State Machine Replication Among Strangers, Fast and Self-Sufficient

Juan Garay[1], Aggelos Kiayias[2] and Yu Shen[3]

[1]Texas A&M University, garay@cse.tamu.edu
[2]University of Edinburgh and IOG, aggelos.kiayias@ed.ac.uk
[3]University of Edinburgh, shenyu.tcv@gmail.com

Abstract

A set of unacquainted parties, some of which may misbehave, communicate with each other over an unauthenticated and unreliable gossip network. They wish to jointly replicate a state machine Π so that each one of them has fair access to its operation. Specifically, assuming parties' computational power is measured as queries to an oracle machine $H(\cdot)$, parties can issue symbols to the state machine in proportion to their queries to $H(\cdot)$ at a given fixed rate. Moreover, if such access to the state machine is provided continuously in expected constant time installments we qualify it as *fast fairness.*

A *state machine replication* (SMR) protocol in this *permissionless* setting is expected to offer consistency across parties and reliably process all symbols that honest parties wish to add to it in a timely manner despite continuously fluctuating participation and in the presence of an adversary who commands less than half of the total queries to $H(\cdot)$ per unit of time.

A number of protocols strive to offer the above guarantee together with fast settlement — notably, the Bitcoin blockchain offers a protocol that settles against Byzantine adversaries in polylogarithmic rounds, while fairness only holds in a fail-stop adversarial model (due to the fact that Byzantine behavior can bias access to the state machine in the adversary's favor). In this work, we put forth the first Byzantine-resilient protocol solving SMR in this setting with both expected-constant-time settlement and fast fairness. Furthermore, our protocol is **self-sufficient** in the sense of performing its own time keeping while tolerating an adaptively fluctuating set of parties.

An abridged version of this paper appears in *Proc. CRYPTO 2025.*

Contents

1 Introduction

In the *state machine replication* (SMR) problem [Sch90], a state machine $\Pi = (Q, \Sigma, \delta)$, with $\delta(q, \sigma) \to q'$ its transition function, is to be replicated across a number of parties. At any given time, the machine Π is at a certain state that results from the application of a sequence of symbols $\sigma_1 \dots \sigma_n$ to its transition function. In the most common interpretation of the problem, it is mandated that each party P maintains a log, denoted by $\mathsf{log}_{\mathsf{P}}$, of such symbols such that the following two fundamental properties are satisfied:

- **Consistency:** At any two times t, t', for any two parties P, P', it holds that $\mathsf{log}_{\mathsf{P}} \not\preceq \mathsf{log}_{\mathsf{P}'}$ implies $\mathsf{log}_{\mathsf{P}'} \preceq \mathsf{log}_{\mathsf{P}}$, where $\preceq$ stands for the prefix relation.
- **Liveness:** If all honest parties attempt to introduce a symbol σ into the log, then after u steps in nominal time it holds that every honest party's log will include σ.

We note that the liveness property, as expressed above, refers to a notion of *nominal* time, which will not be available to the parties — indeed, to achieve it, the parties themselves may have to synchronize with each other in order to approximate nominal time using their local clocks which might be drifting: In particular, the setting of interest here is being equipped with local clocks that run at different speeds but where there is an upper bound ρ on how fast they can run compared to nominal time.

To properly reflect this timing requirement into our objectives, we will focus on state machines that incorporate time into their state. Specifically, such a state machine has the following structure $\Pi = (Q \times \mathbb{N}, \Sigma, \delta)$ and includes a special "clock tick" symbol σ_{ct} that satisfies $\delta((q, t), \sigma_{\mathsf{ct}}) = (q, t+1)$, for any q, t. Let $\mathtt{r}$ be the resulting SMR clock value and, with foresight, denote by $\varGamma$ its accuracy (cf. Section 3.1). The relevant property regarding time keeping is as follows:

- **$\varGamma$-Timekeeping:** Let $\varGamma \in \mathbb{R}^+$. At nominal time t, the state machine internal clock $\mathtt{r}$ satisfies $(1+\varGamma)^{-1} \leq \mathtt{r}/t \leq (1+\varGamma)$. We refer to $\mathtt{r}$ as the "SMR time."

Sometimes we will refer to an SMR that keeps its own time under the mere assumption that parties have drifting local clocks as "self-sufficient."

Contrary to the "classical" SMR setting where parties have point-to-point channels and/or are capable of authenticating digital signatures issued by each other, we are interested in a setting where parties are **unacquainted**, in the sense that they have no information about each other, and communicate via an unauthenticated "diffusion channel" that allows the adversary to inject any arbitrary number of messages. Moreover, parties may come and go without any announcement to others whatsoever, with their overall number fluctuating over time; hence, the common protocol design technique of counting the number of incoming messages may not be particularly helpful. This is what sometimes is referred to as the "permissionless" setting.

The permissionless setting suggests a rather insurmountable target for SMR as it can be easily seen by the following argument: A set A of n parties is about to engage in the protocol, while another set A' of equal number of parties are about to do the same — parties in A are unaware of the existence of A' and vice versa. Consider now two symbols $\sigma \neq \sigma'$ so that parties in A attempt to insert σ while parties in A' attempt to insert σ'. The adversary can expedite message delivery within the sets A, A' and hence eventually A, (resp., A') will have to settle σ (resp., σ'), due to Liveness. It follows that, if we are to solve SMR in this setting, we have to assume that communication delays are always bounded, say, by a quantity $\varDelta$. Even with bounded delays though, an adaptation of the classical FLP impossibility result [FLP85] can be used to show that SMR is still impossible (cf. [PS17b]). The challenge stems from the ability of the adversary to simulate protocol messages "in its head" without any cost and the fact that lack of authentication enables the adversary to serve such messages as if they were part of a legitimate protocol execution. To circumvent this impossibility, we will assume the existence of an (random) oracle $H(\cdot)$ and impose a restriction on how many queries to $H(\cdot)$ the adversary can pose. This small asymmetry between honest and adversarial parties turns out to be sufficient to circumvent this impossibility.

The Liveness property introduced above is preconditioned on all honest parties attempting to insert the same symbol in the log. In the permissioned setting, where all parties are acquainted with each other, it can be straightforward to create circumstances where honest parties attempt to insert the same symbol (e.g., all parties may propose symbols in a round-robin fashion). In the permissionless setting, however, this is problematic: Without any form of authentication or throttling, it is possible for an adversary to flood the incoming tapes with symbols it prefers while stifling useful inputs. To address this consideration we will require an additional *fairness* property for permissionless SMRs: Any honest party gets a chance to introduce a special symbol of their choice[1] with probability in proportion to the number of queries posed by the party to the oracle $H(\cdot)$. More formally:

– **Fairness:** Let $\mathtt{r}$ denote the SMR time when a special symbol σ^* is introduced to the log of all honest parties. Then, for the window of nominal time $W_{\mathtt{r}}$ ending in $\mathtt{r}$ of length $B \in \mathbb{N}$, a property parameter, we say that party P has weight w_{P} that is equal to the number of queries posed by P to $H(\cdot)$ during $W_{\mathtt{r}}$. Fairness requires that special symbol σ^* is issued from an honest party P with probability q such that $q \geq (1-\epsilon)(w_{\mathsf{P}}/\sum_{\mathsf{P}'} w_{\mathsf{P}'})$, for any $\epsilon \in (0,1)$.

A parameter of particular interest of the above property is the length of the time window $W_{\mathtt{r}}$; i.e., how fast does the protocol sample the distribution of queries to $H(\cdot)$ to allow a new symbol in the log. In particular, if the length of this window is of constant size, we will refer to the property as *Fast Fairness.*

1.1 Summary of Our Results

In this paper we put forth a protocol for SMR in the permissionless setting that, for the first time, satisfies all the properties enumerated above, while ensuring in addition that Liveness is achieved in expected-constant time (a condition usually referred to as "fast [transaction] settlement"[2]). See Table 1 for a comparison with existing results.

As the table shows, our permissionless SMR protocol is the first to be "self-sufficient" in terms of being its own time keeper, and in the more realistic model where parties' clocks may drift. In contrast, all existing protocols either have assumed the availability of a global clock; or that such functionality is realized by means of some heuristic, as in the case of Bitcoin [GKL20]; or that parties' clocks might be off but proceed at the same speed [GKS22].

We achieve this by means of a **new clock synchronization** protocol for the permissionless setting that achieves a constant (i.e., asymptotically optimal) "skew" tolerating an adversary that controls less than 50% of the computational power (a bound we prove to be optimal in our setting), and may be of independent interest. Specifically, the skew of the clocks is $\Theta(\rho\Delta)$, where ρ is the upper bound on clock drift rates and Δ is the maximum network latency. Importantly, and in contrast to previous treatments of network delay in the blockchain literature (e.g., [PSs17, BMTZ17, GKL20, CEM+20]), where the delay is specified in terms of network 'rounds,' here it is specified in terms of nominal time, as in the original distributed-computing formulation [DLS88], as in our setting the notion of 'round' is a local/per-party one, and, further, may not be of uniform duration

[1]Think of a 'coinbase' transaction in Bitcoin.

[2]We will use 'symbols' and 'transactions' interchangeably, as it is common in the blockchain literature.

Protocol	**Settlement**	**Dynamic**	**Timekeeping**	**Drifting Clocks**
[GKL15]	$\mathcal{O}(\mathsf{polylog}\kappa)$	✗	✗	✗
[GKL17]	$\mathcal{O}(\mathsf{polylog}\kappa)$	✓	✗	✗
[BKT$^+$19]	Expected $\mathcal{O}(1)^*$	✗	✗	✗
[FGKR20]	Expected $\mathcal{O}(1)^*$	✓	✗	✗
[GKS24]	Expected $\mathcal{O}(1)$	✗	✗	✗
[GKS22]	$\mathcal{O}(\mathsf{polylog}\kappa)$	✓	✓	✗
This work	Expected $\mathcal{O}(1)$	✓	✓	✓

Table 1: *Comparison of permissionless SMR protocols in terms of their settlement time, ability to handle dynamic participation and timekeeping against drifting clocks.* **: Running time here refers to the optimistic case when no transactions are "double-spent."*

(in nominal time). (Refer to Section 3.1 for background on fault-tolerant clock synchronization basics.)

Furthermore, the protocol allows for a fully transient participation pattern—i.e., even if every protocol participant fully functions in only one round and immediately goes offline, our protocol still remains secure—and is based on the following key components: (i) A novel blockchain approach that utilizes *parallel blockchains*; (ii) a new clock adjustment algorithm that runs *approximate agreement* on top of parallel blockchains and enables parties to concentrate a clock shift value so they can adjust their local clocks maintaining a bounded overall skew and near optimal accuracy (up to an arbitrary small constant) with respect to nominal time; and (iii) a new bootstrapping protocol that allows a newly joining party with no knowledge other than the initial setup (i.e., the genesis block) to "catch up" with the synchronized parties. I.e., after bootstrapping, a new party adjusts its local clock to exhibit a deviation from the existing online parties' clocks that is well-bounded (in the order of the network latency).

Our parallel blockchain construction allows, in addition, a state update for every interval of constant duration, thus making it possible to run a PoW-based expected-constant-time consensus protocol (specifically, the "Chain-King Consensus" protocol in [GKS24]) "on top" of our clock synchronization protocol, to yield a distributed ledger whose security does not rely on parties having access to a global clock, and where **all** incoming symbols (e.g., transactions) can be added to parties' logs in expected-constant time. Furthermore, our protocol utilizes a suitable pre-agreement phase that facilitates Fast Fairness.

1.2 Related Work

SMR protocols. In the traditional, permissioned setting [Sch90], an SMR protocol is executed by a fixed set of servers that are "acquainted" with each other (e.g., they know each other's public key and/or have explicit point-to-point communication channels with each other). Numerous protocols have been proposed in this setting, with more recent work focusing on efficient constructions (e.g., [CL99, YMR$^+$19, CPS19, AMN$^+$20, DKSS22]), invariably exploiting the ability of participants to issue votes in the form of signatures and have those votes counted by the recipients to ensure that a suitable quorum has been reached and parties can settle the transactions in the log.

It is clear that such techniques are not readily amenable to the permissionless setting.

The only known design technique for achieving SMR in the permissionless setting with dynamic participation is based on the Bitcoin blockchain (cf. [Nak08, GKL17]). In this protocol, the oracle $H(\cdot)$ is utilized to realize a proof-of-work (PoW) functionality [DN93] with a moderate difficulty that is periodically adjusted to accommodate fluctuations in participation. From our perspective, the protocol exhibits a number of deficiencies, namely, Liveness with a parameter proportional to the security parameter, lack of fairness (due to block withholding selfish mining attacks [ES14, GKL15]) and Timekeeping whose accuracy is based on the participants' having access to a shared global clock (so the protocol is not self-sufficient). Follow up work to Bitcoin addressed some of these issues individually; for example in the static participant setting, Fruitchains focused on the issue of fairness [PS17a], and Prism on the issue of transaction throughput [BKT$^+$19]. For further overview as well as impossibility results see [PS17b, GK20]. Nonetheless, to date, no SMR protocol in the permissionless setting has been proposed that addresses all the relevant considerations simultaneously fairness and self-sufficiency.

Clock synchronization. The clock synchronization problem has been studied for over four decades by the distributed computing community. *Synchronizers* are distributed fault-tolerant protocols that solve the synchronization problem — to mention a few, pulse synchronizer [LMS85, LL84b, HSSD84, ST87, LL22] where parties re-synchronize their clocks periodically and one-shot synchronizer [LL84a, HMM85] where the goal is to synchronize clocks with initial large skews. These traditional protocols operate in the permissioned model where the participants are known *a priori* (or, parties can join upon approval from all honest parties, cf. [HSSD84]).

Dolev *et al.* [DHS86] showed that without setup assumptions, clock synchronization cannot be achieved with more than one-third of the corrupted parties (i.e., it requires $t < n/3$). With unforgeable signatures, the corruption bound can be improved to $t < n/2$ [ST87, LL22], or to the dishonest majority setting [HSSD84]. When the protocol allows new parties to join, a majority of the honest parties is necessary.

Bounded skew (i.e., the level of simultaneity) is a fundamental property when measuring the performance of synchronizers. In the fault-free setting, Lynch and Welch [LL84a] showed that even if clocks run at exactly the same rate, network uncertainty is impossible to overcome. Precisely, in a network with $\varDelta$ delay (again, measured in real time) and n processors, it is impossible to synchronize clocks more closely than $\varDelta(1-1/n)$. This result was later extended to any network by [HMM85]. Since this result holds under strong assumptions, they also apply to the drifting clock model. When it comes to pulse synchronizers, the interval between two synchronization points should be at least $\Theta(\varDelta)$ rounds apart from each other, during which the clock has already drifted for more than $\Theta(\rho\varDelta)$ time. Hence, $\Theta(\rho\varDelta)$ turns out to be the (asymptotically) optimal skew one could expect.

With pulse synchronization, another fundamental metric is the degree of deviation from real time, namely *accuracy.* Dolev *et al.* showed that synchronization is a non-trivial task only when the target logical time stays in a linear envelope of real time [DHS86]. Srikanth and Toueg [ST87] showed that the logical linear envelope cannot be smaller than the physical one; and, that to stay in the same envelope with physical clocks, a majority of the participants should be honest.

More recently, the clock synchronization problem has been re-considered in the context of blockchains, however in a **weaker model** of imperfect local clocks where the adversary can (only) apply an **additive** drift $\varPhi_{\mathsf{clock}}$ to the honest parties' clocks throughout the whole execution [BGK$^+$21, GKS22]. In the "classical" setting, this model is not meaningful in that when parties are always online, their skew will never deviate further than $\varPhi_{\mathsf{clock}}$. The above works considered this weaker model in the permissionless environment with dynamic participation where parties can join and leave as they please. As a result, protocol participants can no longer filter messages based on the total number of parties, and newly joining parties should be able to bootstrap and synchronize their clocks with honest parties by passively listening to the network. Since the clock model is weaker, those results in [BGK$^+$21, GKS22] are not directly comparable to the traditional literature with drifting clocks. From our perspective, the key observation is that there are two challenging dimensions to clock synchronization: drifting clocks, analyzed mostly in the permissioned literature, and imperfect clocks exhibiting a bounded skew in a setting where participants join and leave the protocol at will. Handling both these dimensions at once is a critical missing piece for achieving SMR in the permissionless setting.

Timing models in cryptography. In distributed computing parlance, following the treatment from [DLS88], the synchrony hierarchy yields three levels: (i) Synchronous — there are *known* upper bounds on clock drift and maximum network delay. (ii) Partially-synchronous[3] — upper bounds on clock drift and delay do exist yet they are *unknown* to honest parties; note that in [DLS88], delay is measured by a *real time clock* outside the system. (iii) Asynchronous — there is no upper bound on local clock speeds and message transmission (though messages between honest parties are eventually delivered).

Timing shows as a tool in cryptography, nonetheless, there lacks a unified approach on modeling time and delays, especially they are ill-defined within the UC setting. Here we provide a short survey.

Timing models came into consideration for the secure concurrent zero-knowledge protocols [DNS98, DS98], where Dwork, Naor and Sahai proposed the (α, β)-constraint (for some [known] $\alpha \leq \beta$) — for any two parties P and P', if P measures α elapsed time on its local clock and P', starting after P, measures β elapsed time on its local clock, then P' finishes after P— an assumption that is implicit under the (appropriately-bounded) drifting clock model. In concurrent composition of secure computation [KLP05], Kalai, Lindell and Prabhakaran work in a model where local clocks run within known bounded rates and message transmission takes up to a known Δ time. Yet, they define delay based on *all clocks* — i.e., maximum message transmission is subject to the bound on clock drifts (cf. [KLP05, footnote 10]).

In the UC setting, despite its inherent asynchronous message transmission scheduling, Katz *et al.* [KMTZ13] model synchronous computation via the co-design of a synchronized clock and bounded-delay channel functionalities. In more detail, real time (represented as a global round counter) is forwarded only when all honest parties claim finishing their computation in that round, and message is delivered to a party after receiving sufficiently many FETCH requests where honest parties issue one FETCH query per *local* round, and the adversary can accumulatively increment the fetch counter for up to Δ rounds. Recently, Canetti *et al.* [CHMV17] provide a treatment of the network time protocol by means of a global clock functionality with bounded additive drifts and unbounded response delays. Aided with such global clock, they also present another functionality that measures the local clock drifts (i.e., relative time elapsed between two global clock reads).

Parallel blockchains and $m{\times}1$ PoWs. Parallel blockchain designs have been found applications in improving the performance of blockchain-based SMR. For example, in "Ledger Combiners" [FGKR20] a ranking function is proposed on top of a set of parallel chains to accelerate transaction settlement — for the case of non-conflicting transactions. In [GKS24], parallel blockchains serve as a platform to port classical consensus protocols and enable building a PoW-based, expected-constant-time Byzantine agreement protocol, and achieve SMR with expected-constant settlement

[3] [DLS88] also considers a second type of partially synchrony where there is an *unknown* global stable time (GST) such that a *known* maximum network delay holds after GST however no restriction on message transmission is imposed before GST.

time, albeit for the *static* participation case.

The fundamental cryptographic primitive that secures parallel blockchains is $m{\times}1$ *Proof-of-Work*[4], which guarantees that the mining procedure of single chains are mutually independent (or, sub-independent with bounded statistical distance — cf. [FGKR20]); i.e., the adversary cannot gain advantage on a specific chain by dropping from others. The $m{\times}1$ PoW is in fact a generalization of the $2{\times}1$ PoW technique introduced in [GKL15] that achieves an equitable distribution of inputs contributed into a blockchain based on the oracle queries posed by the participants to the oracle $H(\cdot)$. In [GKL15], the $2{\times}1$ PoW primitive was used to improve the corruption resiliency of permissionless Byzantine agreement, and has also been utilized in [PS17a] to design a blockchain protocol in the *static* participation setting that offers a notion of fairness (although not fast). Combining fairness with dynamic participation and clock synchronization (i.e., self-provided timekeeping) has remained until now an open question.

