F}_{\text{SHARD}}^{s_2,\mathcal{L},\Delta}$ and so on. We start with $\mathcal{F}_{\text{SHARD}}^{s_1,\mathcal{L},\Delta}$ with very poor liveness but a very small committee and consequently high efficiency. On the other hand, $\mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ has strong liveness at the cost of efficiency. The other $\mathcal{F}_{\text{SHARD}}^{s_i,\mathcal{L},\Delta}$ act as intermediary points on the liveness versus efficiency scale, so some of them might be realized by the same consensus mechanism with the same parameters. For example, it may make sense to sample committees of the same size several times to try to hit one with high honesty and hence liveness. However, as long as we have the liveness and safety properties described above, we can adopt the strategy of switching to the next functionality in our gear box every time we lose liveness. When we hit $\mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$, we simply switch to a new instance of $\mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ with a fresh committee, which guarantees that we will eventually get liveness since $\mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ is live with probability at least γ.

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 4.1. 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.

The numbers in Table 1 can thus be used to determine which committee sizes are required to achieve liveness with good probability for a given actual corruption ratio. For example, in the worst case with 30% overall corruption, liveness can be expected for liveness threshold 30%, corresponding to 1713 committee members. With 20% actual corruption, we only need committees of size 207.

5.2 Data Repository

A sharded blockchain ledger achieves its efficiency by avoiding requiring that all parties store the full state of the ledger, only requiring each party to store the state of shard(s) it is responsible for executing. However, parties must still be able to retrieve contents of shards they are not executing, which is necessary for both reading the ledger and operating it (i.e., a party newly assigned to a shard committee must be able to retrieve that shard's past state). In practice, this can be achieved by contacting "full nodes" who store the full state of the ledger, or by using a dedicated data storage provider. We model this mechanism via an abstract notion of a public data repository called $\mathcal{F}_{\text{REPO}}$. Any party can store an entry D under its hash $H(D)$ 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. Moreover, an adversary might delay the storage and retrieval of entries. However, this is inconsequential to the main ideas that we want to present, so we do not model these artifacts into $\mathcal{F}_{\text{REPO}}$ for the sake of a clear presentation.

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

Initialization

The list L of stored entries D is empty.

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

Input: (SEND, D)
Append $(H(D), D)$ to L, where $H()$ is a hash function.
Output $H(D)$.

Input: (GET, h).
if $\exists D\ (h, D) \in L$ **then**
    Output D for the first such entry.
else
    Output $\perp$.

5.3 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 the next shard functionalities in the gear boxes take over the operation of shards that lose liveness. These actions can be divided as follows: (1) Shard Management, which are actions performed by all parties in $\mathcal{P}$ in order to maintain all shards live; (2) Shard Operation, which are actions performed by the parties in the committee responsible for operating a given shard. Members of a shard committee only execute shard management commands related to their shard when these commands have been issued by a majority of parties in $\mathcal{P}$ and appear in a finalized state of the control chain. Moreover, when

receiving inputs, the parties execute instructions that realize the interfaces in $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$. Notice that $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ operates with a fixed number of shards and that start and stop operations are only performed in order to restart shards that have lost liveness but not to add new shards or remove existing shards. While this simplification is done for the sake of a clear exposition, we remark that it is straightforward to generalize the protocol to allow for adding and removing shards arbitrarily. Protocol $\Pi_{\text{BD-STL}}$ works as follows.

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

Parties in $\mathcal{P}$ interact with each other and with functionalities $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, $\mathcal{F}_{\text{Repo}}$ and the gearbox of functionalities $\mathcal{F}_{\text{Shard}}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{\text{Shard}}^{s_\ell,\mathcal{L},\Delta}$. All parties in $\mathcal{P}$ execute the Shard Management steps continuously and execute Shard Operation steps for a given shard when elected as a shard committee member. Each shard is identified by $\mathtt{sid}$ and has a gear parameter h, indicating the current committee size $\mathcal{C}_h$ and functionality $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ responsible for that shard, start time t, indicating when the shard execution with $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ started in terms of $\mathcal{F}_{\text{BD-TL}}^{\Delta}$'s ledger time, and finalization timeout t_{Timeout}.

Shard Management. When execution starts, parties $P_i \in \mathcal{P}$ execute the **Init** steps and then continuously perform **FinalizeCheck**.

Init: Execute Start steps for all shards with parameters $h = 1, t = 1, t_{\text{Timeout}} = \Delta_{init}$ for all $\mathtt{sid}$, where $\Delta_{init} \in \mathbb{Z}, \Delta_{init} > 0$.

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

1: For every message $((\text{Finalize}, \mathtt{sid}, H, \pi, L), t) \in \mathsf{TO}_i$ check that that $t < c_{Timeout}^{\mathtt{sid}}$ and $L > L_{last}^{\mathtt{sid}}$, send $(\text{VerifyFinLength}, L, \pi)$ to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$, obtaining b, and check $b = 1$.
2: Let $((\text{Finalize}, \mathtt{sid}, H, \pi, L_{max}), t) \in \mathsf{TO}_i$ be the message with the maximum L for which the checks of Step 1 succeeded. Set $c_{Timeout}^{\mathtt{sid}} = t + t_{\text{Timeout}}$ and set $L_{last} = L_{max}$.
3: If the checks did not succeed for any message $((\text{Finalize}, \mathtt{sid}, H, \pi, L), t) \in \mathsf{TO}_i$ (i.e., the shard has timed out), execute the Stop procedure for shard $\mathtt{sid}$ and, after the stop command from a majority of parties in $\mathcal{P}$ appears in a future TO_j finalized by $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, execute the Start procedure for shard $\mathtt{sid}$ 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 $h > \ell$, set $h = \ell$ and use a new instance of $\mathcal{F}_{\text{Shard}}^{s_\ell,\mathcal{L},\Delta}$.