In [GKS24], the $m{\times}1$ PoW scheme that yields SMR with expected-constant settlement time works by running $m = \Theta(\mathsf{polylog}\kappa)$ chains in parallel, where κ is the output length of the hash function, with $\Omega(\mathsf{polylog}\kappa)$ bits allocated to each chain. As mentioned above, this protocol is designed for the static participation setting. Finally, note that in the PoS context, full independence among m parallel chains can be achieved by separately evaluating m VRFs with different nonces, which yields a simple but equivalent alternative construction to $m{\times}1$ PoW.

On transient faults and dynamic participation. To wrap up, our setting is typified by a fluctuating number of participants who may come and go without announcement. As a result, at nominal time t the number of participants is n_t, with the initial number of parties being n_0. Dynamic availability has been considered in prior work [PS17c, BGK+18], but not in the completely unacquainted setting as we do here: These works operated under the assumption that a consistent public-key directory is known to all participants and the adversary may choose an arbitrary subset of registered parties to run the protocol. In the permissionless setting, dynamic availability was considered in [GKL17] under the assumption that the sequence $n_1, n_2, \ldots$ is **not** adaptively determined — an important restriction that is not present in our modeling.

Organization of the paper. The rest of the paper is organized as follows. In the next section we provide a technical overview of our results. In Section 3, we introduce our clock, network and adversary models and provide basic notation and definitions. Then, in Section 4, we solve two "one-shot problems" — permissionless Weak Agreement and permissionless Approximate Agreement — as basic building blocks. In Section 5 we present the full permissionless SMR protocol, which is based on the new parallel blockchain construction (Section 5.1) and permissionless clock synchronization procedure (Section 5.2). The full protocol analysis is presented in Section 6.

2 Technical Overview

Our protocol, which we term Permissionless-SMR, divides time (in parties' local views) into consecutive, non-overlapping "intervals." At the end of each interval, based on their local view of their state parties update their internal ledger state and re-synchronize their clocks.

A new parallel blockchain construction. We propose a novel blockchain construction that utilizes parallel blockchains and preserves the optimal corruption resilience threshold while accommodating dynamic participation. We highlight the key sub-components of our new construction.

First, parallel chains are extended independently and continuously — for each chain, parties always select and work on the "longest" (in fact, "heaviest" in terms of accumulated difficulty)

[4]Pronounced "m-for-1" PoW.

chain; however, all forks are maintained for future potential use, i.e., our parallel blockchains are in fact parallel *blocktrees.* The mining procedures of the parallel blocktrees are bound together using $m\times 1$ PoW, where one random oracle query may yield multiple valid blocks on different chains yet all sub-procedures are mutually independent.

Second, we introduce a new timestamping scheme where each timestamp consists of a pair of interval and round indices. Contrary to previous abstractions of blockchain protocols that ask for monotonically increasing timestamps on a single chain, our new scheme allows parties to insert timestamps where monotonicity is only enforced on the interval index, while round indices are allowed to also decrease. In terms of incoming blocks, parties defer their processing when they report timestamps that either in rounds or intervals are in the future, based on the local clock.

Third, we introduce a state retrieval mechanism whereby parties joining the protocol execution can learn all the forks that used to be the longest chain at the end of any previous interval (and remain in the common view of honest participants that are online at the time the new party joins). This applies to all past chains that are now orphaned. To accommodate this retrieval mechanism, our protocol asks parties to include their local views of the previous interval into the blocks they are mining as input blocks for the next interval. If it happens that on sufficiently many parallel chains, the majority of the input blocks report the same block hash, then all these blocks are retrieved, even if they are on orphaned chains that no party is extending any more. It is worth pointing out that this retrieval mechanism can be done in a "trace-back" manner — i.e., newly joining parties can first observe the longest m chains in the current interval, and then retrieve the blocktrees of the previous intervals one by one from the latest interval going all the way back to the first interval. In this way, the retrieval mechanism provides an oblivious agreement over the common blocktrees at the end of each interval, and we show that this weak type of agreement is sufficient to synchronize parties' internal clocks and states in the bootstrapping procedure for a new client. A detailed specification of our new parallel chain construction is presented in Section 5.1.

Clock synchronization. In each interval, with some initial skew Φ_{init}, the honest parties' clocks will deviate from each other as time elapses; to compensate, parties run an *Approximate Agreement* (AA) protocol [DLP^{+}86] in order to maintain their local clocks within the Φ_{init} skew. AA mandates that if all honest parties' inputs belong to a certain range, then the outputs of the parties belong to a more concentrated range (this is a relaxation of the Agreement/Consistency property, since the honest parties do not have to perfectly agree on the output). As part of our clock synchronization objective we propose a *permissionless* AA protocol (see Section 4.2) that can achieve any constant output concentration as a function of the running time. However, naïvely using an AA protocol is still not sufficient for the goal of clock synchronization. To see that, consider parties entering as input to the AA their local timestamps; even if the AA protocol concentrates the output into a single value, due to the fact that there is no simultaneous termination, the parties adjusting their clocks to their output from the AA protocol would not result in a meaningful skew reduction.

To tackle this problem, we take a relative clock correction approach and we run AA on-chain. Recall that our mining procedure is performed using $m\times 1$ PoWs, and hence all parallel instances are mutually independent. We first use ledger agreement to continuously agree on a set of "sync-messages" with a certain bounded probability of failure in each chain individually. We let these parallel chain invocations terminate in constant time, and hence with constant probability parties will have a common view of the sync-messages on chain, and the majority of these messages will convey information from honest participants' inputs (note that some of the m chains may fail to the extent that they do not convey any useful information or even contain misleading information that the adversary injects). Periodically, each party locally computes a clock shift which is the *median* difference between a recorded sync-message on-chain and its corresponding local receiving time at the node. In our m-way parallel blockchain setting, this will result in m clock shifts. Now the nodes perform an AA-style calculation that is reminiscent of the protocol in [DLP+86]: First, outliers are eliminated from the clock-shift sequence, then the sequence is split into segments, and finally a representative is selected from each segment by returning the average. (The detailed recalculation expression is presented in Equation (5) in Section 5.2.) This guarantees a concentrated clock-shift outcome that the party can now use to adjust its local clock in a relative manner (further, note that the adjustment can be either negative or positive). This gives us:

Lemma 1 (Informal). *Assume that parties are equipped with physical clocks with bounded drift ρ, the diffusion network has bounded delays Δ and the fluctuation on computational power (resp., number of participants) is bounded for every time window of fixed length. Then, when appropriately parameterized, there exists a permissionless clock synchronization protocol that achieves (i) bounded skew (equal to $\Theta(\rho\Delta)$) between honest participants' clocks, and (ii) Γ-accuracy, for a constant $\Gamma \approx \rho$, against an adversary that controls a minority of the computational power, except with probability negligibly small in the security parameter.*

Proof (sketch). The proof is based on the following observations. First, in an (unknown yet sufficiently large) subset of the m parallel chains, in constant time, parties will reach agreement over the sync-messages. Second, each chain individually suggests a particular clock shift that a party may apply to adjust its clock — we say that a clock shift is "good" if it is on a chain that the participants agree on. Third, due to the fact that the local arrival times of the sync-messages are different for every participant, good clock shifts are bounded by the network delay Δ plus the clock drift that has taken place during this synchronization window, overall a skew of $\rho \cdot R + \Delta$, where R is the length of the window. Fourth, the approximate-agreement step will concentrate the clock shifts into output values that for all honest parties exhibit a skew very close to $\rho \cdot R + \Delta$.

That completes the bounded skew argument. Regarding accuracy, the intuition behind the proof is that fast parties will continuously adjust their clocks backwards and slow parties will similarly continuously push their clocks forward. The reason this does not achieve perfect drift ρ against nominal time is due to the network delay Δ, but by adjusting the synchronization window's length to be a multiple of Δ we can approximate ρ from above by up to any constant. □

Permissionless and self-sufficient SMR with fast fairness. Next, we show how a permissionless SMR protocol with the desired properties can be built by extending our parallel-blockchain-based clock synchronization mechanism. The starting point is the one-shot PoW-based consensus protocol in [GKS24], which enables a set of unacquainted participants to agree on a batch of transactions in expected-constant time assuming an honest majority of queries to oracle $H(\cdot)$.

Note that a sequential composition of many instances of the protocol in [GKS24] does not suffice for our purposes. To understand why, recall that in their construction, the parallel blockchains are secured via a hard-coded *density* parameter such that for a chain to be valid in an interval, it has to point to sufficiently many dense chains in the previous interval; this guarantees freshness as dense chains contain honest parties' contributions. While employing a hard-coded density parameter in the protocol helps in the **fixed participation** case, where the mining difficulty remains always the same, it fundamentally depends on having access to a shared notion of time that is not available in our setting (unless one runs a clock synchronization procedure). But even if one composes our clock synchronization protocol in parallel with their ledger protocol, the resulting scheme will not be secure in the honest majority setting.

Indeed, when the adversary has the power to influence the mining difficulty (by building chains jointly with the honest parties, or completely dropping out from the protocol), the density parameter value should be set (i) relatively small so that even if the adversary completely drops out, the honest parties alone can build chains that are sufficiently dense, as otherwise the protocol "dies"; and (ii) relatively large so that the adversary cannot mine a dense private fork alone early on before the end of an interval. These two conflicting requirements imply that with dynamic participation, hard-coding a density parameter cannot work with an adversary that controls more than one third of the computational resources, as otherwise he can break the protocol by maliciously influencing the mining target.

We now describe at a high level how our new parallel chains approach is built on top of our clock synchronization protocol. We divide time in synchronization intervals, and our protocol combines two phases, a pre-agreement phase that produces a candidate batch of transactions and an agreement phase where parties try to settle the batch of transactions. As mentioned earlier, our protocol is inspired by the "chain-king" approach proposed in [GKS24] that has parties collect transactions and submit them in the form of a batch together with their sync messages in the m parallel blockchains while it sets one of the parallel chains as the "king chain."

In the pre-agreement phase, parties observe the batch of transactions with the minimum hash value and if their input batch is different to that, they switch their input. Subsequently they follow the chain-king protocol in the agreement phase where parties will follow the batch dictated by the majority of the parallel chains, if that batch has strong support, while if not, the parties will fall back to the view provided by the king chain. Observe that this enables the ledger to advance in expected-constant time. Regarding target recalculation, we treat each blockchain independently, adjusting its mining target following the reverse Bitcoin blockchain's recalculation formula of [GKS22] at regular intervals. Finally, by a suitable sequential composition of expected-constant-time consensus protocol invocations (where we have to deal with the nuisance of non-simultaneous termination), we obtain our permissionless SMR protocol. The resulting protocol gives a distributed ledger that confirms all transactions in expected-constant time, and has no dependency on global clocks, yielding a **self-sufficient SMR protocol**. In addition, regarding newly joining parties, our state retrieval and bootstrapping mechanism allows them to learn the online parties' common view of the previous consensus invocations, which as we show is sufficient for fresh parties to synchronize with the SMR protocol state.

Regarding (fast) fairness, the high-level idea is to let parties generate their own public-key/secret-key pairs, use their public keys as identifiers, and insert them as a special transaction (recall the "coinbase" transactions in the context of the Bitcoin protocol) when they submit their batch of transactions in the pre-agreement step. This gives us:

Theorem 1 (Informal). *Assume that parties are equipped with physical clocks with bounded drift ρ, the diffusion network has bounded delay Δ, and the fluctuation on computational power (resp., number of participants) is bounded for every time window of a given fixed length. Then, assuming an appropriate parameterization, there exists a permissionless SMR protocol that achieves Consistency,* expected-constant *Liveness, Γ-Timekeeping and Fast Fairness, against an adversary that controls a minority of the computational power, except with probability negligibly small in the security parameter.*

Proof (sketch). Our analytical framework is based on that in [GKL17, GKL20]; however, we make the necessary adaptations to our setting.

As a warm-up, we first give a high-level overview of the original framework. In [GKL17, GKL20], an execution is said to be "typical" if, for any time interval of length at least λ rounds (where λ is a protocol security parameter), random variables with respect to the honest/adversarial mining success stay close to their expected value and bad events with respect to the random oracle never happen. It is then shown that when λ is sufficiently large, with overwhelming probability an execution is typical. Based on the convergence properties of the relevant random variables, an induction-style proof shows that when good events — steady block generation rate and accurate block timestamps — hold at the beginning of the execution, they will also hold in all rounds thereafter by induction. Next, all the basic blockchain properties — i.e., chain growth, common prefix and chain quality — are shown to hold throughout the execution.

Unfortunately, the same type of argument cannot directly apply to our setting, as we aim at arguing that good properties on parallel blockchains happen for *every* constant-size interval — in other words, a sequence of constant number of rounds whose security (i.e., quality of convergence) has not been a concern in typical executions.

To tackle the problem of analyzing executions for every constant-size interval, we revisit the definition of typical executions and make the following changes. First, instead of looking for good convergence on random variables for every λ rounds, we now reason about the convergence probability on every sequence of length at least ℓ rounds, where ℓ is a constant and is independent of the protocol security parameter. In other words, our new definition of "typical executions" is stronger than the previous one as it concerns properties on shorter round sequences. After reducing the minimum length of sequences that we are interested in, we then prove that for any such sequence, a set of ideal convergence properties (specified in Definition 8) hold with a constant probability. Next, we argue that, given the constant probability with which random variables yield good convergence, the ideal events of honest parties sharing a common view and producing a majority of input blocks also happens with constant probability for an interval of constant number of rounds.

The above arguments are now sufficient to apply amplification across the m parallel blockchains. Given the independence of the m parallel blockchain instances, we prove that at the end of each interval, honest parties share a common view and produce more input blocks on an (unknown) subset of more than $(1-\delta)m$ chains, where $\delta > 0$ can be an arbitrarily small constant after tuning the protocol parameters (this property is termed *interval oblivious agreement* — see Theorem 10).

Using interval oblivious agreement, we then prove that by executing the synchronization procedure and state update algorithm, honest parties maintain good skews and correctly update their internal ledger state.

Regarding Fairness, we observe that in our pre-agreement step parties will be able to mine blocks with probability proportional to the number of each party's queries to $H(\cdot)$ and the minimum hash value will select one of the transaction batches at random. Provided that the king chain is good (a constant-probability event that can be calibrated close to 1), all honest parties will switch to the same transaction batch which will satisfy the fairness property with a suitable error. Therefore, we conclude that Permissionless-SMR solves SMR with all the desired properties. $\square$

3 Preliminaries

We adopt Canetti's formulation of "real world" notion of protocol execution [Can00a, Can00b] to model the computation for multi-party protocols. The environment $\mathcal{Z}$ provides input to parties that execute the protocol Π. The adversary $\mathcal{A}$ is a single entity that controls all corrupted parties; and $\mathcal{A}$ is both "adaptive" (i.e., take control of parties on the fly) and "rushing" (i.e., $\mathcal{A}$ can observe honest parties' actions and then react). We describe the "resources" that may be available to the protocol instances (e.g., access to a "diffuse" channel) as *ideal functionalities* in the terminology of [Can00b].

3.1 Drifting Clocks and Clock Synchronization

We assume the existence of "nominal time" (cf. [BGK$^+$21, GKS22], also known as "real time" or "Newtonian time" [DHS86]), which is not directly observable by protocol participants. Following the traditional assumption in distributed computing that each party is equipped with a physical clock, whose output is a real-valued function of nominal time, in this paper we consider the **drifting clock** model. Specifically, honest parties possess physical clocks with a bounded rate of drift from nominal time (Figure 1) — i.e., an honest physical clock stays within a *linear envelope* of nominal time.[5] We use ρ to denote the bound on the rate of honest physical clocks. Formally, for any nominal time $u > v$ and an honest physical clock D, it holds that

$$(1+\rho)^{-1}(u-v) \le D(u) - D(v) \le (1+\rho)(u-v).$$

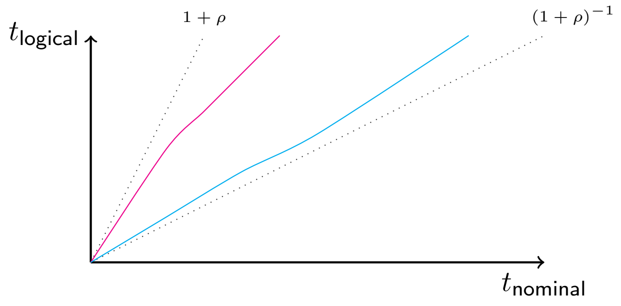

Figure 1: *An illustration of drifting clocks within a* $(0.5, 2)$*-linear envelope (i.e.,* $\rho = 1$*). Without clock synchronization, two clocks (illustrated in magenta and cyan, respectively) deviate from each other unboundedly.*

Note that time in the drifting clock model are real values, while our protocol execution model divides time into discrete integer-numbered steps. In order to “insert” real-valued time into this structure, we model drifting local clocks as $\mathcal{G}_{\mathsf{DClock}}$, derived from the global clock functionality in [KMTZ13]. In $\mathcal{G}_{\mathsf{DClock}}$, nominal time is defined by the number of times that the clock functionality moves forward the time-step variable τ (which is an internal variable and is unknown to the parties). Instead of directly receiving the time from $\mathcal{G}_{\mathsf{DClock}}$, parties receive “ticks” from $\mathcal{G}_{\mathsf{DClock}}$ which indicates that they should advance their local round number. $\mathcal{G}_{\mathsf{DClock}}$ advances the nominal time when all (honest) participants claim they have finished their computation in the current round. Additionally, $\mathcal{G}_{\mathsf{DClock}}$ allows the adversary to “push” or “stall” honest clocks, as long as these operations do not violate the ρ-bounded linear envelope assumption. We present $\mathcal{G}_{\mathsf{DClock}}$ in Functionality 1.

Remark 1. As opposed to the classical distributed computing setting (cf. [DLS88]) where rounds/time steps are defined as intervals of equal length in the view of an external real-time clock, in our model $\mathcal{G}_{\mathsf{DClock}}$ does not guarantee that rounds/time steps are of equal duration. In fact, the notions of ‘time’ and ‘duration’ are not defined in the UC setting. Yet, our model shows the same effect in restricting local clock counters staying in a bounded linear envelope with respect to the nominal clock counter; moreover, the same amount of computation is carried out per local time counter increment — which means that in the same window of nominal time, a faster CPU will solve more PoWs than a slower one.

[5]A function $f : \mathbb{R} \to \mathbb{R}$ is within a (U, L)-linear envelope if and only if it holds that $L \cdot x - c \le f(x) \le U \cdot x + c$, where c is a constant and $x \in \mathbb{R}^+$.

Functionality $\mathcal{G}_{\mathsf{DClock}}$

This functionality maintains state variables as follows.

State Variable	**Description**
$\rho \leftarrow 0$	The bound on clock drifts.
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties $\mathsf{P} = (\text{pid}, \text{sid})$.
$F \leftarrow \emptyset$	The set of registered functionalities (together with their session identifier).
$\tau_{\text{sid}} \leftarrow 0$	The nominal-time variable for session sid.
$d_{\mathsf{P}} \leftarrow 0$	The clock-update variable for $\mathsf{P} = (\text{pid}, \text{sid}) \in \mathcal{P}$. d_{P} is set to 1 after P finishes a round.
$b_{\mathsf{P}} \leftarrow 1$	The tick-budget variable for $\mathsf{P} = (\text{pid}, \text{sid}) \in \mathcal{P}$.
$d(\mathcal{F}, \text{sid}) \leftarrow 0$	The clock-update variable for $(\mathcal{F}, \text{sid}) \in F$.

Setting the drift:

- Upon receiving $(\textsc{set-drift}, \text{sid}, r)$ from the adversary $\mathcal{A}$, **if** SET-DRIFT has never been received then set $\rho = r$. Return $(\textsc{set-drift}, \text{sid}, \mathsf{ok})$ to $\mathcal{A}$.

Clock capabilities:

- Upon receiving $(\textsc{clock-update}, \text{sid}_C)$ from some party $\mathsf{P} \in \mathcal{P}$ set $d_{\mathsf{P}} \leftarrow 1$ and $b_{\mathsf{P}} \leftarrow b_{\mathsf{P}} - 1$; execute *Time-Update* and forward $(\textsc{clock-update}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\textsc{clock-update}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d(\mathcal{F}, \text{sid}) \leftarrow 1$, execute *Time-Update* and return $(\textsc{clock-update}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\textsc{clock-forward}, \text{sid}_C, \mathsf{P})$ from $\mathcal{A}$ where $\mathsf{P} \in \mathcal{P}$, if $d_{\mathsf{P}} = 0$ or it is about to violate the ρ-bounded linear envelope on P, ignore the message. Otherwise, update $d_{\mathsf{P}} = 0$ and $b_{\mathsf{P}} \leftarrow b_{\mathsf{P}} + 1$; return $(\textsc{clock-forward-ok}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\textsc{clock-backward}, \text{sid}_C, \mathsf{P})$ from $\mathcal{A}$ where $\mathsf{P} \in \mathcal{P}$, if $d_{\mathsf{P}} = 0$ or it is about to violate the ρ-bounded linear envelope on P, ignore the message. Otherwise, update $b_{\mathsf{P}} \leftarrow b_{\mathsf{P}} - 1$; return $(\textsc{clock-backward-ok}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\textsc{clock-tick}, \text{sid}_C)$ from any participant P— including the environment on behalf of a party — or the adversary on behalf of a corrupted party P (resp. from any ideal—shared or local—functionality $\mathcal{F}$), execute procedure *Time-Update*, return $(\textsc{clock-tick}, \text{sid}_C, d_{\mathsf{P}})$ (resp. $(\textsc{clock-tick}, \text{sid}_C, d_{(\mathcal{F},\text{sid})})$) to the requestor (where sid is the session id of the calling instance).
- Upon receiving $(\textsc{clock-read}, \text{sid}_C)$ from the adversary or the wapper functionalities, return $(\textsc{clock-read}, \text{sid}_C, \tau_{\text{sid}})$ to the requestor (where sid is the session id of the calling instance).

Procedure Time-Update: For each session sid do: If (i) $d_{(\mathcal{F},\text{sid})} = 1$ for all $\mathcal{F} \in F$, and (ii) $d_{\mathsf{P}} = 1$ and $b_{\mathsf{P}} \leq 0$ for all honest parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$, then update $\tau_{\text{sid}} \leftarrow \tau_{\text{sid}} + 1$, $d_{(\mathcal{F},\text{sid})} \leftarrow 0$ and $b_{\mathsf{P}} \leftarrow b_{\mathsf{P}} + 1$ for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$. Additionally, for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$ with $b_{\mathsf{P}} > 0$, update $d_{\mathsf{P}} \leftarrow 0$.

Functionality 1: *The drifting global clock.*

We adapt the traditional definition of the clock synchronization problem (cf. [DHS86, ST87]) to our permissionless setting. In Definition 1, we consider two properties, *bounded skew* and *accuracy*, that establish upper bounds Φ and Γ on honest clock skew and their deviation from the nominal time, respectively.

Definition 1 (Clock Synchronization). *There exist constants $\varPhi \in \mathbb{N}$, $\varGamma \in \mathbb{R}^+$ such that honest parties' logical clocks satisfy the following two properties:*

- **Bounded skew (with parameter $\varPhi \in \mathbb{N}$).** *Let $\mathtt{r}_1, \mathtt{r}_2$ be the reported logical clocks of two honest parties at any nominal time r. Then $|\mathtt{r}_1 - \mathtt{r}_2| \leq \varPhi$.*
- **Accuracy (with parameter $\varGamma \in \mathbb{R}^+$).** *Each honest party's logical clock stays in a (U, L)-linear envelope with respect to the nominal time r, where $U = 1 + \varGamma$ and $L = 1/(1 + \varGamma)$.*

3.2 Random Oracle, Network and Adversarial Model

Random oracle. By convention, we model the cryptographic hash function H with output in $\{0,1\}^\kappa$ (which is used to generate proofs of work [PoWs]) as a random oracle $\mathcal{F}_{\mathsf{RO}}$ [BR93].

Functionality $\mathcal{F}_{\mathsf{RO}}$

The functionality is parameterized by a security parameter κ.

State Variable	Description
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties.
$H \leftarrow \emptyset$	A dynamically updatable function table where $H[x] = \bot$ denotes the fact that no pair of the form $(x, \cdot)$ is in H.

- **Eval.** Upon receiving $(\textsc{eval}, \mathrm{sid}, x)$ from some party $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted P), do the following:
  1. If $H[x] = \bot$ sample a value y uniformly at random from $\{0,1\}^\kappa$ and set $H[x] \leftarrow y$.
  2. Return $(\textsc{eval}, \mathrm{sid}, x, H[x])$ to the requestor.

Functionality 2: *The random oracle.*

We express our honest majority condition in terms of computational power, measured in particular by the number of queries to the RO that the parties are allowed to make per *nominal time-step*, as opposed to expressing it by the number of parties (i.e., the "flat" model where parties are assumed to have equal computational power—cf. [GKL15]).

Definition 2 (Honest Majority). *Let h_r, t_r denote the number of* alert *and* non-alert *random oracle queries at* nominal *time r, respectively. Then, for all $r \in \mathbb{N}$, it holds that $h_r > t_r$.*

This restriction on the number of RO queries is captured by a wrapper functionality on $\mathcal{F}_{\mathsf{RO}}$ via counting the number of *alert* and other queries (see below) per nominal-time step. The adversary is allowed to dynamically and adaptively determine the number of alert random oracle queries per nominal-time step, as long as it does not violate the restrictions imposed by the (γ, s)-respecting environment (see Definition 3 in the sequel).

Functionality $\mathcal{W}(\mathcal{F}_{\mathsf{RO}})$

This functionality maintains state variables as follows.

State Variable	Description
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties; the current set of corrupted parties is denoted by $\mathcal{P}'$.
$\tau \leftarrow 0$	The (real-time) clock tick counter.
h_τ	An upper bound which restricts the $\mathcal{F}$-evaluations of all alert parties at time τ.
$q_\mathcal{H}, q_\mathcal{A} \leftarrow 0$	The alert/adversary evaluation counter.

Pre-mining attack handling (executed only if $\tau = 0$):

- Upon receiving $(\textsc{eval}, \text{sid}, x)$ from $\mathcal{A}$ on behalf of a corrupted party $P \in \mathcal{P}'$, forward the request to $\mathcal{F}_{\mathsf{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\mathsf{RO}}$ returns.
- Upon receiving $(\textsc{Retrieved}, \text{sid})$ from $\mathcal{F}_{\mathsf{CRS}}$, set $\tau = 1$.

Relaying inputs to the random oracle:

- Upon receiving $(\textsc{eval}, \text{sid}, x)$ from $\mathcal{A}$ on behalf of a corrupted party $P \in \mathcal{P}'$ or a de-synchronized party P, first execute *Round Reset*, then do the following.
  1. Set $q_\mathcal{A} \leftarrow q_\mathcal{A} + 1$.
  2. **If** $q_\mathcal{A} \leq h_\tau$ **then** forward the request to $\mathcal{F}_{\mathsf{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\mathsf{RO}}$ returns.
- Upon receiving $(\textsc{eval}, \text{sid}, x)$ from an alert party P, first execute *Round Reset*, then do the following.
  1. Set $q_\mathcal{H} \leftarrow q_\mathcal{H} + 1$.
  2. **If** $q_\mathcal{H} \leq h_\tau$ **then** forward the request to $\mathcal{F}_{\mathsf{RO}}$ and return to P whatever $\mathcal{F}_{\mathsf{RO}}$ returns.
  3. **If** $q_\mathcal{H} \geq h_\tau$ **then** send $(\textsc{clock-update}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$.

Corruption handling:

- Upon receiving $(\textsc{corrupt}, \text{sid}, \mathsf{P})$ from the adversary, set $\mathcal{P}' \leftarrow \mathcal{P}' \cup \mathsf{P}$.

Procedure Round-Reset: Send $(\textsc{clock-read}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$ and receive $(\textsc{clock-read}, \text{sid}_C, \tau')$ from $\mathcal{G}_{\mathsf{DClock}}$. If $|\tau - \tau'| > 0$, then do the following.

1. Set $q_\mathcal{H}, q_\mathcal{A} \leftarrow 0$ and $\tau \leftarrow \tau'$.
2. Send $(\textsc{next-round})$ to $\mathcal{A}$ and receive as response $(\textsc{next-round}, h^*_{\tau'})$. **If** $(h_1, h_2, \ldots, h'_{\tau'})$ is (γ, s)-respecting (Definition 3) **then** set $h_{\tau'} = h'_{\tau'}$; **else** set $h_{\tau'} = h_{\tau'-1}$.

Functionality 3: *The random oracle wrapper.*

In addition, pre-mining attack prevention is captured by restricting the number of adversarial queries after a fresh CRS is released (which we model as $\mathcal{F}_{\mathsf{CRS}}$).

Functionality $\mathcal{F}_{\mathsf{CRS}}$

The functionality is parameterized by a distribution $\mathcal{D}$.

- **Retrieve.** Upon receiving $(\textsc{Retrieve}, \text{sid})$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), do the following:
  1. If activated for the first time, choose a value $d \leftarrow \mathcal{D}$, and send $(\textsc{Retrieved}, \text{sid})$ to $\mathcal{W}(\mathcal{F}_{\mathsf{RO}})$

and $\mathcal{W}(\mathcal{F}_{\mathsf{Diffuse}})$.

2. Return $(\textsc{Retrieve}, d)$ to P.

Functionality 4: *The common reference string.*

The bounded-delay network. Regarding communication amongst parties, we consider a peer-to-peer diffusion network, where the message dissemination has an (unknown) Δ-bounded delay. In more detail, an honest message sent at time t will be received by all other honest parties before time $t + \Delta$; regarding messages sent by the adversary, if t is the earliest time such that at least one honest party receives those messages, they are guaranteed to be delivered to all honest parties before time $t + \Delta$ (i.e., honest parties keep "echoing" messages).

We capture this communication network with $\mathcal{F}_{\mathsf{Diffuse}}$ (Functionality 5). Recall that existing diffuse functionalities (cf. [BMTZ17]) model delays in the following manner: There is a fetch counter per message per recipient such that when each time an honest party P is activated and received a new tick from the clock, P fetches on $\mathcal{F}_{\mathsf{Diffuse}}$ which reduces counters by 1 for all messages delivering to P; she then receives a subset of those messages with counters reset to 0. Regarding the adversary, he can increase the counter for each message and recipient for up to Δ in accumulation (as well as swapping the order of messages).

By convention, different types of messages are diffused by different functionalities, and we write $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{input}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ to denote the network for chains, input blocks and transactions.

Functionality $\mathcal{F}_{\mathsf{Diffuse}}$

This functionality maintains state variables as follows.

State Variable	Description
$\Delta \leftarrow 0$	The maximum network latency.
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties.
$\vec{M} \leftarrow []$	A dynamically updatable list of quadruples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ where D_{mid} denotes the fetch counter.

Setting the delay:

- Upon receiving $(\textsc{set-delay}, \mathrm{sid}, d)$ from the adversary $\mathcal{A}$, **if** $\textsc{set-delay}$ has never been received **then** set $\Delta = d$. Return $(\textsc{set-delay}, \mathrm{sid}, \mathsf{ok})$ to $\mathcal{A}$.

Network capabilities:

- Upon receiving $(\textsc{diffuse}, \mathrm{sid}, m)$ from some $\mathsf{P}_s \in \mathcal{P}$, where $\mathcal{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ denotes the current party set, do:
  1. Choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$.
  2. Initialize $2n$ new variables $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} \ldots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$ and a per message-delay $\Delta_{\mathsf{mid}_i} = \Delta$ for $i \in [n]$.
  3. Set $\vec{M} := \vec{M} \,\|\, (m, \mathsf{mid}_1, D_{\mathsf{mid}_1}, \mathsf{P}_1) \,\|\, \ldots \,\|\, (m, \mathsf{mid}_n, D_{\mathsf{mid}_n}, \mathsf{P}_n)$.
  4. Send $(\textsc{diffuse}, \mathrm{sid}, m, \mathsf{P}_s, (\mathsf{P}_1, \mathsf{mid}_1), \ldots, (\mathsf{P}_n, \mathsf{mid}_n))$ to the adversary.
- Upon receiving $(\textsc{fetch}, \mathrm{sid})$ from $\mathsf{P} \in \mathcal{P}$, or from $\mathcal{A}$ on behalf of a corrupted party P:
  1. For all tuples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P}) \in \vec{M}$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.

2. Let $\vec{M}_0^{\mathsf{P}}$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ with $D_{\mathsf{mid}} \leq 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}_0^{\mathsf{P}}$ from $\vec{M}$ and in case some $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ is in $\vec{M}_0^{\mathsf{P}}$, where P is honest, set $\Delta_{\mathsf{mid}'} = \Delta$ for any $(m, \mathsf{mid}', D_{\mathsf{mid}'}, (\cdot, \text{sid}))$ in $\vec{M}$ and replace this record by $(m, \mathsf{mid}', \min\{D_{\mathsf{mid}'}, \Delta\}, \mathsf{P}')$.
3. Output $\vec{M}_0^{\mathsf{P}}$ to P (if P is corrupted, send $\vec{M}_0^{\mathsf{P}}$ to $\mathcal{A}$).

Additional adversarial capabilities:

- Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some corrupted $\mathsf{P}_s \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P_s if corrupted), execute it the same way as an honest-sender diffuse, with the only difference that $\Delta_{\mathsf{mid}_i} = \infty$.
- Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\mathsf{mid}_{i_j}}, \mathsf{mid}_{i_j})$: if $D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}} \leq \Delta_{\mathsf{mid}_{i_j}}$ and mid_{i_j} is a message-ID of receiver $\mathsf{P} = (\cdot, \text{sid})$ registered in the current $\vec{M}$, set $D_{\mathsf{mid}_{i_j}} := D_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$ and set $D^{MAX}_{\mathsf{mid}_{i_j}} := D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$; otherwise, ignore this pair.
- Upon receiving $(\text{SWAP}, \text{sid}, \mathsf{mid}, \mathsf{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \mathsf{mid}, D_{\mathsf{mid}}, (\cdot, \text{sid}))$ and $(m, \mathsf{mid}', D_{\mathsf{mid}'}, (\cdot, \text{sid}))$ in $\vec{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.
- Upon receiving $(\text{GET-REG}, \text{sid})$ from $\mathcal{A}$, return the response $(\text{GET-REG}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

Functionality 5: *The diffusion network.*

We highlight that such mechanism does *not* work in our drifting clock model with $\mathcal{G}_{\mathsf{DClock}}$ (it only works with a global clock where parties proceed with the same speed). This is because, when modeling delays via fetches on the diffuse functionality, honest parties that experience relatively fast local rounds would request more FETCH commands than the slow ones in the same window of nominal time — i.e., if an honestly-sent message is set the same delay for two parties then it delivers to the fast one earlier; yet our goal is to model delays measured in the perspective of the nominal time (regardless of parties' local understanding of time).

We resolve this issue by introducing a new wrapper on $\mathcal{F}_{\mathsf{Diffuse}}$ (Functionality 6) that restricts the adversary's capability to delay messages for up to Δ nominal time. After registering on $\mathcal{G}_{\mathsf{DClock}}$ to learn the nominal time ticks, the wrapper functionality dynamically relays parties' FETCH request to $\mathcal{F}_{\mathsf{Diffuse}}$ so that in every nominal-time step, exactly one fetch operation is relayed to $\mathcal{F}_{\mathsf{Diffuse}}$ for each honest party (even if that honest party receives multiple ticks from $\mathcal{G}_{\mathsf{DClock}}$). Meanwhile, for each party P that proceeds slowly and may not activate in a given nominal time, the wrapper queries $\mathcal{F}_{\mathsf{Diffuse}}$ on behalf of P, buffers the response messages and delivers them to P upon the next time P interacts with the wrapper.

Functionality $\mathcal{W}(\mathcal{F}_{\mathsf{Diffuse}})$

This functionality maintains state variables as follows.

State Variable	Description
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties; the current set of corrupted parties is denoted by $\mathcal{P}'$.
$\tau \leftarrow 0$	The (real-time) clock tick counter.

$\mathsf{fetch}(\mathsf{P}, \tau) \leftarrow 0$	The fetch variable for party P at nominal time τ.
$\mathsf{buffer}_{\mathsf{P}} \leftarrow []$	The fetch buffer for party P.

Relaying inputs to the diffuse network:
- Upon receiving $(\textsc{diffuse}, \mathrm{sid}, m)$ from $\mathcal{A}$ on behalf of some corrupted $\mathsf{P} \in \mathcal{P}'$, parse m as blocks $\mathcal{B}_1, \ldots, \mathcal{B}_n$. For each $\mathcal{B}_i$, if $\mathcal{B}_i$ has not been queried to $\mathcal{F}_{\mathsf{RO}}$, send $(\textsc{eval}, \mathrm{sid}, \mathcal{B}_i)$ from a corrupted party.
- Upon receiving $(\textsc{fetch}, \mathrm{sid})$ from an honest party P, if $\mathsf{fetch}(\mathsf{P}, \tau) = 1$ ignore this request. Otherwise, execute the following:
  1. Forward $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}$ and recevie as response $\vec{M}$, return $\mathsf{buffer}_{\mathsf{P}} \parallel \vec{M}$.
  2. Set $\mathsf{buffer}_{\mathsf{P}} \leftarrow []$ and $\mathsf{fetch}(\mathsf{P}, \tau) \leftarrow 1$.

Corruption handling:
- Upon receiving $(\textsc{corrupt}, \mathrm{sid}, \mathsf{P})$ from the adversary, set $\mathcal{P}' \leftarrow \mathcal{P}' \cup \mathsf{P}$.

Procedure Round-Reset: Send $(\textsc{clock-read}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$ and receive $(\textsc{clock-read}, \mathrm{sid}_C, \tau')$ from $\mathcal{G}_{\mathsf{DClock}}$. If $|\tau - \tau'| > 0$, then do the following.
1. Set $\tau \leftarrow \tau'$.
2. For each honest party P such that $\mathsf{fetch}(\mathsf{P}, \tau) = 0$, send $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}$ from P and recevie as response $\vec{M}$, set $\mathsf{buffer}_{\mathsf{P}} \leftarrow \mathsf{buffer}_{\mathsf{P}} \parallel \vec{M}$. For each honest party P such that $\mathsf{fetch}(\mathsf{P}, \tau) = 1$, set $\mathsf{fetch}(\mathsf{P}, \tau) \leftarrow 0$.
3. Send $(\textsc{clock-update}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$.

Functionality 6: *The wrapper of diffuse network.*

Dynamic availability and respecting environment. In order to apply a more fine-grained classification on protocol participants, we follow the treatment in [BGK+18] and classify parties into different types based on their accessible resources and synchronization states. Specifically, a party is (i) *operational* if she is registered with the random oracle $\mathcal{F}_{\mathsf{RO}}$, and *stalled* otherwise; (ii) *online* if she is registered with the network $\mathcal{F}_{\mathsf{Diffuse}}$, and *offline* otherwise; (iii) *time-aware* if she is registered with the drifting clock $\mathcal{G}_{\mathsf{DClock}}$, and *time-unaware* otherwise; and (iv) *synchronized* if she has been participated in the protocol for sufficiently long time and held "synchronized state" and "synchronized time" with other synchronized parties, and *desynchronized* otherwise.

We define *alert* parties based on the classification above. Specifically, alert parties are those who have access to all the resources and are synchronized. They are the core set of parties to carry out the protocol.

Next, we define a "respecting environment" in terms of the computational power (cf. [GKS22]) as opposed to number of parties (cf. [GKL17, GKL20]). Our honest-majority assumption is that during the whole protocol execution, the alert computational power is higher than the adversarial one. We restrict the environment so that the number of such queries will be bounded in a certain fashion.