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

1: Send $(\text{Send}, (\text{Start}, \mathtt{sid}, h, t, t_{\text{Timeout}}))$ to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, i.e., a command to start shard $\mathtt{sid}$ with $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ at ledger time t w.r.t. $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ with finalization timeout t_{Timeout}.
2: Send $(\text{InitShard}, \mathtt{sid})$ to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$.
3: Set a finalization timeout counter to $c_{Timeout}^{\mathtt{sid}} = t + t_{\text{Timeout}}$.

Stop: All parties $P_i \in \mathcal{P}$ send $(\text{Send}, (\text{Stop}, \mathtt{sid}))$ to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, instructing parties to stop executing shard $\mathtt{sid}$.

Shard Operation. For every shard identified by $\mathtt{sid}$, parties $P_i \in \mathcal{P}$ continuously send (Get) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i. A command for starting or stopping a shard is only considered valid if it has been posted to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ by a majority of the parties in $\mathcal{P}$. Parties $P_i \in \mathcal{P}$ execute the following Shard Operation instructions according to the messages in TO_i and the ledger time:

Start Shard: When there is a command $((\text{Start}, \mathtt{sid}, h, t, t_{\text{Timeout}}), t') \in \mathsf{TO}_i$ posted by a majority of the parties in $\mathcal{P}$, all $P_i \in \mathcal{P}$ wait for $(\mathtt{sid}, \mathcal{C}_{\mathtt{sid}})$ from $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$. If $P_i \in \mathcal{C}_{\mathtt{sid}}$, it sets $c_{Timeout}^{\mathtt{sid}} = t + t_{\text{Timeout}}$ and responds to inputs $(\text{Send}, \mathtt{sid}, m)$ and $(\text{Get}, \mathtt{sid})$.

Finalize: A system parameter Δ_e is estimated in order to ensure that finalization messages are finalized by $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ before the timeout. Every time a shard is restarted, Δ_e is incremented by all parties. At ledger time $c_{Timeout}^{\mathtt{sid}} - \Delta_e$, parties $P_i \in \mathcal{C}_{\mathtt{sid}}$ for shard $\mathtt{sid}$ proceed as follows:

1: Send Get to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ for shard $\mathtt{sid}$, obtaining $\mathsf{TO}_i^{\mathtt{sid}}$.
2: Send GetFinProof to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ for shard $\mathtt{sid}$, obtaining the corresponding finalization proof π.
3: Send $(\text{Send}, \mathsf{TO}_i)$ to $\mathcal{F}_{\text{Repo}}$, receiving $H(\mathsf{TO}_i)$. Notice that if a prefix of TO_i has already been stored in $\mathcal{F}_{\text{Repo}}$, only the new messages in TO_i w.r.t. this prefix need to be sent to $\mathcal{F}_{\text{Repo}}$.
4: Send $(\text{Send}, (\text{Finalize}, \mathtt{sid}, H(\mathsf{TO}_i), \pi, |\mathsf{TO}_i|))$ to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$.

Stop Shard: When parties $P_i \in \mathcal{C}_{\mathtt{sid}}$ see a command $((\textsc{Stop}, \mathtt{sid}), t) \in \mathsf{TO}_i$ from a majority of the parties in $\mathcal{P}$, they send Close to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ and stop executing further shard operation instructions for shard $\mathtt{sid}$.

Interfaces from $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ for Parties $P_i \in \mathcal{P}$.

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

1: Send Get to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i.
2: Find the latest message $((\textsc{Start}, \mathtt{sid}, h, t, t_{\textsc{Timeout}}), t_1) \in \mathsf{TO}_i$ posted by a majority of parties in $\mathcal{P}$. If there exists a message $((\textsc{Stop}, \mathtt{sid}), t_2)$ for $t_2 > t_1$ posted by a majority of parties in $\mathcal{P}$, repeat this step until a new message $((\textsc{Start}, \mathtt{sid}, \mathsf{cid}, \mathcal{C}_h, t, t_{\textsc{Timeout}}), t_3) \in \mathsf{TO}_i$ for $t_3 > t_2$ posted by a majority of parties in $\mathcal{P}$ appears.
3: If $P_i \in \mathcal{C}_{\mathtt{sid}}$, when receiving m (as input or from another party), P_i sends $(\textsc{Send}, m)$ to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ for shard $\mathtt{sid}$. Otherwise, Send m to all parties in $\mathcal{C}_{\mathtt{sid}}$ (obtained from $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ for shard $\mathtt{sid}$).
4: Continuously send (Get) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i and checking that there is a message $((\textsc{Finalize}, \mathtt{sid}, H, \pi, L), t) \in \mathsf{TO}_i$ such that $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ returns 1 when queried with $(\textsc{VerifyFinProof}, \vec{m}, \pi)$, where $\vec{m}$ is in D obtained by sending $(\textsc{Get}, H)$ to $\mathcal{F}_{\text{Repo}}$. If a message $((\textsc{Stop}, \mathtt{sid}), t)$ appears before these checks succeed, P_i goes to Step 1 and waits for a new message $((\textsc{Start}, \mathtt{sid}, h', t', t'_{\textsc{Timeout}}), t) \in \mathsf{TO}_i$ posted by a majority of parties in $\mathcal{P}$.