Definition 3. *For $\gamma \in \mathbb{R}^+$ we call the sequence $(h_r)_{r \in [0,B)}$, where $B \in \mathbb{N}$, (γ, s)-respecting if for any set $S \subseteq [0, B)$ of at most s consecutive integers, $\mathsf{max}_{r \in S} h_r \leq \gamma \cdot \mathsf{min}_{r \in S} h_r$.*

3.3 Blockchain Notation

A block with target $T \in \mathbb{N}$ is a quadruple of the form $\mathcal{B} = \langle ctr, r, h, x \rangle$ where $ctr, r \in \mathbb{N}$, $h \in \{0,1\}^{\kappa}$ and $x \in \{0,1\}^{*}$. A blockchain $\mathcal{C}$ is a (possibly empty) sequence of blocks; the rightmost block is denoted by head$(\mathcal{C})$ (note head$(\varepsilon) = \varepsilon$). These blocks are chained in the sense that if $\mathcal{B}_{i+1} = \langle ctr, r, h, x \rangle$, then $h = H(\mathcal{B}_i)$. We use $\mathsf{TS}(\mathcal{B})$ to denote the *timestamp* of $\mathcal{B}$. We denote by $\mathcal{C}^{t\rceil}$ the chain resulting from "pruning" the least number of rightmost blocks so that all block timestamps are less than t. Let $\mathbb{C} = \langle \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m \rangle$ denote m *parallel* chains and $\mathbb{C}_j$ the j-th chain $\mathcal{C}_j$ in $\mathbb{C}$.

Next, we introduce some basic string notation, which will be useful when describing our multi-chain-oriented PoW mechanism. For a κ-bit string s, where κ is the security parameter, we will use s_i ($i \in [m]$) to denote the i-th bit of s, $[s]_{i\sim m}$ to denote the i-th segment after s is equally divided into m segments—i.e., $[s]_{i\sim m} = s_{[(i-1)*\kappa/m]+1}, \ldots, s_{i*\kappa/m}$. Further, we will write $[s]^{\mathsf{R}}$ as the reverse of string s (i.e., the string obtained by reversing the order of its bits), and use $[s]^{\mathsf{R}}_{i\sim m}$ to denote the reverse of the i-th segment.

Finally, we introduce some array operations, following the notation in [DLP+86], that will be useful when describing the protocol. Let $\mathbf{V} = (v_1, \ldots, v_n)$ be a real array of n elements and denote $\tilde{\mathbf{V}} = (\tilde{v}_1, \ldots, \tilde{v}_n)$ the array after ordering $\mathbf{V}$ non-decreasingly. We define the operations reduce and select. Intuitively, reduce with parameter $\mathbf{V}$ and η first orders $\mathbf{V}$ non-decreasingly and then "trims" the η largest and η smallest elements in $\tilde{\mathbf{V}}$; select with parameter $\mathbf{V}$ and η first orders $\mathbf{V}$ non-decreasingly and then selects every first element of η consecutive elements and forms them as a new array. Formally,

$$\mathsf{reduce}(\mathbf{V}, \eta) = (\tilde{v}_{\eta+1}, \ldots, \tilde{v}_{n-\eta}) \text{ and } \mathsf{select}(\mathbf{V}, \eta) = (\tilde{v}_1, \tilde{v}_{\eta+1}, \tilde{v}_{2\eta+1}, \ldots). \tag{1}$$

Additionally, for an array $\mathbf{V}$, let $\mathsf{avg}(\mathbf{V})$ denote the average of its elements, and $\mathsf{med}(\mathbf{V})$ denote the median.

3.4 Weak/Approximate Agreement

Weak agreement. A variant of Byzantine agreement — *Weak Agreement* – relaxes the agreement property to allow some parties to output a special failing symbol $\perp$ while requiring all non-$\perp$ outputs being consistent [Dol82]. This simple primitive has been widely used in building stronger notion of agreement (e.g., graded agreement). We here provide the definition of Weak Agreement.

Definition 4 (Weak Agreement). *A protocol* Π *implements Weak Agreement provided it satisfies the following two properties:*
- **Weak Agreement***: There exists* $y \in \{0,1\}$ *such that all honest parties output* $y_i \in \{y, \perp\}$*.*
- **Validity***: If all honest parties have the same input* $b \in \{0,1\}$*, they all output* $y_i = b$*.*

Approximate agreement. *Approximate Agreement* (AA), formulated in [DLP+86], is a variant of the Byzantine agreement problem [PSL80, LSP82] in which processes start with arbitrary real values rather than Boolean values or values from some bounded range, and in which approximate, rather than exact, agreement is the desired goal. In the classical distributed computing literature, AA has served as a fundamental building block to achieve clock synchronization (see, e.g., [LL84b, MS85, LL22]).

Next, we present a definition of AA which captures adaptive security. Conventionally, the goal is to let honest parties agree on outputs where the difference between any two of them is upper-bounded by a constant. In this paper, we provide an alternate yet equivalent definition which asks for a "concentration" on the outputs compared with the inputs. We denote ϵ as the "quality" of the agreement—i.e., the ratio between honest-input distance and output distance.

Definition 5 (Approximate Agreement). *A protocol* Π *is an* ϵ*-secure protocol for Approximate Agreement provided it satisfies the following two properties:*

- ϵ**-Agreement***: There is a round after which (i) any two honest parties hold inputs with difference at most* ℓ*, and (ii) any two honest parties return outputs with difference at most* $\epsilon \cdot \ell$ *(where* $0 < \epsilon < 1$*) if queried by the environment.*
- **Validity***: The output returned by an honest party* P *falls in the convex hull of the inputs of all parties at round* 1 *that are honest at the round* P*'s output is produced.*

4 Permissionless Weak/Approximate Agreement

In this section we consider weaker forms of agreement — namely, weak agreement and approximate agreement as introduced in Section 3.4 — in the permissionless setting, and show that, when appropriately parameterized, they can be achieved in constant time against a minority of corrupted computational power. We present two protocols: first, a protocol that is simple and easy to understand, but which tolerates only one-third of corrupted computational power. We then improve this protocol to tolerate a dishonest minority and, in the context of AA, achieve better output quality (recall Definition 5).[6]

For simplicity, we consider a single protocol invocation and present our protocols in the *static* setting where parties are always online and their number is fixed yet unknown to the protocol participants, and we assume a global clock and a diffusion network with Δ-bounded delay. Later, in Section 5, we use the weak/approximate agreement protocols as sub-routines in a dynamic environment with drifting clocks.

4.1 Weak/Approximate Agreement against a $(1/3)$-Bounded Adversary

We first provide a simple protocol that solves Weak/Approximate Agreement against an adversary that controls up to one-third of the computational power. At a high level, in this protocol parties first use $m{\times}1$ PoW to mine and exchange messages containing their input value (concatenated with a random string for uniqueness) for a pre-determined number of ℓ rounds; then each party P decides its output locally based on the messages that it has received.

The protocol is parameterized by ℓ (the termination time), m (the number of parallel mining procedures) and T (a pre-determined PoW target value). A party P starting with input v queries the RO and gets $u = H(r \parallel v)$ where r is a κ-bit random string. For every "chunk" $[u]_{i\sim m}$ of κ/m bits, P checks if $[u]_{i\sim m} < T$. If P solves a PoW in any of the chunks, it diffuses (r, v) to the network. Note that (r, v) will be counted independently for different succeeding chunks; e.g., if a valid PoW is found in $m/2$ segments, (r, v) will be counted for $m/2$ times. Parties keep listening to the network and book-keep all valid PoW messages, extracting their corresponding values in an array $\mathbf{V}$.

Parties decide their output based on their local array $\mathbf{V}$ at the end of round ℓ. For Weak Agreement, they output 0 (1 resp.) iff. more than two-thirds of the elements in $\mathbf{V}$ are 0s (1s resp.); and $\perp$ otherwise. Regarding Approximate Agreement, the output decision procedure then follows the approach in [DLP+86]: For an ordered array $\mathbf{V}$ of size n, the smallest and largest $n/3$ values are dropped and the party outputs the midpoint of the remaining values (i.e., the average of the maximum and minimum element).

[6]Translating protocols presented in this section to the PoS setting (with dynamic availability) is straightforward, as the core mechanism that we use—$m{\times}1$ PoW—can be emulated by independently evaluating multiple different VRF outputs.

The full description of the protocol is presented in Protocol 1.

Protocol $\frac{1}{3}$$-\mathsf{ApproxAgr}(\ell, m, T)$

```
    ▷ The following code is executed in each round r (val is the party input)
 1: if $\mathtt{r} \le \ell$ then                    ◁ Step 1: mine and exchange messages.
 2:     Fetch information and denote incoming PoW messages by $((r_1, v_1), \ldots, (r_n, v_n))$
 3:     for each $(r_i, v_i)$ do
 4:         Set $c = |\{j \mid j \in [m] \wedge [H(r_i, v_i)]_{j\sim m} < T\}|$
 5:         Append $v_i$ to $\mathbf{V}$ for $c$ times
 6:     end for
 7:     Set $u \leftarrow H(ctr, \mathtt{val})$
 8:     if $\exists j \in [m]$ s.t. $[u]_{j\sim m} < T$ then Diffuse $(ctr, \mathtt{val})$
 9:     $ctr \leftarrow ctr + 1$
10: else                                               ◁ Step 2: decide output.
11:     Order $\mathbf{V}$ non-decreasingly and denote $n \leftarrow |\mathbf{V}|$
12:     if deciding output for weak agreement then
13:         if $\mathbf{V}_{n/3} = \mathbf{V}_{2n/3}$ then output $\mathbf{V}_{n/3}$ else output $\perp$
14:     else if deciding output for approximate agreement then
15:         Output the mid point of $\mathsf{reduce}(\mathbf{V}, n/3)$
16:     end if
17: end if
```

Protocol 1: *Weak/Approxmate Agreeement tolerating a* (1/3)*-bounded adversary.*

Note that different parties may work on different sets of messages when the adversary chooses to hold and deliver its own PoW message at the last round (so that honest parties have no time to diffuse this message to others). Nonetheless, for sufficiently large κ and suitable value of ℓ, the messages generated by different parties are roughly proportional to their computational power. Thus, for all arrays held by honest parties, they will share a large (yet unknown) common subset.

Lemma 2. *Let Δ denote the upper bound on network delay, h and t denote the number of honest and adversarial queries per round ($h > 2t$), and m be the number of independent PoW mining. Let V denote the honest input set ($V = \{0, 1\}$ for Weak Agreement and $V \subseteq \mathbb{R}$ for Approximate Agreement) and $\mathbf{V}$ denote the ordered array of PoW messages received by party P after round ℓ (line 11 in Protocol 1), and assume that $\ell > 4\Delta$ and $m = \Theta(\log^2 \kappa)$. Then, for any two honest parties P, P' (possibly $\mathsf{P} = \mathsf{P}'$) and $\mathbf{V}, \mathbf{V}'$ with size n, n', respectively, it holds that*

$$\mathsf{min}V \le \mathbf{V}_{n/3} \le \mathbf{V}'_{2n'/3} \le \mathsf{max}V,$$

except with probability negligible in κ.

Proof. Let $\mathbf{V}^h \subseteq \mathbf{V}$ denote the subset of messages generated by the honest parties before round $\ell - \Delta$. Let $\mathcal{H}$ denote the set of honest parties at the end of round ℓ, and suppose it holds that

$$\forall \mathsf{P}_j \in \mathcal{H}, \Big| \bigcap_{\mathsf{P}_i \in \mathcal{H}} \mathbf{V}_i^h \Big| \ge 2|\mathbf{V}_j|/3, \tag{2}$$

then for $\mathbf{V}$ held by any honest party, it would hold that $\mathsf{min}V \le \mathbf{V}_{n/3} \le \mathsf{med}(\bigcap_{\mathsf{P}_i \in \mathcal{H}} \mathbf{V}_i^h) \le \mathbf{V}_{2n/3} \le \mathsf{max}V$, which would conclude the proof.

Now we only need to verify Equation (2). Let X denote the number of PoW messages honest parties generated in the first $(\ell - \Delta)$ rounds, and Y denote the number of successful PoWs the adversary generated in the first ℓ rounds, plus all the honest PoWs in the last Δ rounds. Since $\ell > 4\Delta$, we have $\mathbb{E}[X] > \mathbb{E}[2Y]$. Note that $\mathbb{E}[X] = \Theta(\mathbb{E}[2Y]) = \Theta(\log^2 \kappa)$. Now, by letting $\epsilon \in (0, (\mathbb{E}[X]/\mathbb{E}[2Y] - 1)/2)$ be a constant and applying the Chernoff bound (Theorem 5), we get

$$\mathbf{Pr}[X > 2Y] \geq \mathbf{Pr}[X > (1-\epsilon)\mathbb{E}[X]] \wedge \mathbf{Pr}[2Y < (1+\epsilon)\mathbb{E}[2Y]] \geq 1 - \exp(-\Omega(\log^2 \kappa)),$$

which happens with overwhelming probability in κ, the security parameter. □

Given the good properties satisfied by $\mathbf{V}$ as described in Lemma 2, we conclude that (i) by comparing $\mathbf{V}_{n/3}$ and $\mathbf{V}_{2n/3}$, honest parties reach Weak Agreement; and (ii) by applying the midpoint on reduce, honest parties reach Approximate Agreement with concentration quality 1/2. We prove this for Approximate Agreement in Theorem 2 (and Weak Agreement can be argued alike).

Theorem 2. *Under the same assumption as in Lemma 2, Protocol 1 is a* (1/2)*-secure Approximate Agreement protocol against an adversary that controls less than one-third of the computational power, except with negligible probability in the security parameter.*

Proof. Validity follows directly, since for any honest party P_i, all values in $\mathsf{reduce}(\mathbf{V}_i, |\mathbf{V}_i|/3)$ are within the convex hull of the honest inputs, and so is the midpoint.

Regarding (1/2)-Agreement, note that there exist a value u such that for any honest party P and $\mathbf{V}$, $\mathbf{V}_{n/3} \leq u \leq \mathbf{V}_{2n/3}$ (cf. Lemma 2). For any two honest parties P and P' and their respective outputs v, v', let v_1, v_2 (v_1', v_2', resp.) denote the smallest and largest values in $\mathsf{reduce}(\mathbf{V}_i, |\mathbf{V}_i|/3)$ ($\mathsf{reduce}(\mathbf{V}_i', |\mathbf{V}_i'|/3)$, resp.). We have

$$|v - v'| = (|v_1 - v_1'| + |v_2 - v_2'|)/2 \leq (\mathsf{max}V - u + u - \mathsf{min}V|)/2 = (\mathsf{max}V - \mathsf{min}V)/2,$$

which concludes the proof. □

Remark 2. We note that the output quality of a single invocation of AA as Protocol 1 can be improved when the adversary controls less computational power. This is in line with the classical AA algorithm by Dolev *et al.* [DLP+86].

The output quality ϵ can be improved by calling the same Approximate Agreement protocol sequentially, with each invocation using the output of the previous one as input, resulting in n sequential calls improving the output quality to $\epsilon' = \epsilon^n$. In the classical setting(s) (point-to-point channels, or PKI), since deterministic termination is guaranteed, the sequential composition of AA protocols is trivial. In a permissionless environment, however, where only a public setup is available, the sequential composition becomes more challenging in that a common reference string is needed for *every* invocation in order to avoid pre-mining. This problem has been addressed in [GKS24] for the case of synchronous protocols (i.e., no [unknown] bounded delay) with static number of participants. In Section 5, we show how this mechanism can be adapted to the drifting clock model with fluctuating number of parties.

4.2 Weak/Approximate Agreement with Optimal Corruption Threshold

Protocol 1 works only when the adversarial computational power is bounded by one-third, and in terms of AA the protocol offers concentration quality $\epsilon = 1/2$. In this section we present a new protocol that improves the corruption threshold to honest majority for both Weak and Approximate Agreement. Moreover, for Approximate Agreement, it achieves ϵ-Agreement for an arbitrarily small $\epsilon > 0$ after a single invocation.

We first provide some intuition on why Protocol 1 fails in an honest majority setting, taking Approximate Agreement as an example. Notice that with stronger adversarial computational power, the intersection of PoW message arrays held by an honest party at the decision phase becomes too small to account for two-thirds of its PoW message array. Thus, since the common subset is not large enough, the reduce and mid-point computations can no longer guarantee ϵ-Agreement for $\epsilon = 1/2$ and Validity at the same time. In more detail, when the adversarial messages (and inconsistent honest messages due to network delay) can account for more than one-third of the messages, parties need to trim more values from both sides of the ordered array; otherwise, Validity would not hold because the mid-point computation may take some adversarially proposed values as input. However, trimming more than one-third of the elements from both sides hurts ϵ-Agreement, as the inequalities in Lemma 2 no longer stand.

To circumvent the above situation, our solution is to categorize the different PoW messages based on their indices (in the m parallel procedures) where they succeed, and select one message in each procedure to form a message array $\mathbf{V}$ of fixed size of m values, thus providing a more refined approach to decide on an output value.

Protocol description. Here we show how to extend Protocol 1 to the honest majority setting. The protocol takes two additional parameters k and η, which are explained below, and runs m PoW-based Byzantine agreement (BA) procedures in parallel (cf. [GKL15])[7]. In each procedure, it builds a PoW-based blockchain (which at a high level follows the Nakamoto protocol) and binds the mining procedure of the chain with "input-blocks" using $2{\times}1$ PoWs (thus, each party maintains m parallel blockchains which we denote by $\mathbb{C}$). Specifically, for a RO query output u and each chain index $i \in [m]$, alongside with checking if a block that is used to extend the chain is produced by evaluating $[u]_{i\sim m} < T$, the $2{\times}1$ mining further evaluates the reverse of this chunk $[u]^{\mathsf{R}}_{i\sim m}$. If $[u]^{\mathsf{R}}_{i\sim m} < T$, an input-block $\mathtt{IB}$ containing the miner's input is mined and $\mathtt{IB}$ can be included in the corresponding i-th blockchain. Yet, a single RO query can produce different valid blocks to extend different blockchains, as well as an input block that can be valid on multiple chains. Refer to Protocol 2 Line 6 to 17 to see how the $m{\times}1$ and $2{\times}1$ mining procedures are bound together in order to get m independent parallel instances of the PoW-based BA protocol. Also note that, parties extend each blockchain $\mathcal{C}$ independent following the longest chain selection rule (as specified in the Bitcoin backbone protocol [GKL15] which we omit the details on chain validation and selection here), and keeps including valid and unique input-blocks with respect to $\mathcal{C}$.

After ℓ rounds, for each parallel chain $\mathcal{C}_i$, parties extract all input-blocks in its prefix by pruning blocks based on their timestamp, subject to the "common prefix" parameter k.[8] Then, these input-blocks are ordered based on their value contained and the median one is picked as the output of chain $\mathcal{C}_i$. This forms an array $\mathbf{V}$ of size m. For Approximate Agreement, each party then decides its output based on its local $\mathbf{V}$ and a parameter $\eta < m/2$, by first ordering $\mathbf{V}$ and removing the smallest and largest η values, selecting the first element of every η elements, and computing the average of the selected ones. I.e., they output $\mathsf{avg}(\mathsf{select}(\mathsf{reduce}(\mathsf{V}, \eta), \eta))$ and terminate. In terms of Weak Agreement, parties output a value that accounts for a super-majority of the chains if it exists, and $\perp$ otherwise.

[7]We note that, while running a single invocation of such BA procedure [GKL15] for $\Theta(\log^2 \kappa)$ rounds yields a full agreement, terminating after constantly many rounds does not give any form of weak agreement with overwhelming probability.

[8]In ordinary blockchains, after pruning k blocks, parties hold a chain that is a prefix of any other party's with overwhelming probability, hence the term "common prefix." However, since our protocol terminates in constant time, this pruning operation will lead to a common prefix only with constant probability.

Protocol $\frac{1}{2}$−$\mathsf{ApproxAgr}(\ell, m, T, k, \eta)$

```
▷ The following code is executed in each round r (val is the party input)
if $r \leq \ell$ then
    Fetch incoming chains $(\mathcal{C}_1, \ldots, \mathcal{C}_N)$
    $\mathbb{C}_{\mathsf{local}} \leftarrow \mathsf{UpdateLocalChain}(\mathbb{C}_{\mathsf{local}}, \mathcal{C}_1, \ldots, \mathcal{C}_N)$    ◁ Select longest valid chains for each index
    Fetch incoming input-blocks $(\mathtt{IB}_1, \ldots, \mathtt{IB}_k)$
    Add valid $\mathtt{IB}_1, \ldots, \mathtt{IB}_{k'}$ to buffer
    ▷ $m$×1 PoW mining
    $h \leftarrow \varepsilon$, $st \leftarrow \varepsilon$
    for $i = 1$ to $m$ do
        $h \leftarrow h \,\|\, [H(\text{head}(\mathbb{C}_i))]_{i\sim m}$
        Let $\mathtt{buffer}_i$ denote all IB that are valid yet not included w.r.t. $\mathbb{C}_i$
        $st \leftarrow st \,\|\, \mathsf{Blockify}(\mathtt{buffer}_i)$[a]
    end for
    $u \leftarrow H(ctr, r, h, st, val)$
    for $i = 1$ to $m$ do    ◁ Check if PoW succeeds on any type of block.
        if $[u]_{i\sim m} < T$ then set $\mathbb{C}_i \leftarrow \mathbb{C}_i \,\|\, \langle ctr, r, h, st, val\rangle$ and diffuse $\mathbb{C}_i$    ◁ Extend chain
        if $[u]^{\mathsf{R}}_{i\sim m} < T$ then diffuse $\mathtt{IB} = \langle ctr, r, h, st, val\rangle$
    end for
    $ctr \leftarrow ctr + 1$
else
    Initialize $\mathbf{V}$ to an empty array
    for $i$ from 1 to $m$ do    ◁ Extract output from parallel chains
        Initialize $\mathbf{M}$ to an empty array
        for $\mathtt{IB} \in \mathcal{B} \in \mathbb{C}_i^{\ell-k\rceil}$ do
            Parse IB as $\langle \cdot, \cdot, \cdot, \cdot, val\rangle$ and add $val$ to $\mathbf{M}$
        end for
        Sort $\mathbf{M}$ and add $\mathsf{med}(\mathbf{M})$ to $\mathbf{V}$
    end for
    ▷ Decide final output
    if deciding output for weak agreement then
        Order $\mathbf{V}$ non-decreasingly and denote $n \leftarrow |\mathbf{V}|$
        if $\mathbf{V}_\eta = \mathbf{V}_{n-\eta}$ then output $\mathbf{V}_\eta$ else output $\perp$
    else if deciding output for approximate agreement then
        Output $\mathsf{avg}(\mathsf{select}(\mathsf{reduce}(\mathbf{V}, \eta), \eta))$
    end if
end if
```

[a] Blockify translates a sequence of transactions to the ledger state (cf. [BMTZ17]).

Protocol 2: *Honest-majority Weak/Approxmate Agreeement.*

Next, in the following lemma we show that running the m parallel chains and terminating in a constant number of rounds in a bounded delay network yields good properties on a fraction of the chains. The two properties in Lemma 3 have been proven for a synchronous network in [GKS24]. Here we extend this result to the bounded-delay network setting.

Lemma 3. *There exist parameterizations of Protocol 2 such that the following holds. Let $\mathbb{C}$ and $\mathbb{C}'$ denote the parallel chains held by two honest parties P and P' at the end of round ℓ, respectively. There exists a subset $S \subseteq \{1, 2, \ldots, m\}$, $|S| > m - \eta$, such that for all $i \in S$, the following properties hold on chains $\mathcal{C} = \mathbb{C}_i$ and $\mathcal{C}' = \mathbb{C}'_i$:*

- **Agreement**: $\mathcal{C}^{\ell-k\rceil} = \mathcal{C}'^{\ell-k\rceil}$.
- **Honest input-block majority**: *More than half of the input-blocks included in $\mathcal{C}^{\ell-k\rceil}$ and $\mathcal{C}'^{\ell-k\rceil}$ are produced by honest parties.*

Proof (sketch). Consider the i-th single chain $\mathcal{C} = \mathbb{C}_i$ of a party P. There exist parameterizations such that at the end of round ℓ, with constant probability $p > (m-\eta)/m$, $\mathcal{C}^{\ell-k\rceil}$ yields a common view with any other honest party and includes a majority of honest input blocks. Proving the above claim is mainly a reminiscence of the proof of [GKS24, Theorem 2], with additionally extending it to the bounded-delay network (see, e.g., [GKL14, Section 7], for more details).

Since any two single chains are mutually independent from each other, the probability that these two properties hold on more than $p \cdot m > m - \eta$ chains can be computed by the Chernoff bound (Theorem 5), with error probability negligibly small in the security parameter κ (recall that $m = \mathsf{polylog}\kappa$). □

Given that the 'good' properties hold on a sufficiently large subset of the values in the output decision array, we conclude that Protocol 2 solves the Approximate Agreement problem with an arbitrarily small ϵ.

Theorem 3. *There exist protocol parameterizations such that Protocol 2 is an ϵ-secure Approximate Agreement protocol, for an arbitrarily small $\epsilon > 0$, against an adversary that controls less than half of the computational power, and all honest parties terminate at the end of round ℓ, except with probability negligible in κ.*

Proof. Let $\eta = [\epsilon/(1+2\epsilon)]m$ and consider the arrays of size m held by honest parties at the end of the execution. Due to Lemma 3, there exist protocol parameterizations such that all honest parties share a common subset of at least size $m - \eta$, and since the output is picked as the median among a set with a majority of honest messages, all values in this subset stay in the convex hull of the input. By following a similar argument as that in Lemma 2, it holds that for any (possibly the same) $\mathsf{P}_1, \ldots, \mathsf{P}_k \in \mathcal{H}$,

$$\mathsf{min}V \leq \mathbf{V}^{\mathsf{P}_1}_{\eta+1} \leq \mathbf{V}^{\mathsf{P}_2}_{2\eta+1} \leq \ldots \leq \mathbf{V}^{\mathsf{P}_k}_{m-\eta} \leq \mathsf{max}V.$$

Validity follows directly, and since all parties pick $1/\epsilon$ values after the select operation (Equation (1)), the output quality is ϵ. □

5 Permissionless and Self-Sufficient State Machine Replication with Fast Fairness

In this section, we present our permissionless state machine replication protocol Permissionless-SMR. We first present a novel parallel blockchain construction serving as a basic platform to run the "high-level" tasks (e.g., agreement and clock synchronization) in Section 5.1. In Section 5.2, in order for parties to synchronize their clocks periodically, we propose a new time adjustment algorithm (cf. Equation (5)) based on the honest-majority Approximate Agreement protocol in Section 4.2. We then, in Section 5.3, show how Byzantine agreement and state machine replication with fast fairness can be built "on top" of the parallel blockchains. A full description of protocol Permissionless-SMR is presented in Section 5.4.

5.1 A New Parallel Blockchain Construction

We describe how parallel blockchains can be run continuously for any (polynomially bounded) number of intervals, thus allowing Weak/Approximate Agreement protocols (and later clock synchronization and Byzantine agreement) to be sequentially invoked.

The PoW-based Approximate Agreement protocol presented in Section 4.2 terminates in constant time, and the security of an invocation relies on a high-entropy CRS to invalidate all RO queries made by corrupted parties before the activation of honest parties (a.k.a. pre-mining attack). Nonetheless, since CRS is available only at the beginning of the first invocation, sequentially running multiple invocations of Protocol 2 does not provide any security guarantee for the second and later invocations. More precisely, the naïve sequential composition of AA protocols, by keep extending m separate chains and let online parties periodically update their internal state based on the blocks on the tip of the chains in their local view, incurs the following problem: Since any interval might be "reverted" in the future as the adversary manages to create a longer fork and surpass the current one, parties that join after the beginning of an execution cannot synchronize with the online parties, as a large fraction of chains that online parties used to synchronize their clocks will get orphaned in the future.

To tackle this problem, we introduce a novel parallel blockchain construction, which we later show supports the following features: (i) it enables interval-based state update, thus parties update their internal state at the end of each interval; (ii) it has no dependency on a global clock and can self-synchronize all parties' local clocks as long as they proceed at some bounded rates; and (iii) it allows difficulty adjustment thus supports dynamic participation while preserving the optimal corruption resiliency.

The main difference between our parallel blockchain construction and that employed by Chain-King Consensus in [GKS24] is that, in [GKS24] it requires a chain to be "dense" such that it possesses sufficiently many blocks in any time window of fixed length in an interval, and each chain should point to sufficiently many dense chains in the previous interval (i.e., cross chain reference). Contrary, in our construction we completely eliminate the hardcoded density parameter and the cross chain references, hence switching from a "fragmented" chain structure to a "continuous" one.

Parallel blocktrees. Next, we elaborate on our construction. We extend the parallel chain structure to *parallel blocktrees* consisting of m independent blocktrees. Recall that, on a single chain, all blocks on the same height competes with each other and only the one on the longest chain survives; thus blocks in the current view might get discarded from the longest chain in the future. To help parties retrieve previously-longest yet now-orphaned forks, all blocks needs to be bookkeeped. In this sense, a single chain is extended to a *blocktree*. The root of this tree is the genesis block and each path corresponds to one fork. We note that this adaption is for future retrieval only, and parties still follow the heaviest (i.e., with most accumulated difficulty; longest in case of static participation) chain rule to select and process incoming chains. In other words, parties adopt the heaviest fork in a blocktree and try to extend that fork using PoW, but they also bookkeep all forks.

A parallel blocktree is simply a parallel repetition of m blocktrees whose mining procedure on the heaviest fork is bounded using $m\times 1$ PoW. See Figure 2 for an illustration. We also extend the blockchain notations to capture this modification. Specifically, we write $\mathcal{T}$ denoting a blocktree. I.e., a genesis block with no incoming edges and all other blocks are connected by an incoming edge from exactly one other block. We write $\mathcal{C} \in \mathcal{T}$ if $\mathcal{C}$ matches one path in $\mathcal{T}$. Regarding parallel chains, let $\mathbb{T}$ denote m parallel chains and we write $\mathbb{C} \in \mathbb{T}$ if $\forall \mathcal{C} \in \mathbb{C}$, it holds that $\mathcal{C} \in \mathcal{T}$ where $\mathcal{T}$ is the corresponding blocktree in $\mathbb{T}$.

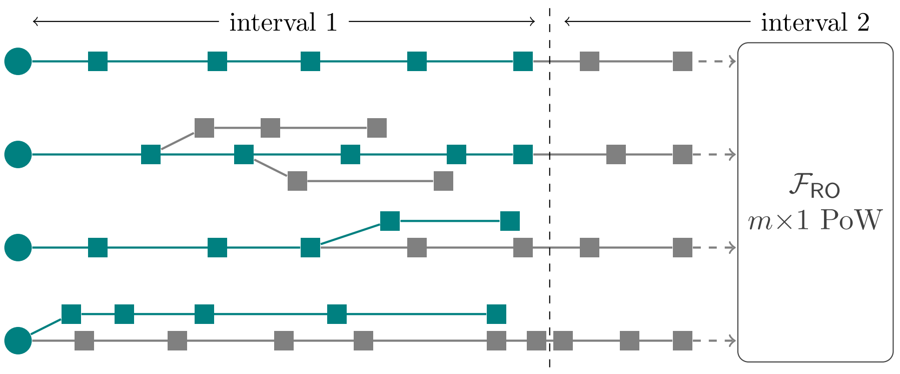

Figure 2: *An illustration of the parallel blocktrees and the mining procedure. Genesis blocks on each chain are represented as circles. Teal blocks and forks depicts the view of a party at the end of the first interval. There is no fork on the first chain; and on the second chain, forks are bookkeeped yet fail to revert the longest one. The last two blocks in the view of a party on the third chain are discarded from the longest chain by another longer fork. On the forth chain, all blocks in the first interval are reverted.*

Intervals and stages. We divide protocol time into consecutive, non-overlapping "intervals" in order to periodically let protocol participants adjust their local clocks and update internal states. An interval consists of R rounds. Parties decide which interval they are in based on their local time (thus different parties may stay in different intervals). An interval is further divided into three consecutive, non-overlapping stages — view convergence (VC), output generation (OG) and reference convergence (RC). We denote their duration by R_{VC}, R_{OG} and R_{RC} respectively. In the view convergence (VC) stage, parties extend their local parallel chains and wait for the second stage. Then, in the output generation (OG) stage, parties generate input-blocks which contain their suggested output value (cf. Section 5.3), local time (see Section 5.2) and also interval retrieval information (see this section below). They emit these input-blocks and record them on the blockchain. Regarding the last stage RC, parties wait for a (possibly) common view of the OG stage and they adjust their clocks and update their internal state at the end of this stage (which is also the end of an interval).

A new timestamping scheme. Next, we introduce our new timestamping scheme for protocols to work with drifting clocks. We extend the block timestamp from the conventional single round index $r \in \mathbb{N}^+$ to a pair of both interval index and round index, i.e., $\langle itvl, r\rangle \in \mathbb{N}^+ \times \mathbb{N}^+$. We say a block $\mathcal{B}$ on a chain $\mathcal{C}$ is in interval $itvl$, if it reports a timestamp $\langle itvl, \cdot\rangle$. A block $\mathcal{B}$ owns a valid timestamp iff. it satisfies the predicate $\mathsf{validTS}(r, itvl) \triangleq r \le itvl \cdot R$. Note that this allows for a chain in interval $itvl$ to start with blocks reporting timestamps smaller than $(itvl - 1) \cdot R$. Such adaption is necessary, since the clock adjustment algorithm might set clocks backwards; however, in the PoW setting parties must try to produce blocks in the next interval hence they should be able to mine with "retorted" timestamps.

Regarding blocks on the same chain (same fork on the blocktree), our protocol employs a new validation rule which asks for the monotonicity in the magnitude of stages. To be precise, let $\mathsf{stage} : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+ \times \{\mathsf{VC}, \mathsf{OG}, \mathsf{RC}\}$ be a function that takes a (valid) timestamp as input and outputs its corresponding stage. We define a canonical order on $(itvl, stage) \in \mathbb{N}^+ \times \{\mathsf{VC}, \mathsf{OG}, \mathsf{RC}\}$ as $(1, \mathsf{VC}), (1, \mathsf{OG}), (1, \mathsf{RC}), (2, \mathsf{VC}), (2, \mathsf{OG}), (2, \mathsf{RC}), \ldots$ so that operations like "$<, =, \le$" are defined canonically. A chain $\mathcal{C}$ owns valid timestamps iff. for any two blocks $\mathcal{B}, \mathcal{B}'$ on $\mathcal{C}$ with timestamps $\langle itvl, r\rangle$, $\langle itvl', r'\rangle$ respectively, $\mathcal{B}$ is an ancestor of $\mathcal{B}'$ implies $\mathsf{stage}(itvl, r) \le \mathsf{stage}(itvl', r')$.

In summary, our timestamp validation predicate, taking two consecutive blocks as input, is defined as follows.[9]

$$\mathsf{validOrder}(\langle itvl, r\rangle, \langle itvl', r'\rangle) \triangleq \left\{ \begin{array}{c} \mathsf{validTS}(itvl, r) \wedge \mathsf{validTS}(itvl', r') \\ \wedge\, \mathsf{stage}(itvl, r) \leq \mathsf{stage}(itvl', r') \end{array} \right\}. \tag{3}$$

Note that, our blockchain notation $\mathcal{C}^{t\rceil}$ provides a chain with all block timestamps less than t, which can be ill-defined with non-monotonically increasing timestamps. We stress that in our protocol we will only use this operation with a stage-boundary time (e.g., $\mathcal{C}^{R_{\mathsf{VC}}+R_{\mathsf{OG}}\rceil}$) to get chain segments up to the end of a stage, hence resulting in an unambiguous chain.

In addition, in our protocol parties defer the processing of a chain with future timestamps until their local clocks reach that stage. For instance, when the local clock of a party stays in the output generation stage of the second interval, it processes chains with blocks no later than that stage, but record all chains with blocks in the future stages for future processing.

We provide a high-level intuition behind this new timestamping scheme. Suppose we stick to the conventional monotonically-increasing timestamp scheme, in our protocol analysis, an isolated success which asserts the progress of a chain should be considered with a time period of $(\Delta + \Phi)$ rounds where $\Phi = \Theta(R)$ is the maximum skew during an interval. Nonetheless, as the duration of an interval is set as a linear function of the isolated success period, this implies the clock drift ρ can only be set inversely proportional to many other protocol parameters (details see our analysis). With our new timestamp scheme which does not ask for the monotonicity in stages, we eliminate such correlation between clock drift ρ and other protocol parameters.

Retrieving views from previous intervals. Recall that an interval lasts for a constant number of rounds; however the adversary can, with non-negligible probability, successfully prepare a private fork longer than the honest one that diverges up to poly-logarithmically many rounds (in κ), when the honest parties share a common view of, e.g., the first chain $\mathcal{C}$ as $\mathbb{C}_1$ at the end of the i-th interval, a block $\mathcal{B}$ on $\mathcal{C}$ might get discarded in the future and no longer stay on the longest chain in $\mathbb{C}_1$. (Note that $\mathcal{B}$ is still bookkeeped in the blocktree, i.e., $\mathcal{B} \in \mathcal{C}$ and $\mathcal{C} \in \mathbb{T}_1$.) Thus, conventionally, if a party P joins after the i-th interval (or, internal state of P gets flushed after the i-th interval), she has no way to realize that $\mathcal{B}$ is indeed in the common view on $\mathbb{C}_1$ at the end of the i-th interval. This observation, at first glance, seems prohibitive for newly joint parties to catch up the internal state of existing parties, as to bootstrap they should realize the common views of honest parties at the end of each interval.

To tackle this problem, we introduce a retrieval mechanism based on Weak Agreement that allows parties to *obliviously* agree on all forks that were in the common view of all honest parties at the end of each interval[10]. We write $\mathtt{snapshot}_{\mathsf{P}}[i]$ to denote the set of longest chains at the end of interval i in party P's local view. Note that since bad events on random oracles (e.g., two blocks share the same hash) happens with only negligible probability, $\mathtt{snapshot}_{\mathsf{P}}[i]$ can be initialized as a vector of m block hashes. P bookkeeps $\mathtt{snapshot}_{\mathsf{P}}[i]$ when her local clock reaches interval $i+1$.

During the $(i+1)$-th interval, parties include their local view of all longest chains at the end of interval i (i.e., $\mathtt{snapshot}_{\mathsf{P}}[i]$) in the input-block and emit and collect these input blocks in the output generation stage. To rule out pre-mining attacks, we require each input-block $\mathtt{IB}$ pointing to the last block in the view convergence stage (VC) on its corresponding chain, thus providing fresh randomness asserting that $\mathtt{IB}$ is not generated too early. Note that this procedure relies on parties

[9]A similar timestamping scheme with dual indices is proposed in [GKS22]. Yet their scheme still requires monotonically increasing timestamps within an interval while ours removes this restriction.

[10]Our retrieval mechanism provides no security guarantee for forks that honest parties do not share a unanimous view. Nevertheless, as we shall observe later, our protocol relies only on the oblivious agreement of honest common views.

being online at the end of the previous interval and preserve their internal state since then. Recall that we assume the total computational power of corrupted parties and honest parties that suffer from transient faults (e.g., temporarily lose access to network or the RO) account for less than the majority, this guarantees that on sufficiently many parallel chains, at least half of the input blocks report local views of alert parties in the previous interval (see Section 3.2 for more details on party classification).

We now explain how a fresh party $\mathsf{P}_{\mathsf{new}}$ (either newly joining the protocol, or suffering from transient internal faults) can learn the common view of intervals in the past. Our approach is simple and works in a "trace-back" manner: For $\mathsf{P}_{\mathsf{new}}$ to bootstrap the blockchain and retrieve all previous views of interval $1, 2, \ldots, itvl$ where $itvl$ is the current protocol interval, $\mathsf{P}_{\mathsf{new}}$ first listens to the protocol passively and build parallel blocktrees. $\mathsf{P}_{\mathsf{new}}$ then needs to decide which interval the protocol stays in (as she has no knowledge on the protocol time at this stage). Given our goal to bootstrap in constant time, this is a delicate task since the adversary can gain a sudden mining success spark on all parallel chains, thus producing heaviest forks with future block timestamps in the suffix to fool the joining parties. In order to eliminate such attack, after bootstrapping the blockchain, $\mathsf{P}_{\mathsf{new}}$ decides the interval by first pruning k_{bootstr} blocks on all chains and then adopt the *median* block timestamp on the tip of all chains[11].