On input $(\mathbf{Get}, \mathtt{sid})$**:** If $P_i \in \mathcal{C}_{\mathtt{sid}}$ (obtained from $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ for shard $\mathtt{sid}$), P_i sends (Get) to $\mathcal{F}_{\text{BD-TL}}^{\Delta}$, receiving TO_i. Otherwise, P_i ignores the next steps. P_i determines $\mathsf{TO}_i^{\mathtt{sid}}$ by executing the following instructions starting from the largest value of L_j for each $((\textsc{Finalize}, \mathtt{sid}, H_j, \pi_j, L_j), t_j) \in \mathsf{TO}_i$:

1: Send $(\textsc{VerifyFinLength}, L_j, \pi_J)$ to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ (where h is determined by the last valid message $((\textsc{Start}, \mathtt{sid}, h, t, t_{\textsc{Timeout}}), t') \in \mathsf{TO}_i$), obtaining b.
2: Send $(\textsc{Get}, H_j)$ to $\mathcal{F}_{\text{Repo}}$ to get D_j, and send $(\textsc{VerifyFinProof}, D_j, \pi_j)$ to $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ to get b'.
3: If $b = 0$ or $b' = 0$, ignore the next step and proceed to the next message $((\textsc{Finalize}, \mathtt{sid}, H_{j+1}, \pi_{j+1}, L_{j+1}), t_{j+1}) \in \mathsf{TO}_i$ with $L_{j+1} < L_j$. Otherwise, Let D' be the new messages in D_j that are not contained in D_{j-1}. Append (D', t_j) to $\mathsf{TO}_i^{\mathtt{sid}}$.

Notice that if a previous version of $\mathsf{TO}_i^{\mathtt{sid}}$ 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 $\mathsf{TO}_i^{\mathtt{sid}}$. Finally, P_i outputs $\mathsf{TO}_i^{\mathtt{sid}}$.

Theorem 2. *Protocol $\Pi_{BD\text{-}STL}$ described above UC-realizes functionality $\mathcal{F}_{BD\text{-}STL}^{\Delta',\nu}$ in the $(\mathcal{F}_{BD\text{-}TL}^{\Delta}, \mathcal{F}_{Repo}, \mathcal{F}_{SHARD}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{SHARD}^{s_\ell,\mathcal{L},\Delta})$-hybrid model in the partially synchronous model (i.e., where Δ, Δ' 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 $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ 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 $\mathcal{F}_{\text{BD-TL}}^{\Delta}, \mathcal{F}_{\text{Repo}}, \mathcal{F}_{\text{Shard}}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{\text{Shard}}^{s_\ell,\mathcal{L},\Delta}$ 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. When simulating $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ and the functionality $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ 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 $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ as in a real world execution of $\Pi_{\text{BD-STL}}$.

Upon receiving a message $(P_i, \mathtt{sid}, m)$ from $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ 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 with 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 $\mathcal{F}_{\text{Shard}}^{s_h,\mathcal{L},\Delta}$ corresponding to shard $\mathtt{sid}$), $\mathcal{S}$ sends $(\textsc{Add}, \mathtt{sid}, m, t)$ to $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$, 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 $\mathcal{F}_{\text{BD-TL}}^{\Delta}$. If these messages were not sent by $\mathcal{S}$ on behalf of a simulated honest party, $\mathcal{S}$ adds them to $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$: When m appears in the simulated $\mathcal{F}_{\text{REPO}}$ along with 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$\mathcal{F}_{\text{SHARD}}^{s_h,\mathcal{L},\Delta}$ corresponding to shard $\texttt{sid}$) 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 $(\text{ADD}, \texttt{sid}, m, t)$ 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 messages in $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$. However, messages are only added to $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ 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 $\texttt{sid}$ of $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ 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 $\mathcal{F}_{\text{SHARD}}^{s,\mathcal{L},\Delta}$ in the gearbox $\mathcal{F}_{\text{SHARD}}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$, 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 $\mathcal{F}_{\text{BD-STL}}^{\Delta',\nu}$ from a real execution of $\Pi_{\text{BD-STL}}$ with any adversary $\mathcal{A}$. □

Remark 1. In the description above, a heartbeat is the hash of TO_i. This is due to the abstraction boundary of TO_i, which has no concept of blocks. In practice, if shards are implemented using the protocol from Section 4.2, or, e.g., HotStuff [YMR+19], it is sufficient to hash only the last block, since it implicitly finalizes all previous blocks. It furthermore makes sense to have structured heartbeats that allow parties to produce short proofs about the current state of the shard. This can be used, e.g., for intershard transactions and is further discussed in Section 5.5.