Once $\mathsf{P}_{\mathsf{new}}$ is about to finish the i-th interval in her local view, $\mathsf{P}_{\mathsf{new}}$ performs the following operations recursively to retrieve the views from interval $i-1$ to 1. Precisely, in order to set the view of a previous interval, e.g., $\texttt{snapshot}_{\mathsf{P}}[i-1]$, $\mathsf{P}_{\mathsf{new}}$ tries to find block hashes in her local parallel blocktrees that has been referred to by sufficiently many input-blocks in the next interval. For the j-th chain, if in the i-th interval, there exists more than $3m/4$ chains such that the majority of the input-blocks report the same block reference chunk $[h']_{j\sim m}$ (where h' is a κ-bit string in input blocks to report the miner's local view of the previous interval) and a block with hash $[h']_{j\sim m}$ do exist in $\mathbb{T}_{\mathsf{local}}$, then the j-th chunk of $\texttt{snapshot}_{\mathsf{P}}[i-1]$ is set to $[h']_{j\sim m}$; otherwise (either no block hash is refereed by sufficiently many chains, or if no such block exists), it is set to $\perp$.

Note that Weak Agreement on parallel chains is not sufficient to tighten newly joint party's local clock. The decision on which interval $\mathsf{P}_{\mathsf{new}}$ is in only gives a coarse notion of time. We introduce the full bootstrapping protocol for new parties to synchronize with online ones in Section 5.2, using the retrieval mechanism in this section as a sub-routine. We also elaborate in Section 5.3 on how to bootstrap the ledger state when obliviously agreeing on a large fraction of parallel chains in all intervals.

Remark 3. Our usage of Weak Agreement to retrieve previous intervals is also adopted in the bootstrapping procedure of the distributed ledger protocol in [GKS24]. Nonetheless, we stress that we adapt this technique to our new parallel blockchain construction, which achieves bootstrapping with no knowledge of time and optimal corruption resiliency with dynamic participation, which cannot be achieved by the bootstrapping algorithm in [GKS24].

The difficulty adjustment algorithm. We adjust the mining difficulty on each chain independently, using a "reversed" version of Bitcoin's target recalculation function (cf. [GKS22]) which adjusts the mining target every M rounds[12].

In more detail, we divide time into consecutive, non-overlapping epochs and set the duration M of an epoch as a multiple of intervals so that the end of an epoch coincides with the end of an interval. Note that since our timestamping scheme asks for the explicit interval indices, the epoch

[11]Our analysis will show that, after pruning k_{bootstr} blocks, the adversary can only push forward or stall the block timestamps on a bounded fraction of chains. Hence, after adopting block timestamp that is median among all chains, $\mathsf{P}_{\mathsf{new}}$ learns a time that deviates from alert parties for a bounded amount of time.

[12]In Bitcoin, the mining difficulty is adjusted for every epoch of 2016 blocks.

that a block belongs to can be directly decided based on its timestamp; and on a single chain, the epoch indices of blocks increase monotonically. Epochs also apply to the blocktrees as every block on the tree must belong to a unique chain. Further, parties also maintain an internal variable $\mathtt{ep}$ indicating the epoch index that they stay in.

Let Λ_{epoch} denote denote the ideal number of blocks in an epoch. We have $\Lambda_{\mathsf{epoch}} = M \cdot f$, where f is the ideal block generation rate. For the first epoch ($\mathtt{ep} = 1$), parties adopt the same target as the genesis block (T_0). I.e., $T_1 = T_0$. Regarding the second and later epochs ($\mathtt{ep} > 1$), parties figure out how many blocks are produced in the previous epoch, and set the next target based on the previous one. This variation is proportional to the ratio of expected number of blocks Λ_{epoch} and their actual number. I.e., for epoch $\mathtt{ep} + 1$,

$$T_{\mathtt{ep}+1} \triangleq \mathsf{min}\Big\{\mathsf{max}\Big\{\frac{\Lambda_{\mathsf{epoch}}}{\Lambda} \cdot T_{\mathtt{ep}}, \frac{1}{\tau}\Big\}, \tau\Big\}, \tag{4}$$

where Λ is the number of blocks in epoch $\mathtt{ep}$, and τ is the bound on the relative amount of difficulty that can be adjusted each time.

In our analysis, we show that while the difficulty adjustment is performed on each chain independently, it can be set appropriately to stay close to the ideal block generation rate, thus accommodating the varying computational power.

5.2 The Clock Synchronization Procedure

Synchronization beacons. In the context of clock synchronization, we also call input-blocks as "synchronization beacons" (cf. [BGK$^+$21, GKS22]). When parties are in the output generation (OG) stage of an interval $\mathtt{itvl}$ (in their local view), they mine beacons with their local time $\langle \mathtt{itvl}, \mathtt{r} \rangle$ and include all beacons (which are valid with respect to interval $\mathtt{itvl}$) into their corresponding chains. Note that beacons should still convey inputs for building the state machine (refer to Section 5.3) as well as local views of the previous interval to help retrieve the views and secure the parallel blockchain (Section 5.1); for brevity we omit those details in this section.

Our shift calculation algorithm is based on comparing the timestamp recorded in beacons and their earliest local arrival time. In order to perform such computation, parties need to bookkeep the time that they receive fresh beacons (in case parties receive duplicate beacons, they consider the one that arrives earliest in their local view). Once a party P receives a beacon $\mathtt{IB}$ (either directly or by observing it on a chain) at local time $\langle \mathtt{itvl}, \mathtt{r} \rangle$, P bookkeeps its arrival time in a local beacon registry $\mathsf{arrivalTime}(\cdot)$ as an entry $(\mathtt{IB}, \langle \mathtt{itvl}, \mathtt{r} \rangle, \text{flag})$ where flag $\in \{\mathsf{temp}, \mathsf{final}\}$. When the beacon $\mathtt{IB}$ is valid with respect to a fork in the parallel blocktrees in the current or previous interval, P assigns final to the flag of $\mathtt{IB}$; otherwise, it assigns temp. Note that for a beacon $\mathtt{IB}$ labeled with temp, it will be removed from the registry when P enters the corresponding interval and P's local blockchain invalidates $\mathtt{IB}$ (invalid beacons are those that do not provide good fresh randomness with respect to any fork in party's local blocktrees). Since timestamp is part of the block header, a beacon $\mathtt{IB}$ can be a valid one on multiple chains; yet, it suffices to bookkeep only one entry in $\mathsf{arrivalTime}(\cdot)$ and it can be reused for the same beacon in multiple chains.

A beacon $\mathtt{IB}$ is said to be *valid* with respect to an interval *itvl* and a chain $\mathcal{C}$, if (i) $\mathtt{IB}$ reports a timestamp in the OG stage of interval *itvl*; (ii) $\mathtt{IB}$ points to the last block in the VC stage of interval *itvl* on $\mathcal{C}$; and (iii) $\mathtt{IB}$ is a valid PoW message w.r.t. *itvl* and $\mathcal{C}$. Refer to Algorithm 2 for a full description.

Shift calculation algorithm. We define $\mathsf{arrivalTime}$ as a function, taking a beacon $\mathtt{IB}$ as input, outputs its local arrival time $\langle \mathtt{itvl}, \mathtt{r} \rangle$; recall that TS is a function that outputs the timestamp recorded in a block $\mathcal{B}$, we slightly abuse this notation and let it take also a beacon as input. We write $\mathtt{IB} \in \mathcal{C}^{(i)}$ if a beacon $\mathtt{IB}$ is recorded in the output generation stage of i-th interval on a chain $\mathcal{C}$ and $\mathtt{IB}$ is valid with respect to $\mathcal{C}$. When the interval index is clear in the context, we drop the superscript and simply write $\mathtt{IB} \in \mathcal{C}$.

Parties compute a value shift at the end of an interval (i.e., when their local clock enters a round such that $\mathtt{r} = (\mathtt{itvl} - 1) \cdot R$), using beacons recorded in the current interval, and adds shift to its local time. Specifically, for a party P with local parallel chains $\mathbb{C}_{\mathsf{local}}$ at the end of interval $itvl$, its local shift $\mathsf{shift}_{\mathsf{P}}$ is computed as

$$\mathsf{shift}_{\mathsf{P}} \triangleq \mathsf{avg}\Big(\mathsf{select}\Big(\mathsf{reduce}\Big(\Big\{\mathsf{med}\Big\{\mathsf{TS}(\mathtt{IB}) - \mathsf{arrivalTime}(\mathtt{IB}) \;\Big|\; \mathtt{IB} \in \mathcal{C}\Big\} \;\Big|\; \mathcal{C} \in \mathbb{C}_{\mathsf{local}}\Big\}, \eta\Big), \eta\Big)\Big). \quad (5)$$

where $\eta < m/2$ is a protocol parameter, and select and reduce are as defined in Equation (1).

Roughly speaking, this algorithm can be viewed as first adjusting a party P's local clock separately on each single chains by computing the median shift (i.e., the difference between beacon time and its local receiving time) thus yielding m different clock times; then, the same decision procedure as that in Protocol 2 is applied on these clocks, by filtering the largest and smallest η ones, picking the first clock time every η remaining values, and then compute their average.

With a positive shift at the end of interval $\mathtt{itvl}$, party P skips shift rounds and forward her local clock to $\langle \mathtt{itvl} + 1, \mathtt{itvl} \cdot R + \mathsf{shift} \rangle$. Otherwise, her clock is set backward $|\mathsf{shift}|$ rounds and in the next round she tries to mine a block with timestamp $\langle \mathtt{itvl} + 1, \mathtt{itvl} \cdot R - |\mathsf{shift}| \rangle$ for the $(itvl + 1)$-th interval. Recall our timestamping scheme in Section 5.1, such "retorted" timestamp is allowed in the view convergence stage. Additionally, parties will update their beacon registry, by applying shift on the local receiving time of all future beacons.

We note that, with overwhelming probability, every shift value that honest parties compute at the end of each interval is well-bounded, so that their logical time stays in a good linear envelope of the nominal time, which guarantees accuracy. We also prove that, when parties start an interval with bounded skew $\Phi_{\mathsf{init}} = \Theta(\rho\Delta)$ and certain good properties on the parallel blockchains in this interval hold, after all honest parties enter the next interval, their local clocks are at most $\Phi = \Theta(\rho\Delta)$ apart from each other, which solves the synchronization problem. Looking ahead, our full analysis (Section 6) will show that these good properties hold throughout the entire execution and thus all intervals serve as good synchronizers.

The bootstrapping procedure. We now show how a fresh party, with total lack of knowledge other than the genesis block (CRS), can join the Permissionless-SMR protocol by passively observing the protocol execution for a *constant* number of rounds. We highlight that even without dynamic participation, this joining procedure is still of interest as parties that passively listening to the protocol can learn a precise time in the protocol and use it as a timestamping service. This procedure also allows for an existing protocol participant, whose internal state suddenly gets flushed, to catch-up with other honest parties.

Consider a newly joining party $\mathsf{P}_{\mathsf{new}}$. Given the interval retrieval mechanism introduced in Section 5.1, $\mathsf{P}_{\mathsf{new}}$ can bootstrap her local parallel blocktrees $\mathbb{T}_{\mathsf{local}}$, by listening to the protocols for a constant amount of time and updating the heaviest chain such that for each interval, $\mathsf{P}_{\mathsf{new}}$ obliviously learns a subset of forks that are in the common view of alert parties. Nonetheless, recall our timestamping scheme in Section 5.1, blockchains only provide a coarse notion of time such that the local clock of $\mathsf{P}_{\mathsf{new}}$ can deviate from alert ones for R_{VC} rounds. Our goal is to further tighten this skew to $\Phi = \Theta(\rho\Delta)$ — a small constant.

We now explain how the fresh party $\mathsf{P}_{\mathsf{new}}$ can synchronize with alert parties with well-bounded skews. Upon executing the bootstrapping procedure, $\mathsf{P}_{\mathsf{new}}$ resets her local clock to $\langle 1, 1 \rangle$. During the bootstrapping phase, $\mathsf{P}_{\mathsf{new}}$ also listens to the beacon diffusion network $\mathcal{F}^{\mathsf{input}}_{\mathsf{Diffuse}}$ and bookkeep the arrival time of all beacons. This phase lasts for more than two interval durations so that $\mathsf{P}_{\mathsf{new}}$ observes at least one complete output generation stage in an interval. Since $\mathsf{P}_{\mathsf{new}}$ is not synchronized, she cannot filter any invalid beacons so she temporarily mark all beacons as temp. Nonetheless, after she finishes bootstrapping her parallel blockchain, $\mathsf{P}_{\mathsf{new}}$ roughly understands which interval she stays in hence $\mathsf{P}_{\mathsf{new}}$ starts to remove invalid beacons based on her local heaviest chain. Note that our protocol guarantees that a large fraction of chains yield good properties at the end of an interval (e.g., common view of the chain in the output generation stage), $\mathsf{P}_{\mathsf{new}}$ thus shares the same beacon set (marked as final) with alert parties on sufficiently many chains. At the end of the bootstrapping procedure, by iterating all complete intervals that $\mathsf{P}_{\mathsf{new}}$ has observed and applying sequentially the shift calculation algorithm in Equation (5), $\mathsf{P}_{\mathsf{new}}$ synchronizes her clock with all alert parties. Note that when $\mathsf{P}_{\mathsf{new}}$ has observed two or more complete intervals, the local arrival time of all beacons in the second and later intervals needs to be updated each time after clock adjustment, by adding shift computed from the last interval to these beacons. The full description of the bootstrapping procedure is presented in Protocol 15.

An improved honest-majority lower bound for clock synchronization. We discuss under what circumstances honest majority is necessary for clock synchronization. Regarding corruption resiliency of synchronization problems, in the information-theoretic setting, Dolev *et al.* [DHS86] show that clock synchronization is impossible when more than one-third of the parties are corrupted. In the authenticated setting (assuming unforgeable signatures and a PKI), on one side, Srikanth and Toueg [ST87] show that honest majority is necessary when the resulting linear envelope on logical time is as good as that on nominal time. On the other side, Halpern *et al.* [HSSD84] presents a dishonest majority protocol however its logical time can deviate drastically from the real time as the number of corrupted parties grows.

We extend the argument for honest majority in [ST87] to a more general accuracy condition. In a nutshell, when physical clocks drifts in a $(1+\rho)$-linear envelope, we show that to stay in the time that is *strictly tighter* than a $(1+\rho)^2$-linear envelope of nominal time, a majority of honest parties (resources) is necessary. We state this result in Theorem 4, and a depiction on the logical linear envelope where honest majority is necessary can be found in Figure 3.

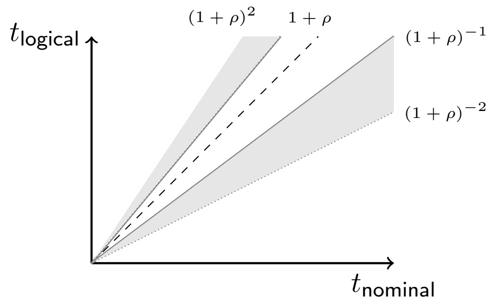

Figure 3: *An illustration of the logical time linear envelope (in gray) such that honest majority is necessary.*

Theorem 4. *Any clock synchronization protocol running by parties with ρ-linear-envelope physical clocks that achieves Φ-bounded skews and Γ-accuracy such that $\Phi \in \mathbb{N}$ and $\rho \leq \Gamma < 2\rho + \rho^2$ must have a majority of honest parties.*

Proof. Let $D_i(r)$ denote the clock of party P_i at nominal time r and $C_i(r)$ denote the (exported) logical time of party P_i at nominal time r.

Assume $0 < \epsilon \ll \rho$ and there exists a protocol that achieves $((1+\rho)^{-1}(1+\rho-\epsilon)^{-1}, (1+\rho)(1+\rho-\epsilon))$-accuracy with dishonest majority. We show that it is impossible by first considering a system with two parties P, P' and the following three executions.

- *Execution E_1.* Both the parties follow the protocol, and party P has the maximum clock speed (i.e., $D_1(r) = (1+\rho)r$); party P' has the minimum speed ($D'_1(r) = (1+\rho)^{-1}r$). Moreover, all messages have delay exactly $\Delta/(1+\rho)^2$.
- *Execution E_2.* Party P is honest and has rate $R_2(r) = (1+\rho)^{-1}r$; party P' is corrupted and runs at $R'_2(r) = (1+\rho)^{-3}r$ but is otherwise correct. Moreover, all messages have delay exactly Δ.
- *Execution E_3.* Party P' is honest and has rate $R'_3(r) = (1+\rho)r$; party P_1 is corrupted and runs at $R_3(r) = (1+\rho)^3 r$ but is otherwise correct. Moreover, all messages have delay exactly $\Delta/(1+\rho)^4$.

All these three executions follow the clock and network assumptions and are hence admissible. In addition, executions E_1 and E_2 are indistinguishable for party P; and executions E_1 and E_3 are indistinguishable for party P'. Since accuracy is achieved, P in execution E_1 will report a time $C_1(r) \le (1+\rho)(1+\rho-\epsilon)r + b_1$. Since $D_1(r) = (1+\rho)r$, we find that $C_1(r) \le (1+\rho-\epsilon)D_1(r) + b^1$. Similarly, by considering execution E_2 we have $(1+\rho-\epsilon)^{-1}C_2(r) + a_2 \le C_2(r)$. Since executions E_1 and E_2 are indistinguishable for party P, the relation between its exported time and reading time must be the same in both executions. Therefore, we have for $k = 1, 2$,

$$(1+\rho-\epsilon)^{-1}D_k(r) + a_2 \le C_k(r) \le (1+\rho-\epsilon)D_k(r) + b_1.$$

I.e., in execution E_1 we can find a time τ_1 such that for all $r > \tau_1$,

$$(1+\rho-\epsilon)^{-1}(1+\rho)r + a_2 \le C_1(r) \le (1+\rho-\epsilon)(1+\rho)r + b_1.$$

Similarly, by considering party P' in executions E_1 and E_3, we see that there exist a time τ_2 in execution E_1 such that for all $r > \tau_2$,

$$(1+\rho-\epsilon)^{-1}(1+\rho)^{-1}r + a_1 \le C'_1(r) \le (1+\rho-\epsilon)(1+\rho)^{-1}r + b_3.$$

Hence, we find that there exist a time $\tau \ge \mathsf{max}\{\tau_1, \tau_2\}$ in execution E_1 such that for all $r > \tau$, the deviation between the exported time of two correct parties is greater than a constant Φ, which violates the bounded skew condition. $\square$

5.3 The New State Machine Replication Protocol

The parallel blockchain framework that we have introduced, complemented with the clock synchronization mechanism, allow a varying number of parties to reach weak agreement sequentially at the end of each interval. Based on this, we build permissionless Byzantine Agreement and state machine replication protocols.

Our approach in this section is based on the constructions in [GKS24], which provide a generic way of building BA that terminates in expected-constant time, and SMR that settles all incoming transactions in expected-constant time on top of parallel chains. Their underlying parallel blockchains, however, are not truly permissionless, operate in a synchronous network and rely on a global clock, shortcomings that we overcome in this paper.

Permissionless BA in expected-constant time. We run "Chain-King Consensus" as introduced in [GKS24] on top of our parallel blockchain. At a high level, this protocol emulates the classical "Phase-King Consensus" [BG89, BGP89] with randomized king election [FG03, KK06] in the following way.

We map the i-th parallel chain to the i-th party in a classical randomized Phase-King protocol[13]. For each interval, parties might share possibly different views on the output of a single chain, which emulates that parties share possibly different views on the message sent by a party in the classical protocol. If a chain is good (i.e., honest parties share common views and the majority of the input blocks are honestly generated) then it emulates an honest party since its output is unanimous and validity holds. Otherwise, a chain emulates a corrupted party as its output might differ among parties or validity might not hold. The protocol runs in iterations by checking if parties start with the same input. If this is not the case and parties are "confused" about the output, then they adopt the output of the king chain (set *deterministically* as the first chain) as their new input (this mimics the oblivious king election in classical protocols). Since the duration of intervals are of constant rounds, and the king chain yields a good chain with constant probability, the protocol terminates in expected-constant time.

In more detail, consider a BA invocation where a party P starts with an input v. Party P maintains three internal variables: (i) the suggestion of output `val` which is initialized as v and can be modified at the end of each interval; (ii) a bit variable `lock` that indicates whether parties should take the output of king chain; (iii) a bit variable `decide` that indicates whether a party P should decide on the current `val` as output[14]. The protocol proceeds in iterations where each iterations consists of three intervals. In the first interval, parties will set both their `decide` and `lock` to `true` if they have seen more than $3m/4$ chains output the same value; and in the second interval, they set only `lock` to `true` if they have seen more than $3m/4$ chains output the same value. In both the first and second intervals, if the majority of chains output the same value, then parties will switch their `val` to that output. In the third interval, parties update their internal state based on `decide` and `lock`. A party will replace `val` with the output of the first (king) chain if both `decide` and `lock` remains `false`; and it will terminate if `decide` is `true`; when `decide` remains `false` however `lock` is set to `true`, it resets `lock` back to `false`.

A detailed description of the core state update algorithm in Chain-King Consensus is presented in Protocol 11.

Permissionless and self-sufficient SMR with fast fairness. Given that an invocation of permissionless BA can be run on top of our parallel chain framework, the translation to permissionless SMR works so long as (i) there are means to coordinate the second and later BA invocations that parties may start at different intervals; and (ii) an honest input is selected as output with *constant* probability which guarantees liveness in expected-constant time.

At a high level, our approach here is reminiscent of that in [GKS24]. In order to coordinate parties that may start at different times, we employ (i) the Bracha-style termination [Bra84] to let parties terminate in neighboring intervals at the end of a BA invocation; and (ii) super-interval expansion (which mimics the classical super-round expansion in round-preserving sequential composition of BA [CCGZ19]) to allow parties expand normal interval to super-interval that consists of multiple intervals so that even if parties start non-simultaneous they still reach weak agreement on the output of this super interval.

[13]A subtle difference compared with classical protocols is that in Chain-King Consensus, the set of chains that emulates an honest party keeps changing, while in classical protocols the set of honest parties remains the same (yet unknown). The protocol still works in that all state-update operations are based on counting the number of messages and has nothing to do with party identifiers.

[14]Note that due to non-simultaneous termination, in [GKS24], for one-shot consensus, there is a fourth "exit" parameter indicating whether parties should drop from extending the blockchains. For simplicity, we avoid this part in our description and refer to [GKS24] for more details.

Note that since parties in SMR use a sequence of transactions as their BA input, they are almost unlikely to start unanimously with the same set of transactions (as we will show soon, if parties also want to add the coinbase transaction, then they will never invoke BA with the same input). In order to prevent the adversary from dominating the output all the time (due to the weak validity property), we proceed as follows. Recall that Chain-King Consensus runs in iterations and each iteration consists of three intervals, when parties are in the first and second interval of an iteration, they decide the output of each chain as follows: A party P outputs v for the i-th chain if the majority of input-blocks reports v; otherwise P outputs $\perp$ (in this way, the adversary cannot let parties decide on a batch of transactions that is not an honest input in the first two stages). When P is in the third interval and about to switch her input, she replaces her BA input with the input-block of smallest hash value in the king chain.

We now show how parties can build an ever-growing log with fast fairness. Once a block (a sequence of transactions from the BA output) is appended to the log in SMR, it sets up a public key pk such that the probability that pk belongs to an honest party is roughly proportional to the fraction of her computational among all protocol participants. Upon joining the protocol, a party P generates her own public-key pair $(\mathsf{pk}, \mathsf{sk})$ as her identifier (pseudonym). Recall that our SMR is built on the sequential composition of BA invocations, and in each invocation parties prepare their own input using all unsettled transactions in their buffer. We additionally introduce the following mechanism.

At the onset of a BA invocation, P prepares a special transaction $\mathtt{tx}^{\text{base-tx}}$ that contains her public key pk. P then appends all valid, unsettled transactions in the buffer after $\mathtt{tx}^{\text{base-tx}}(\mathsf{pk})$ as her BA input. Moreover, we add an additional interval at the beginning of BA such that at the end of this interval, parties listen to the king chain and switch their $\mathtt{val}$ to the block with minimum hash. As a result, the output of each BA in our SMR protocol is a sequence of transactions, with the first transaction being coinbase of public key pk.

Next, we provide some intuition on how the above mechanism helps to achieve fast fairness (refer to Section 6 for more details). We focus on the first interval which serves as a "lottery" letting parties replace their input with the block with minimum hash. Suppose parties share a common view on the king chain until the end of that interval, if an honest party manages to produce an input-block with the minimum hash among all parties, this implies that she makes a RO query such that the RO response yields minimum among all hash queries made that can contribute to the king chain (that is, the output generation stage of that interval). Also note that an input-block with minimum hash will be a valid block on the king chain (since it is definitely smaller than the mining target T). Hence, the probability that a party wins the lottery is the same as winning the minimum block hash on king chain, and is (roughly) proportional to her computational power among all participants. Even though the adversary can gain some bounded advantage by slightly cheating on the block timestamp, yet, since the honest parties share a common view of the king chain, the adversary can only keep including more input-blocks with bogus timestamps for a bounded amount of time. In case the king chain is a bad chain, all bets are off. Nonetheless, our analysis shows that the protocol can be parameterized so that the probability that the king chain is a bad chain is an arbitrarily small constant. Combining these two bounded, constant errors together, we can conclude that whenever a new batch of transactions is to be appended to the SMR log, the probability a party gets selected is proportional to her relative computational power.

5.4 Full Protocol Description

Main protocol instance. We introduce the main Permissionless-SMR protocol instance that dispatches to the relevant subprocesses following the UC notions.

Protocol $\mathsf{Permissionless\text{-}SMR}(\mathsf{P}, \mathrm{sid}; \mathcal{G}_{\mathsf{DClock}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{F}_{\mathsf{CRS}})$

Global Variables:

- Read-only: R, M, t_{off}, t_{gather}
- Read-write: `localTime`, `ep`, $\mathbb{C}_{\mathsf{local}}$, $\mathbb{T}_{\mathsf{local}}$, T^{ep}, `isInit`, t_{work}, `buffer`, `IBBuffer`, `futureChains`, `isSync`, `fetchCompleted`, $\mathsf{arrivalTime}(\cdot)$, `state`, `snapshot`

Registration / Deregistration:

- Upon receiving input $(\text{REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\mathsf{ledger}}, \mathcal{G}_{\mathsf{DClock}}\}$ execute protocol $\mathsf{Registration}(\mathsf{P}, \mathrm{sid}, \mathtt{reg}, \mathcal{R})$.
- Upon receiving input $(\text{DE-REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\mathsf{ledger}}, \mathcal{G}_{\mathsf{DClock}}\}$ execute protocol $\mathsf{Deregistration}(\mathsf{P}, \mathrm{sid}, \mathtt{reg}, \mathcal{R})$.
- Upon receiving input $(\text{IS-REGISTERED}, \mathrm{sid})$ return $(\text{REGISTER}, \mathrm{sid}, 1)$ if the local registry `Reg` indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}_{\mathsf{ledger}}$ (and did not de-register since then). Otherwise, return $(\text{REGISTER}, \mathrm{sid}, 0)$.

Interacting with the Ledger: Upon receiving a ledger-specific input $I \in \{(\text{SUBMIT}, \ldots), (\text{READ}, \ldots), (\text{MAINTAIN-LEDGER}, \ldots)\}$ verify first that all resources are available. If not all resources are available, then ignore the input; else (i.e., the party is operational and time-aware) execute one of the following steps depending on the input I:

- **If** $I = (\text{SUBMIT}, \mathrm{sid}, \mathtt{tx})$ **then** set $\mathtt{buffer} \leftarrow \mathtt{buffer} \,\|\, \mathtt{tx}$, and send $(\text{DIFFUSE}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \mathrm{sid}, \mathrm{minerID})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathsf{P}, \mathrm{sid})$; **if** $\mathsf{LedgerMaintenance}$ halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\text{READ}, \mathrm{sid})$ then invoke protocol $\mathsf{ReadState}(\mathsf{P}, \mathrm{sid})$.
- **If** $I = (\text{EXPORT-TIME}, \mathrm{sid})$ then do the following: if `isSync` or `isInit` is false, then return $(\text{EXPORT-TIME}, \mathrm{sid}, \bot)$; otherwise call $\mathsf{UpdateLocalTime}(\mathsf{P}, \mathrm{sid})$ and return $(\text{EXPORT-TIME}, \mathrm{sid}, \mathtt{localTime})$ to the caller.

Handling calls to the shared setup:

- Upon receiving $(\text{CLOCK-TICK}, \mathrm{sid}_C)$, forward it to $\mathcal{G}_{\mathsf{DClock}}$ and output $\mathcal{G}_{\mathsf{DClock}}$'s response.
- Upon receiving $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$, record that a CLOCK-UPDATE was received in the current round. If the party is registered to all its setups, then do nothing further. Otherwise, do the following operations before concluding this round:
  1 If this instance is currently time-aware but otherwise stalled or offline, then call $\mathsf{UpdateLocalTime}(\mathsf{P}, \mathrm{sid})$ to update `localTime`. If the party has passed a synchronization slot, then set $\mathtt{isSync} \leftarrow \mathsf{false}$.
  2 If this instance is only stalled but $\mathtt{isSync} = \mathsf{true}$, then additionally execute $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$, extract all new input blocks (synchronization beacons) IB from the fetched chains and record their arrival times and set $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$. Also, any unfinished interruptible execution of this round is marked as completed.
  3 Forward $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$ to finally conclude the round.

Protocol 3: *The main protocol instance of* $\mathsf{Permissionless\text{-}SMR}$.

Registration / de-registration. In order to perform basic operations, a party P needs to register to all resources. Note that the protocol will initialize local time $\mathsf{P}.\mathtt{localTime}$ to $\langle 1, 1 \rangle$, and P is aware whether he is not synchronized not and will set the bit variable `isSync` correspondingly.

Protocol $\mathsf{Registration}(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{G})$

```
 1: if $\mathcal{G} = \mathcal{G}_{\mathsf{DClock}}$ then send $(\textsc{register}, \mathrm{sid})$ to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and
    output the valued received by $\mathcal{G}$.
 2: if $\mathcal{G} = \mathcal{G}_{\mathsf{ledger}}$ then
 3:     if $\mathsf{P}$ is not registered with $\mathcal{G}_{\mathsf{DClock}}$ or already registered with all setups then
 4:         ignore this input
 5:     else
 6:         Send $(\textsc{clock-tick}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$ and receive $(\textsc{clock-tick}, \mathrm{sid}_C, \mathrm{tick})$
 7:         Send $(\textsc{register}, \mathrm{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}$
 8:         $\mathtt{localTime} \leftarrow \langle 1, 1\rangle$ and $\mathtt{isSync} \leftarrow \mathsf{false}$
 9:         If this is the first registration invocation for this ITI, then set $\mathtt{isInit} \leftarrow \mathsf{false}$.
10:         Output $(\textsc{register}, \mathrm{sid}, \mathsf{P})$ once completing the registration with all the above resources $\mathcal{F}$
11:     end if
12: end if
```

Protocol 4: *Parties register on necessary resources (functionalities) to run the protocol.*

The deregistration process is an analogous action that sets variables to the initial values.

Protocol $\mathsf{Deregistration}(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{G})$

```
1: if $\mathcal{G} = \mathcal{G}_{\mathsf{DClock}}$ then
2:     Set $\mathtt{isSync} \leftarrow \mathsf{false}$
3:     Send $(\textsc{de-register}, \mathrm{sid})$ to $\mathcal{G}$ and set registration status as de-registered with $\mathcal{G}$
4:     Output the valued received by $\mathcal{G}$
5: end if
6: if $\mathcal{G} = \mathcal{G}_{\mathsf{ledger}}$ then
7:     Set $\mathtt{isSync} \leftarrow \mathsf{false}$
8:     Send $(\textsc{de-register}, \mathrm{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}$, set its registration status as de-registered with $\mathcal{F}_{\mathsf{Diffuse}}$ and
   output $(\textsc{de-register}, \mathrm{sid}, \mathsf{P})$.
9: end if
```

Protocol 5: *Parties de-register from corresponding resources.*

Ledger maintenance. The protocol LedgerMaintenance groups all the steps regarding the main ledger operation. Note that, depending on a party is alert or not, she might execute different sub protocols. For parties that are not synchronized, after querying $\mathcal{F}_{\mathsf{CRS}}$, they first enter the bootstrapping mode by calling JoiningProcedure. By executing this sub protocol, they set their internal state `isSync` to true and then start to execute the normal ledger maintenance operations.

Protocol $\mathsf{LedgerMaintenance}(\mathsf{P}, \mathrm{sid})$

```
   ▷ The following steps are executed in an $(\textsc{maintain-ledger}, \mathrm{sid}, \mathrm{minerID})$-interruptible manner:
1: if $\mathtt{isInit} = \mathsf{false}$ then
2:     Send ($\textsc{Retrieve}$, sid) to $\mathcal{F}_{\mathsf{CRS}}$ and set response as $\mathsf{CRS}$ and set $\mathtt{isInit} \leftarrow \mathsf{true}$.
3: end if
   ▷ Bootstrap if not synchronized.
```

```
4: if not isSync then Call $\mathsf{JoiningProcedure}(\mathsf{P}, \mathrm{sid})$        ◁ Protocol 15
   ▷ Normal operations when alert.
5: Invoke $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_N)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$
6: $\mathtt{buffer} \leftarrow \mathtt{buffer} \,\|\, (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and $\mathtt{futureChains} \leftarrow \mathtt{futureChains} \cup \{\mathcal{C}_1, \ldots, \mathcal{C}_N\}$
7: Call $\mathsf{UpdateLocalTime}(\mathsf{P}, \mathrm{sid})$        ◁ Protocol 8
   ▷ Ensures the processing of new input-blocks arrived in chains only.
8: Extract input-blocks $IB \leftarrow \{\mathtt{IB}_1, \ldots, \mathtt{IB}_n\}$ contained in $\mathcal{C}_1, \ldots, \mathcal{C}_N$ and not yet contained in IBBuffer.
9: Call $\mathsf{ProcessInputBlocks}(P, \mathrm{sid}, IB)$        ◁ Protocol 9
10: Let $\mathcal{N}_0$ be a set of (single) chains s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \mathcal{C} \in \mathbb{C} \in \mathtt{futureChains} \wedge \forall \mathcal{B} \in \mathcal{C} : \mathsf{TS}(\mathcal{B}) \leq \mathtt{localTime}$
11: Remove each $\mathcal{C} \in \mathcal{N}_0$ from futureChains
12: fetchCompleted ← true
13: Call $\mathsf{UpdateLocalChain}(\mathbb{C}_{\mathsf{local}}, \mathbb{T}_{\mathsf{local}}, \mathcal{N}_0)$ to update $\mathbb{C}_{\mathsf{local}}$ and $\mathbb{T}_{\mathsf{local}}$
14: if $t_{\mathsf{work}} < \mathtt{localTime}$ then
15:     Call $\mathsf{MiningProcedure}(\mathsf{P}, \mathrm{sid}, r)$        ◁ Protocol 10
16:     if $\mathtt{r} = \mathtt{itvl} \cdot R$ then
17:         Call $\mathsf{StateUpdate}(\mathsf{P}, \mathrm{sid})$        ◁ Protocol 12
18:         Call $\mathsf{SyncProcedure}(\mathsf{P}, \mathrm{sid})$        ◁ Protocol 13
19:     end if
20:     Set $t_{\mathsf{work}} \leftarrow \mathtt{localTime}$
21: end if
22: Call $\mathsf{FinishRound}(\mathsf{P})$        ◁ Mark normal round actions as finished
```

Protocol 6: *The main operations for parties to maintain the ledger.*

Fetch information. Parties fetch block information from $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$ to learn new parallel chains (precisely, blocktrees) with possibly future timestamps. Note that in order to simplify the chain validation and selection procedure, we let FetchInformation return all single chains $\mathcal{C}_1, \ldots, \mathcal{C}_N$ extracted from all received blocktrees[15]. Parties also fetch transactions from $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ to learn new transactions.

Protocol $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$

```
1: if fetchCompleted then return        ◁ Fetch once per round and never catch up missed round.
   ▷ Fetch blocks on $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$
2: Send (FETCH, sid) to $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$; denote the response by (FETCH, sid, $bc$).
3: Extract blocktrees $\mathcal{T}_1, \ldots \mathcal{T}_n$ from $bc$
4: Extract chains $\mathcal{C}_1, \ldots \mathcal{C}_N$ from $\mathcal{T}_1, \ldots \mathcal{T}_n$
   ▷ Fetch transactions on $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$
5: Send (FETCH, sid) to $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$; denote the response by (FETCH, sid, $tx$).
6: Extract transactions $(\mathtt{tx}_1, \ldots \mathtt{tx}_k)$ from $tx$
7: if not isSync or P is stalled then
8:     $\mathtt{buffer} \leftarrow \mathtt{buffer} \,\|\, (\mathtt{tx}_1, \ldots \mathtt{tx}_k)$
```

[15]Parties associate all blocktrees/chains with their index $i \in [m]$ in parallel chains(trees), and thus apply chain validation and validation for chains with the same index. For brevity, we ignore these details in our protocol description and assume all chains are processed correspondingly.

```
 9:     futureChains ← futureChains ∪ {$\mathbb{C}_1, \dots \mathbb{C}_N$}
10: end if
OUTPUT: The protocol outputs ($\mathcal{C}_1, \dots, \mathcal{C}_N$) and ($\mathtt{tx}_1, \dots, \mathtt{tx}_k$) to its caller (but not to $\mathcal{Z}$).
```

Protocol 7: *Fetching new blocks and transactions from the diffusion functionality.*

Chain and input block validation. We present the chain and input block validation procedure in Permissionless-SMR respectively. Note that, different from the single chain protocols where the validity of each chain can be attested independently, chains in Permissionless-SMR need to be verified per interval (see the chain selection procedure in Algorithm 4). Hence the algorithm IsValidChain takes as input a chain $\mathcal{C}$, an integer i the index of $\mathcal{C}$ in parallel chains, a parallel blocktree $\mathbb{T}$ that $\mathcal{C}$ is associate with and an integer $itvl$ the target interval on $\mathcal{C}$ to be verified.

The blocks (input-blocks) in Permissionless-SMR are of form

$$\mathcal{B} = \langle ctr, \langle itvl, r\rangle, h, st, h^*, h', val\rangle,$$

where ctr is the nonce for PoW, $\langle itvl, r\rangle$ shows the block timestamp, h is the hash pointer to the previous block, st represents the block content (note that h and st are meaningful only for a block $\mathcal{B}$), h^* is the fresh randomness when mining input-blocks, and h', val is the chain reference for previous intervals (cf. Section 5.1) and input-block content respectively.

The following functions help us simplify the validation process:

- We use ValidBlock to verify if a block is a successful PoW on the i-th chain (that is, the nonce ctr is valid and the block hash — i-th segment of the RO output is less than target T).

$$\mathsf{ValidBlock}(\mathcal{B}, i, T) \triangleq [H(\mathcal{B})]_{i\sim m} < T \wedge \mathcal{B}.ctr < 2^{32}.$$

- Analogously, we use use ValidInputBlock to verify if a synchronization beacon is a successful PoW on its associated chain by checking the reverse of the string segment. I.e.,

$$\mathsf{ValidInputBlock}(\mathtt{IB}, i, T) = [H(\mathtt{IB})]^{\mathsf{R}}_{i\sim m} < T \wedge \mathtt{IB}.ctr < 2^{32}$$

- We adopt isStageOG as a predicate to justify if a timestamp r is in the output generation stage of interval $itvl$. Precisely,

$$\mathsf{isStageOG}(itvl, r) \triangleq r \in [(itvl - 1) \cdot R + R_{\mathsf{VC}}, itvl \cdot R - R_{\mathsf{RC}}].$$

Given the above, Algorithm 1 presents a full specification of the chain validation procedure.