Furthermore, if the shards are implemented using a leader-based consensus, it is natural to let only the leader post heartbeats on the control chain.

5.4 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 $\mathcal{F}_{\text{SHARD}}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ for each shard. Here we describe some of these extensions informally.

5.4.1 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.

5.4.2 Adaptive and Mobile Corruptions

Our analysis so far has assumed static corruptions. An inherent problem with committee-based protocols is that an adaptive adversary can corrupt all parties in a committee to break security. A straightforward way to tolerate slowly adaptive corruptions (as recently formalized as δ-delayed

corruptions in [MNT22]), one can resample committees repeatedly. This can even 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.

5.5 Inter-Shard Transactions and Communication

There are different approaches for inter-shard communication in the literature (for an overview of different solutions, see [ZABZ+21, AKW19, WSNH19]). Describing such a solution in full detail in our model would require us to include a notion of transactions and accounts (or `UTXO` [Nak09]), which we have not included in our formalism for clarity. A formal treatment of inter-shard communication is thus out of the scope of this paper, where we want to focus on how to minimize committee sizes using the safety-liveness dichotomy. Below, we sketch how existing solutions can be adapted to our setting.

Special care needs to be taken in our setting because it can happen that shards are not live and consequently get restarted with a new committee. Thus, one cannot immediately trust transactions appearing on one shard for intershard transactions. One can, however, leverage the heartbeats posted on the control chain, which is always safe and live, to ensure transaction finality.

Adapting Atomix. We give a high-level overview of how Atomix[7], which is the intershard transaction protocol of OmniLedger [KJG+18], can be adapted to our setting. When a user wants to initiate a transaction from shard A to shard B using Atomix, the user's client first creates an "export" transaction on shard A, which effectively locks the funds of the user. Once the transaction is included in shard A, the user's client posts a proof that the transaction was accepted by shard A on shard B, which makes the funds available on shard B.

If such a protocol is naively used with our sharding approach, it may happen that a lock transaction appears on shard A and the funds are unlocked on shard B, but shard A crashes before a heartbeat can finalize the lock transaction. When shard A gets restarted with a fresh committee starting from the last heartbeat, the new committee may include other transactions on shard A first, invalidating the lock transaction and allowing to double spend.

To avoid this issue, the acceptance proof must not only prove that a transaction has happened on shard A, but also that it was finalized on the control chain. Recall that in our sharding protocol, all shards regularly post heartbeats on the control chain, where each heartbeat contains a hash of the latest block in the shard. For intershard transactions, we additionally assume that block headers contain the root of a Merkle tree containing (among other data) all the transactions in a block produced by the shard as well as hashes of the blockheader of the parent block. An acceptance proof now consists of a Merkle proof for the transaction in the block plus additional block-header hashes if the transaction was in a block between two heartbeats. These proofs can be further optimized by arranging the hashes of block headers in a tree.

Note that with this approach, even though all intershard messages are finalized through the control chain, only a single heartbeat is needed to finalize all messages of a given shard since the last heartbeat, independently of how many intershard transactions were initiated on that shard. Hence, the throughput of the shards is not bottlenecked by the control chain.

Shard-committee driven intershard transactions. Atomix is a client-driven approach in which the client directly sends the messages to the receiving shard. This puts an additional

[7]Note that the Atomix protocol allowed for a subtle replay attack, which was fixed in a subsequent work [SBAD20]. This issue is, however, not relevant for the high-level overview we give here.

responsibility on the clients, which may be undesirable in practice. An alternative solution is to let the (leader of the) shard committee who prepares the heartbeats directly post the acceptance proofs of outgoing intershard transactions to all receiving shards. An inter-committee routing protocol such as the one from RapidChain [ZMR18] should then be used to minimize communication overheard. Nevertheless, this may put an undesirable overhead on the committee leaders.

Another possibility is to give rewards to the users including the acceptance proofs to the receiving shards. In that case, the user's clients and the committee leaders can still include those proofs, but in case they fail to do so, there is an incentive for other parties to add the missing proofs. Exploring these different options in detail is beyond the scope of this paper.

6 Instantiations

Our treatment so far has been mostly at an abstract level, where we construct a sharded ledger from several building blocks that can be instantiated with different protocols. This makes our treatment more general and allows for modularity. In this section, we provide some concrete instantiations with data points to evaluate the efficiency of the approach. Note that these are simply examples of how one could instantiate the building blocks, and other options with potentially better performance can also be used.

6.1 Instantiation of Timed Ledger

We suggest to instantiate the timed ledger $\mathcal{F}_{\text{BD-TL}}^{\Delta} = \mathcal{F}_{\text{BD-STL}}^{\Delta,1}$ using HotStuff [YMR+19] (for an alternative instantiation using a Nakamoto-style blockchain, see Appendix B). This directly provides a ledger via a sequence of blocks containing a list of messages. The blocks are proposed by a leader and validated by other parties signing them. HotStuff includes a mechanism to replace the leader in case the leader fails or is corrupted. To ensure a dishonest leader cannot censor messages, leaders should also be rotated regularly. As discussed in [YMR+19], this comes without a significant overhead.

The only feature required by $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ that a basic HotStuff implementation does not provide are the timestamps. These can be added easily assuming (weakly) synchronized clocks: The leader adds the current time as a timestamp to the proposed block, and validators only accept a block if its timestamp is larger than that of the previous block and the timestamp is not in the future. Given that basic HotStuff has bounded latency, it is easy to see that this provides the "bounded timestamps" property.

While all parties in the system must be able to read data from the control chain and post messages there, not all of them are required to act as validators in the consensus. We need that the control chain is always safe and live, i.e., less than 1/3 corruption can be tolerated on the control chain. As one can see from the data in Table 1, sampling a subset of size 16 037 is sufficient in the worst case to guarantee less than 1/3 corruption in the subset. Hence, the control chain consensus never needs to be run with more than 16 037 validators.

6.2 Instantiation of Shard Consensus and GearBox

As discussed in Section 4.2, one can use our simple consensus protocol without leader replacement, or alternatively a protocol such as HotStuff [YMR+19] that includes leader rotation for the shard consensus. If a protocol without leader replacement is used, this will come at the cost of resampling the whole committee in case of a corrupted leader, but it will potentially provide better performance with an honest leader due to the simplicity of the protocol. In our analysis

below, we will use performance data from HotStuff, but count the expected resamplings assuming no leader replacement happens in the protocol, providing a worst-case analysis.

For our sharding protocol, we need a "gearbox" of shard functionalities $\mathcal{F}_{\text{SHARD}}^{s_1,\mathcal{L},\Delta}, \ldots, \mathcal{F}_{\text{SHARD}}^{s_\ell,\mathcal{L},\Delta}$ with increasing committee sizes $s_1 \leq s_2 \leq \cdots \leq s_\ell$ (see Section 5). A decent choice is to have four gears with liveness thresholds 10%, 20%, 25%, and 30%. As one can see in Table 1, this requires committee sizes of 82, 232, 528, and 2264, respectively, in the worst case with up to 30% overall corruption and no upper limit on the total number of parties.

6.3 Instantiation of Randomness Beacon

It is observed in [AKW19] that all currently known sharding protocols require a source of randomness. We can leverage the modular nature of our approach and use an external unbiasable randomness beacon such as DRAND (`https://drand.love`) in a practical implementation. We could also use the (publicly verifiable) randomness intrinsically generated in an external proof-of-stake (PoS) based blockchain such as Algorand [CM19], Ouroboros [KRDO17] and Ouroboros Praos [DGKR18]. These protocols use an internal randomness beacon in order to generate seeds for randomly selecting parties to generate blocks. Notice that these randomness beacon protocols have been proven practical as they are implemented as part of, e.g., the Algorand and Cardano cryptocurrency platforms, which respectively run the Algorand and Ouroboros Praos protocols. In case a PoS blockchain is used as the control chain, these same seeds can be re-used in our sharding protocol to *sample random committees at no extra cost*. Given a seed seed and a random oracle $H : \{0,1\}^\star \to \{1,\ldots,n\}$ where n is the total number of parties, this can be easily done for a shard $\mathtt{sid}$ by computing $i_j \leftarrow H(\mathsf{seed}|j|\mathtt{sid})$ for $j = 1,\ldots,s$, where s is the number of parties in the shard committee, finally assigning each party P_{i_j} to the shard committee executing shard $\mathtt{sid}$.

In order to obtain a self-contained solution departing from a non-PoS based control chain (such as HotStuff), randomness beacons traditionally used by PoS blockchains can be executed on top of our control chain regardless of how it is instantiated, since these protocols only rely on the chain for writing information in a black-box manner. Concretely, one can instantiate a UC-secure unbiasable randomness beacon based on Publicly Verifiable Secret Sharing (PVSS) using the approach of ALBATROSS, which would require a honest majority committee of size k, where each party performs $1.5k^2 - 2k$ modular exponentiations and a total $|\mathbb{G}|(2.5k^2 - 2k)$ bits are written to the chain, where $|\mathbb{G}|$ is the size of the representation of a group where DDH is hard. Using our estimates from Table 1, in case the corruption ratio is $t = 30\%$, we have concretely $k = 528$ and consequently 417120 modular exponentiations per party and about 21.2 megabytes required to generate one unbiasable output. Notice that the initial committee for running such a solution is given in the genesis block, while subsequent committees are selected using the outputs of the beacon itself.

A more efficient solution can be obtained by running a Verifiable Random Function (VRF) based randomness beacon on top of the control chain. A UC-secure instantiation of such a beacon has been shown in [DGKR18]. Using again a honest majority committee of size k, each committee member only needs to compute around $4k$ exponentiations and the total storage on chain is only $4k|\mathbb{G}|$ where $|\mathbb{G}|$ is the size of a group element where CDH is hard. Concretely this corresponds to 2112 modular exponentiations per party and 0.065 megabytes of total communication. Once again, the initial committee for running such a solution is given in the genesis block, while subsequent committees are selected using the outputs of the beacon itself. However, such beacons come with a caveat that they produce bounded bias randomness. Fortunately, the only use we make of this randomness is in selecting committees with a certain upper bound on the corruption ratio. It is observed in [DGKR18] that this is possible if the randomness used for committee

selection is has a bounded bias by selecting slightly larger committees. In the remainder of our analysis we only address the case of committee sizes when selection is done with unbiased randomness for the sake of simplicity and due to the fact that this is an orthogonal problem to our core results. However, an analysis similar to that of [DGKR18] could be done in order to derive concrete committee sizes given that the selection is done using bounded biased seeds, although this is out of the scope of our current work.

6.4 Efficiency Analysis of Overall Protocol

Number of sampled committees to obtain liveness. Using the numbers from Section 6.2, we have a GearBox with 4 gears, having liveness thresholds between 10% and 30%. If the actual corruption in the overall system is below 10%, the first sampled committee will already have a corruption ratio below the threshold with high probability. That is because the Chernoff bound implies that these corruption ratios are very close with high probability. To achieve liveness (if our protocol with a fixed leader is used), we additionally need that the uniform leader is honest, which is the case with probability at least 90%. Hence, the expected number of sampled committees is less than 2 in this case.

In the worst case, where the actual corruption in total population equals the maximal 30%, we most likely have to move to the last gear with 30% liveness threshold. In that gear, the probability to sample a committee with not more than the expected 30% corruption is roughly 1/2. Additionally, an honest leader is selected with probability at least 70%, i.e., the overall probability to sample a live committee is at least 35%. Hence, the expected total number of sampled committees is less than 6 (3 committees in the lower gears plus 3 committees in the highest gear on expectation).

Latency and throughput. We here give some rough estimates on the concrete performance based on experimental data for a prototype implementation from the HotStuff paper [YMR+19]. That paper only provides data for up to 128 nodes. We thus extrapolate that data to get some rough idea on the performance. The numbers here are thus not to be understood as precise data points, but only as rough estimates. The actual performance in a production system with code optimized for the particular setting may deviate significantly.

For the control chain, use the experimental data with 128 byte payload data. Since the control chain only needs to process heartbeat messages consisting of hash values and some metadata, 128 byte are sufficient for control-chain messages. The data available in the paper suggests that in that setting, the latency l in milliseconds of messages with committees of size s can be approximated by the linear function $l = 0.37s + 6$. The throughput t in messages per second is roughly inversely proportional to the latency and can be approximated by the function $t = \frac{2\,400\,000}{l}$. As discussed in Section 6.1, the committee size of the control chain will never have to exceed 16 037. Hence, we can estimate a latency of around 6 seconds and a throughput of about 404 messages per second.

For the shard consensus, we use the data for 1024 bytes of payload data because the shards contain the actual transactions, which may be more complex. As above, we approximate the latency in that setting by the function $l = 0.67s + 20$ and the throughput again by $t = \frac{2\,400\,000}{l}$. Thus, we obtain for the four gears with committee sizes of 82, 232, 528, and 2264, the latency values of 75, 175, 374, and 1537 milliseconds, and the throughput values of 32 000, 13 714, 6417, and 1561 transactions per second, respectively.

Note that transactions can only be considered fully final and intershard transactions can only be imported to the receiving shard once their hashes appear on the control chain. Thus the perceived latency (but not the throughput) is limited by the latency of the control chain.

Nevertheless, the numbers show that our approach of minimizing committee sizes can vastly improve performance, even in the worst case with 30% corruption, were we can still operate with shards of size 2264 instead of 16 037 required by prior work.

Scalability and limits on the number of shards. To analyze the scalability of sharding proposals, Avarikioti et al. [AKW19] consider the overhead in terms of communication, storage, and computation. In our protocol, the overhead is limited by ensuring that parties only need to communicate with parties in the same committee (of the shard or the control chain). This is even true for intershard communication when using the solution based on Atomix sketched in Section 5.5. Since HotStuff has linear communication complexity [YMR+19] and our committee sizes are bounded, our protocol scales with respect to communication complexity. With respect to space and computational overhead, note that nodes only need to store and validate transactions in the shards they are assigned to, including outgoing or incoming intershard transactions of these shards, plus the data on the control chain. Since the control chain only contains Merkle tree hashes of the transactions and is thus independent of the number of transactions (intra- or intershard), our protocol scales up to a large constant with the number of shards.

For unlimited scalability, the number shards should be able to scale with the size of the total population. This is not the case in our protocol since all shards need to post their heartbeats to the control chain, and thus the number of shards is limited by the throughput of the control chain and the frequency of the heartbeats. Assuming 400 messages per second throughput on the control chain as estimated above, this means that if shards post their heartbeats every 10 seconds, one can have up to 4000 shards, and if heartbeats are only posted once a minute, this allows for 24 000 shards. While this does not allow for unlimited scalability, we believe these numbers to be more than sufficient in practice in the foreseeable future.

If truly unlimited scalability is desired, an approach with multiple control chains, e.g., arranged in a tree, is necessary. We leave the exploration thereof as a direction for future work.

A 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 [DS82] 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.

A.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 $S \cup L = \mathcal{C}$.[8] 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.

A.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

[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 $S \cup L \neq \mathcal{C}$ for disjoint sets translates into $s + \ell < n$. Dividing by n we get the simplified dichotomy $s/n + \ell/n < 100\%$ of the introduction.

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 $m_i = m$.

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.

A.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 $L_0 \cup L_1 \cup S = \mathcal{C}$.[9] 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.

B Implementing the Timed Ledger using a Nakamoto-Style Blockchain

As discussed in Section 6.1, our timed ledger functionality $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ can be implemented using a BFT consensus protocol such as HotStuff [YMR+19], or similarly using, e.g., Algorand [CM19], or Tendermint [Kwo14, Buc16]. In this section, we show how one can alternatively use a

[9] 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.

Nakamoto-style blockchain such as Bitcoin [Nak09] or Ouroboros Praos [DGKR18]. We note that in case a Nakamoto-style blockchain is used, combining it with a finality layer, e.g., Casper the Friendly Finality Gadget [BG17] or Afgjort [DYMM+20], is advisable to improve the latency.

Following the analysis and the discussion in for instance [GKL17] and in [DGKR18] it seems straight-forward that $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ can be implemented using the Bitcoin protocol or the Ouroboros Praos [DGKR18] 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 [DGKR18]. 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 $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ 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.

In the following, we go into a bit more detail on how the Bitcoin protocol achieves $\mathcal{F}_{\text{BD-TL}}^{\Delta}$. According to the analysis in [GKL17] 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 $\mathcal{F}_{\text{BD-TL}}^{\Delta}$ 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 [GKL17, DGKR18] 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 [GKL17, DGKR18] 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).

C Tight Analytic Bound for Committee Sizes

We analyze the asymptotic probability that when sampling a random committee of size n from a population with corruption ratio p, we get a committee with corruption ratio $\leq q$ for some $q > p$. We will calculate the probability of the bad event that there are more than qn corruptions, and give a formula for picking n such that this bad event has small probability. For an asymptotically large total population, the probability of picking a corrupted party essentially remains equal after removing a party. Hence, we can consider a binomial distribution with n trials and success probability p, corresponding to the probability of selecting a corrupted party.

The lemma below shows that for constant p and q, and large enough κ, the minimum required committee size n linearly depends on κ.

Lemma 1. *Let $\kappa \in \mathbb{N}$, and let X be a random variable following a binomial distribution with n trials and success probability p for each trial. Further let $q \in (p, 1)$, $\alpha \coloneqq q - p$, and $\beta \coloneqq \frac{e}{2\pi\alpha}\frac{\sqrt{q}(1-p)}{\sqrt{1-q}}$. Then, $\Pr[X > qn] < 2^{-\kappa}$ if*

$$n \geq \max\left\{\beta^2, \kappa / \log_2\left(\left(\frac{q}{p}\right)^q \left(\frac{1-q}{1-p}\right)^{1-q}\right)\right\}.$$

Proof. Let $P_x \coloneqq \Pr[X = x]$. Then,

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

We are interested in upper bounding $\Pr[X > qn] = \sum_{x=qn+1}^{n} P_x$. 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

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

Hence

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

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

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

Using $q = p + \alpha$, this yields

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

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

$$\begin{aligned}\binom{n}{qn} &< \frac{e\sqrt{n} n^n}{\sqrt{2\pi qn}(qn)^{qn}\sqrt{2\pi(n-qn)}(n-qn)^{n-qn}} \\ &= \frac{e\sqrt{n}}{\sqrt{2\pi qn}\sqrt{2\pi(n-qn)}} \frac{n^n}{(qn)^{qn}(n-qn)^{n-qn}} \ .\end{aligned} \tag{2}$$

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

$$\begin{aligned}\frac{n^n}{(qn)^{qn}(n-qn)^{n-qn}} &= \frac{n^n}{q^{qn}n^{qn}n^{n-qn}(1-q)^{n-qn}} \\ &= \frac{1}{q^{qn}(1-q)^{n-qn}} = \left(q^q(1-q)^{1-q}\right)^{-n} \ .\end{aligned} \tag{3}$$

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

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

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

If $n \geq \beta^2$, we thus have

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

Therefore, when additionally

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

we get $\sum_{x=qn+1}^{n} P_x < 2^{-\kappa}$. Putting this together concludes the proof. □

D Python Code for Computing Minimal Committee Sizes

In this section we list the python code used to compute the numbers for committee sizes displayed in Table 1 and Figure 1.

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

# For all functions, we use
# n = total population
# t = number of corruptions in total population
# k = security parameter

# Probability of having < h honest parties in committee
# s = committee size
# h = minimum number of honest parties required in committee
# p_max = maximal value for which pFail returns correct value. Output is 1 if p_fail >
    p_max.
def p_fail(n, t, s, h, p_max) :
  p = 0
  # compute n choose s as exact integer
  denom = scipy.special.comb(n, s, exact=True)
  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 > p_max :
      return 1
  return p

# Minimum committee size with corruption ratio at most cr such that p_fail <= 2^{-k}
def min_csize(n, t, cr, k) :
  p_max = Fraction(1, 2**k)
  for s in range(1, n+1) :
    # we want at least h honest parties
    h = math.ceil((1 - cr) * s)
    if p_fail(n, t, s, h, p_max) <= p_max :
      return s

# Compute analytical upper bound
# p = fraction of corruption in total population
# cr = maximal corruption ratio in committee
def analytic_bound(p, cr, k) :
```

```
  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))
  beta_bound = math.ceil(beta*beta)
  return max(bound, beta_bound)

# Print committee sizes for different parameters
# ns = list of values of n, total population
# ps = list of corruption fractions in total population
# k = security parameter
# crs = list of maximal corruption ratios in committee
def print_csizes(ns, ps, k, crs) :
  header = ["", ""] + [float(cr) for cr in crs]
  table = []
  for p in ps:
    table += [ ["analytic", "p = " + str(p)] + [analytic_bound(p, cr, k) for cr in crs]]
    for n in ns:
      t = n * p
      table += [ ["n = " + str(n), "p = " + str(p)] + [min_csize(n, t, cr, k) for cr in
          crs]]

  print(tabulate(table, header))

# Print coordinates of committee sizes over guaranteed honesty
def print_graph_coordinates(n, t, k) :
  for crp in range(99, 32, -1) : # corruption from 99% down to 33%
    cr = Fraction(crp, 100) # convert percentage to fraction
    print("(",int(100 - 100*cr), ",", min_csize(n, t, cr, k),")",sep='',end=' ')
  print("\n")

# Print analytic bound for coordinates of committee sizes over guaranteed honesty
# p = fraction of corruption in total population
def print_analytic_graph_coordinates(p, k) :
  for crp in range(99, 32, -1) : # corruption from 99% down to 33%
    cr = Fraction(crp, 100) # convert percentage to fraction
    print("(",int(100 - 100*cr), ",", analytic_bound(p, cr, k),")",sep='',end=' ')
  print("\n")


def main():
  # Consider total populations 10000 and 2000 with 30% and 20% corruption, with 60 bit
      security
  ns = [10000, 2000]
  ps = [Fraction(3, 10), Fraction(2, 10)]
  k = 60

  # maximal corruption ratios in committees from 99% to 33%
  crs = [Fraction(99, 100), Fraction(89, 100), Fraction(79, 100), Fraction(69, 100),
      Fraction(59, 100), Fraction(49, 100), Fraction(39, 100), Fraction(1, 3)]

  # print table with committee sizes
  print_csizes(ns, ps, k, crs)
  print("\n")

  # print graph coordinates
  for p in ps:
    print("Analytic graph coordinates, p = ", p, ", k = ", k, sep='')
    print_analytic_graph_coordinates(p, k)
    for n in ns:
      print("Graph coordinates, n = ", n, ", p = ", p, ", k = ", k, sep='')
      t = n * p
      print_graph_coordinates(n, t, k)

if __name__ == '__main__':
  main()
```

References

[AKW19] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. Divide and scale: Formalization of distributed ledger sharding protocols. *CoRR*, abs/1910.10434, 2019.

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[BH04] 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.

[BKT+19] Vivek Kumar Bagaria, Sreeram Kannan, David Tse, Giulia C. Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 585–602. ACM, 2019.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Katz and Shacham [KS17], pages 324–356.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, The University of Guelph, Guelph, Ontario, Canada, 6 2016.

[Can04] 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.

[CD17] 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.

[CD20] Ignacio Cascudo and Bernardo David. ALBATROSS: Publicly AttestabLe BATched Randomness based On Secret Sharing. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 311–341. Springer, Heidelberg, December 2020.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[CM19] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

[DGKR18] 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.

[DPS19] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 23–41. Springer, Heidelberg, February 2019.

[DS82] 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.

[DYMM+20] 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, *SCN 20*, volume 12238 of *LNCS*, pages 24–44. Springer, Heidelberg, September 2020.

[GHM+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.

[GKL15] 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.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Katz and Shacham [KS17], pages 291–323.

[KJG+18] 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.

[KMTZ13] 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.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Katz and Shacham [KS17], pages 357–388.

[KS17] Jonathan Katz and Hovav Shacham, editors. *CRYPTO 2017, Part I*, volume 10401 of *LNCS*. Springer, Heidelberg, August 2017.

[Kwo14] Jae Kwon. Tendermint: Consensus without mining. manuscript, 2014. https://tendermint.com/static/docs/tendermint.pdf.

[LNZ+16] 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.

[MNT22] Christian Matt, Jesper Buus Nielsen, and Søren Eller Thomsen. Formalizing delayed adaptive corruptions and the security of flooding networks. In *Advances in Cryptology – CRYPTO 2022*, Cham, 2022. Springer International Publishing. To appear.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. manuscript, 2009. http://www.bitcoin.org/bitcoin.pdf.

[PS17] 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.

[RKTV22] Ranvir Rana, Sreeram Kannan, David Tse, and Pramod Viswanath. Free2shard: Adversary-resistant distributed resource allocation for blockchains. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1):11:1–11:38, 2022.

[SBAD20] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, pages 294–308. IEEE, 2020.

[SC21] Abdurrashid Ibrahim Sanka and Ray C.C. Cheung. A systematic review of blockchain scalability: Issues, solutions, analysis and future research. *Journal of Network and Computer Applications*, 195:103232, 2021.

[WSNH19] 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.

[WW19] 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.

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In Peter Robinson and Faith Ellen, editors, *38th ACM PODC*, pages 347–356. ACM, July / August 2019.

[YNHS20] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. OHIE: blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 90–105. IEEE, 2020.

[ZABZ+21] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. Sok: Communication across distributed ledgers. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 3–36, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[ZMR18] 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.