Algorithm IsValidChain($\mathcal{C}, i, \mathbb{T}, itvl$)

```
   ▷ This algorithm has five internal Boolean variables goodHash, goodNonce, goodTime, goodCRS
   and goodInputBlock, all initialized as true.
1: if $\mathcal{C}$ starts with a block with hash reference other than CRS then goodCRS ← false
2: if isSync and $\exists \mathcal{B} \in \mathcal{C}$ s.t. validOrder(TS($\mathcal{B}$), localTime) = false then goodTime ← false
   ▷ Derive fresh randomness for $itvl$ as indicated by $\mathcal{C}$
3: Extract $h^*_{itvl}$ from head($\mathcal{C}^{\lceil (itvl-1)\cdot R + R_{\mathsf{VC}} \rceil}$)
```

```
 4: for each block $\mathcal{B}$ in $\mathcal{C}$ from interval $itvl$ do
 5:     Parse $\mathcal{B}$ as $\langle ctr, \langle itvl, r\rangle, h, st, \cdot, \cdot, \cdot\rangle$
 6:     Set $T_{\mathcal{B}} \leftarrow \mathsf{ExtractMiningTarget}(\mathcal{C}, \lceil itvl/(M/R)\rceil)$.    ◁ Extract mining difficulty set for $\mathcal{B}$
        ▷ Check hash
 7:     Let $\mathcal{B}^{-1}$ be the last block in $\mathcal{C}$ before $\mathcal{B}$
 8:     Set $\mathsf{goodHash} \leftarrow ([h]_{i\sim m} = [H(\mathcal{B}^{-1})]_{i\sim m})$
        ▷ Check nonce
 9:     Set $\mathsf{goodNonce} \leftarrow \mathsf{ValidBlock}(\mathcal{B}, i, T_{\mathcal{B}})$
        ▷ Check input blocks
10:     if $\neg\mathsf{isStageOG}(itvl, r)$ and $\exists \mathtt{IB} \in \mathcal{B}$ then
11:         Set $\mathsf{goodInputBlock} \leftarrow \mathsf{false}$    ◁ Block should include IB only in OG
12:     else
13:         for each input block $\mathtt{IB} \in \mathcal{B}$ do
14:             Parse IB as $\langle ctr', \langle itvl', r'\rangle, \cdot, \cdot, h^*_{\mathtt{IB}}, \cdot, \cdot\rangle$
15:             if $\mathsf{isStageOG}(itvl, r) = \mathsf{false}$ or $\mathsf{TS}(\mathtt{IB}) > \mathsf{TS}(\mathcal{B})$ then $\mathsf{goodInputBlock} \leftarrow \mathsf{false}$
16:             if $\mathcal{C}$ contains duplicate IB or $[h^*_{\mathtt{IB}}]_{i\sim m} \neq [h^*_{itvl}]_{i\sim m}$ then $\mathsf{goodInputBlock} \leftarrow \mathsf{false}$
17:             Set $T_{\mathtt{IB}} \leftarrow \mathsf{ExtractMiningTarget}(\mathcal{C}, \lceil itvl'/(M/R)\rceil)$.
18:             Set $\mathsf{goodInputBlock} \leftarrow \mathsf{ValidInputBlock}(\mathtt{IB}, i, T_{\mathtt{IB}})$
19:         end for
20:     end if
21: end for
22: if $\mathsf{goodHash} \wedge \mathsf{goodNonce} \wedge \mathsf{goodInputBlock} \wedge \mathsf{goodTime} \wedge \mathsf{goodCRS}$ then return true
23: return false
```

Algorithm 1: *The chain validation procedure.*

Regarding input blocks that have not yet been included in chains, the validity of an IB with respect to a chain index i is checked only when the party's local time has forwarded to at least the same interval as IB (if not, this procedure returns "undecided" which defers the validity check).

Algorithm $\mathsf{IsValidInputBlock}(\mathtt{IB}, \mathcal{C}, i, \mathbb{T})$

```
    ▷ Precondition: Chain $\mathcal{C}$ is valid. Returns true if the beacon is a valid beacon w.r.t. $\mathcal{C}$, undecided
    if no judgement is possible, and false if the beacon is invalid w.r.t. $\mathcal{C}$.
1: Parse IB as $\langle ctr, \langle itvl, r\rangle, \cdot, \cdot, h^*_{\mathtt{IB}}, \cdot, \cdot\rangle$
2: if $\mathcal{C}$ contains no block in interval $itvl$ then return undecided    ◁ no judgement possible
    ▷ Check nonce value and freshness
3: Extract $h^*_{itvl}$ from $\mathsf{head}(\mathcal{C}^{\lceil (itvl-1)\cdot R + R_{\mathsf{VC}}\rceil})$
4: Set $T_{\mathtt{IB}} \leftarrow \mathsf{ExtractMiningTarget}(\mathcal{C}, \lceil itvl/(M/R)\rceil)$.
5: if $\mathsf{ValidInputBlock}(\mathtt{IB}, i, T_{\mathtt{IB}}) = \mathsf{true}$ and $[h^*_{\mathtt{IB}}]_{i\sim m} \neq [h^*_{itvl}]_{i\sim m}$ then return true
6: return false
```

Algorithm 2: *The input-block validation procedure.*

Extract mining target. We adopt algorithm $\mathsf{ExtractMiningTarget}$ to calculate the mining difficulty set for an epoch index ep on a single chain $\mathcal{C}$. Starting from the initial target T_0, this algorithm iteratively calculates targets from the first epoch to the targeted one, based on the number of blocks in each epoch and Equation (4).

Algorithm $\mathsf{ExtractMiningTarget}(\mathcal{C}, ep)$

```
1: if $ep = 1$ then return $T_0$                                    ◁ Return initial target
2: for $i$ from 2 to $ep$ do
3:     $\Lambda_i = |\{\mathcal{B} \mid \mathcal{B} \in \mathcal{C} \wedge \lceil \mathsf{TS}(\mathcal{B})/(M/R) \rceil = i\}|$
4:     $T_i = \min\{\max\{\Lambda_{\mathsf{epoch}}/\Lambda_i \cdot T_{i-1}, 1/\tau\}, \tau\}$          ◁ Equation (4)
5: end for
6: return $T_{ep}$
```

Algorithm 3: *Extracting the mining target for a given epoch.*

Update local time. Parties will send CLOCK-TICK to $\mathcal{G}_{\mathsf{DClock}}$ to check if it receives a tick = 0, which indicates the beginning of a new (local) round.

Protocol $\mathsf{UpdateLocalTime}(\mathsf{P}, \mathrm{sid})$

```
   ▷ Precondition: Only executed if time-aware.
1: Send (CLOCK-TICK, sid) to $\mathcal{G}_{\mathsf{DClock}}$ and receive (CLOCK-TICK, sid, tick)
2: if tick = 0 then
3:     localTime ← localTime + 1
4:     fetchCompleted ← false
5: end if
6: ep ← $\lceil$itvl$/(M/R)\rceil$
OUTPUT: The protocol outputs localTime, ep to its caller (but not to $\mathcal{Z}$)
```

Protocol 8: *Parties update their local clock due to $\mathcal{G}_{\mathsf{DClock}}$.*

Process input blocks and arrival times. The following procedure processes incoming input blocks, bookkeeps their arrival times and filters out duplicate ones. The predicate to verify input-block validity is presented in Algorithm 2. Regarding the duplicate beacons, only one with the earliest arrival time will be preserved.

Protocol $\mathsf{ProcessInputBlocks}(\mathsf{P}, \mathrm{sid}, IB)$

```
1: if fetchCompleted = true then return
2: Send (FETCH, sid) to $\mathcal{F}^{\mathsf{input}}_{\mathsf{Diffuse}}$ and denote the response by (FETCH, sid, $b$)
3: Extract all received input blocks ($\mathtt{IB}_1, \ldots, \mathtt{IB}_k$) contained in $b \cup IB$.
4: for each $\mathtt{IB}_i$ with arrivalTime(IB) = ⊥ do
5:     IBBuffer ← IBBuffer ∪ {IB}
6:     Parse TS(IB) as $\langle itvl', r' \rangle$
7:     if isSync ∧ (itvl ≥ $itvl'$) then
8:         Set arrivalTime($\mathtt{IB}_i$) ← (localTime, final)          ◁ The measurement is final.
9:     else                                               ◁ Will be adjusted upon next time shift.
10:        arrivalTime($\mathtt{IB}_i$) ← (localTime, temp)
11:    end if
12: end for
    ▷ Buffer cleaning.
```

```
13: if isSync then
14:     for each IB $\in$ IBBuffer do
15:         if TS(IB) has timestamp later than itvl then skip IB
16:         goodInputBlock $\leftarrow$ false
17:         for $i$ from 1 to $m$ do
18:             if $\exists \mathcal{C} \in \mathbb{T}_{\mathsf{local}}$ s.t. IsValidInputBlock(IB, $\mathcal{C}, i, \mathbb{T}_{\mathsf{local}}$) = true then goodInputBlock $\leftarrow$ true
19:         end for
20:         if goodInputBlock = false then Remove IB from IBBuffer
21:     end for
22: end if
```

Output: The protocol outputs ok to its caller (but not to $\mathcal{Z}$).

Protocol 9: *Parties filter invalid input blocks.*

Chain selection. Our chain selection rule (which should only be used by time-aware parties) works in two steps. First, it filters all invalid chains, by verifying them in an interval-by-interval fashion. I.e., chains are split into intervals and the $(i+1)$-th interval is checked after the i-th interval of all chains has been checked. An invalid chain $\mathcal{C}^*$ is removed out of the chain buffer once any interval of $\mathcal{C}^*$ fails the IsValidChain verification. And all valid chains are added to $\mathbb{T}$ as a fork (we do this interval-by-interval as well). After filtering all invalid chains, the next step runs the chain selection procedure for every chain index in the current interval, and replaces each chain in $\mathbb{C}$ with the longest fork in the corresponding tree in $\mathbb{T}$.

Algorithm UpdateLocalChain($\mathbb{C}, \mathbb{T}, \mathcal{N}_0 = \{\mathcal{C}_1, \ldots, \mathcal{C}_N\}$)

```
    ▷ This algorithm should only be called by fully-synchronized parties.
1:  for $itvl$ from 1 to itvl do
2:      for each $\mathcal{C} \in \mathcal{N}_0$ do
3:          Let $i$ be the chain index of $\mathcal{C}$
4:          if IsValidChain($\mathcal{C}, i, \mathbb{T}, itvl$) = true then
5:              Add $\mathcal{C}^{itvl \cdot R \rceil}$ to the $i$-th tree in $\mathbb{T}$
6:          else                                          ◁ invalid chain from $itvl$
7:              Remove $\mathcal{C}$ from $\mathcal{N}_0$
8:          end if
9:      end for
10: end for
11: for $i$ from 1 to $m$ do
12:     Set $\mathcal{C}_{\max} \leftarrow \mathbb{C}_i$
13:     for each $\mathcal{C} \in \mathbb{T}_i$ do
14:         if diff($\mathcal{C}$) > diff($\mathcal{C}_{\max}$) then $\mathcal{C}_{\max} \leftarrow \mathcal{C}$
15:     end for
16:     Replace the $i$-th chain in $\mathbb{C}$ by $\mathcal{C}_{\max}$
17: end for
18: return $\mathbb{C}, \mathbb{T}$
```

Algorithm 4: *Parties filter invalid chains and select the heaviest one.*

Mining procedure. Once a party P has prepared all information and updated its state, she can run the core mining procedure in Protocol 10. When `localTime` reports in the output generation phase, P will include the fresh input blocks and check if she succeeds in the input-block mining procedure. At the onset of a BA invocation (when internal variable `val` is set to $\perp$), P starts to build her own input, starting from a coinbase transaction $\mathtt{tx}_{\mathsf{P}}^{\text{base-tx}}$ that contains her public key pk and she also signs this transaction.

Note that, for simplicity, we adopt two functions Blockify and ValidTx (cf. [BMTZ17]) that translates a sequence of transactions to the ledger state and verifies an incoming transaction w.r.t. a ledger state respectively, which we omit the details.

Protocol MiningProcedure(P, sid)

```
    ▷ The following steps are executed in an (maintain-ledger, sid, minerID)-interruptible manner:
    $h \leftarrow \varepsilon$                                              ◁ Prepare chain head
    for $i$ from 1 to $m$ do
        Parse the last block of $i$-th chain in $\mathbb{C}_{\mathsf{local}}$ as $\mathcal{B}$        ◁ Possibly genesis block (CRS)
        $h \leftarrow h \,\|\, [H(\mathcal{B})]_{i\sim m}$
    end for
    if isStageOG(localTime) then
        Set $h^* \leftarrow \varepsilon$ and $st \leftarrow \varepsilon$
        ▷ Prepare fresh randomness
        for $i$ from 1 to $m$ do
            Parse the last block in the VC stage of $i$-th chain in $\mathbb{C}_{\mathsf{local}}$ as $\mathcal{B}$
            $h^* \leftarrow h^* \,\|\, [H(\mathcal{B})]_{i\sim m}$
        end for
        ▷ Prepare block content
        $\mathbf{N} \leftarrow \varepsilon$
        for $i$ from 1 to $m$ do
            Parse the $i$-th chain in $\mathbb{C}_{\mathsf{local}}$ as $\mathcal{C}$
            Set $IB \leftarrow \{\mathtt{IB}' \in \mathtt{IBBuffer} \mid \mathsf{IsValidInputBlock}(\mathtt{IB}', \mathcal{C}, i, \mathbb{T}_{\mathsf{local}}) = \mathsf{true}\}$
            Set $IB' \leftarrow \{\mathtt{IB}' \in IB \mid \mathtt{IB}' \in \mathcal{B}' \in \mathcal{C}\}$
            $\mathbf{N} \leftarrow \mathbf{N} \,\|\, (IB \backslash IB')$
        end for
        $st \leftarrow \mathsf{blockify}(\mathbf{N})$
    else
        Set $h^* \leftarrow 0^\kappa$ and $st \leftarrow 0^\kappa$
    end if
    ▷ Prepare chain reference $h'$
    Set $h' \leftarrow \mathtt{snapshot}[\mathtt{itvl} - 1]$
    ▷ Prepare input val
    if $\mathtt{val} = \perp$ then
        Set $\mathtt{buffer}' \leftarrow \mathtt{buffer}$, $\vec{N} \leftarrow \mathtt{tx}_{\mathsf{P}}^{\text{base-tx}}(\mathsf{pk})$, and $\mathtt{val} \leftarrow \mathsf{Blockify}(\vec{N})$
        repeat
            Parse $\mathtt{buffer}'$ as sequence $(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
            for i from 1 to $n$ do
                if $\mathsf{ValidTx}(\mathtt{tx}_i, \mathtt{state} \,\|\, \mathtt{val}) = 1$ then
                    Set $\vec{N} \leftarrow \vec{N} \,\|\, \mathtt{tx}_i$
                    Remove $\mathtt{tx}_i$ from $\mathtt{buffer}'$
                    Set $\mathtt{val} \leftarrow \mathsf{Blockify}(\vec{N})$
                end if
            end for
```

```
35:     until $\vec{N}$ does not increase any more
36: end if
    ▷ $m\times 1$ PoW mining
37: $u \leftarrow H(ctr, \texttt{localTime}, h, st, h^*, h', \texttt{val})$
38: Set newChain, newIB as false
39: for $i = 1$ to $m$ do
40:     Set $T_i \leftarrow \mathsf{ExtractMiningTarget}(\mathcal{C}, \mathsf{ep})$ where $\mathcal{C}$ is the $i$-th chain in $\mathbb{C}_{\mathsf{local}}$
41:     if $[u]_{i\sim m} < T_i$ then                                ◁ Extend $i$-th chain
42:         Set $\mathcal{B} \leftarrow \langle ctr, \texttt{localTime}, h, st, h^*, h', \texttt{val}\rangle$ and newChain ← true
43:         Append $\mathcal{B}$ to the $i$-th chain of $\mathbb{C}_{\mathsf{local}}$ and $\mathbb{T}_{\mathsf{local}}$
44:     end if
45:     if isStageOG(localTime) and $[u]^{\mathsf{R}}_{i\sim m} < T_i$ then
46:         Set IB $\leftarrow \langle ctr, \texttt{localTime}, h, st, h^*, h', \texttt{val}\rangle$ and newIB ← true
47:     end if
48: end for
    ▷ Diffuse the extended chain and wait
49: if newChain = true then send (DIFFUSE, sid, $\mathbb{T}_{\mathsf{local}}$) to $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$ and set anchor here.[a]
50: if newIB = true then Send (DIFFUSE, sid, IB) to $\mathcal{F}^{\mathsf{input}}_{\mathsf{Diffuse}}$ and set anchor here.
51: Set $ctr \leftarrow ctr + 1$, give up activation and set anchor here.
```

[a]Upon next activation of this procedure, it resumes from the anchor set last time.

Protocol 10: *The mining procedure of m parallel blocks/input-blocks.*

Interval output algorithm. The following algorithm, on input parallel chains $\mathbb{C}$ and a target interval $itvl$, outputs a triple $(\{val_i\}_{i\in[m]}, \{ref_i\}_{i\in[m]}, king)$. The i-th element in the first vector val_i is the output of the i-th chain in interval $itvl$ (i.e., the majority value of all input-blocks); and the i-th element in the second vector ref_i is the reference to i-th chain in the previous interval (possibly being $\perp$); and $king$ is a single value extracted from the input block with minimum hash from the first chain.

Note that, each ref_i is extracted by observing an invocation of weak agreement over the parallel chains in this interval. I.e., we run m weak agreement protocol instances in parallel, where each one follows Protocol 2 in Section 4.2.

Algorithm ExtractIntervalOutput($\mathbb{C}, itvl$)

```
1: Initialize $\{val_i\}_{i\in[m]}, \{ref_i\}_{i\in[m]}$ and $king$
2: Initialize $m$ empty vectors $\{\mathbf{R}_i\}_{i\in[m]}$
3: for $i$ from 1 to $m$ do
4:     Initialize an empty vector $\mathbf{V}$ and $m$ empty vectors $\{\mathbf{R}'_i\}_{i\in[m]}$
5:     for each IB $\in \{\mathcal{B} \mid \mathcal{B} \in \mathbb{C}_i \wedge \mathsf{isStageOG}(\mathcal{B})\}$ do
6:         Parse IB as $\langle \cdot, \cdot, \cdot, \cdot, \cdot, h', val\rangle$
7:         Append $val$ to $\mathbf{V}$
8:         For $i$ from 1 to $m$ do Append $[h']_{i\sim m}$ to $\mathbf{R}'_i$
9:     end for
10:    if $v'$ accounts for majority in $\mathbf{V}$ then $val_i \leftarrow v'$
11:    For $i$ from 1 to $m$ do Sort $\mathbf{R}'_i$ non-decreasingly then append $\mathsf{med}(\mathbf{R}'_i)$ to $\mathbf{R}_i$
```

```
12: end for
13: for $i$ from 1 to $m$ do
14:     if $\exists v$ that account for more than $3m/4$ elements in $\mathbf{R}_i$ then
15:         Set $ref_i \leftarrow v$
16:     else
17:         Set $ref_i \leftarrow \bot$
18:     end if
19: end for
20: Choose $\mathtt{IB}^*$ s.t. $H(\mathtt{IB}^*) = \mathsf{min}\{H(\mathtt{IB}) \mid \mathtt{IB} \in \mathcal{B} \in \mathbb{C}_1 \land \mathsf{isStageOG}(\mathcal{B})\}$
21: Set $king \leftarrow \mathtt{IB}^*.val$
22: return $(\{val_i\}_{i\in[m]}, \{ref_i\}_{i\in[m]}, king)$
```

Algorithm 5: *Parties extract the output of a given interval.*

State update procedure. We first present the basic state update algorithm in Chain-King Consensus.

Protocol $\mathsf{ChainKingUpdateState}(\mathsf{P}, \mathrm{sid})$

```
    ▷ This algorithm is called once in each interval.
 1: Set $(\{val_i\}_{i\in[m]}, \cdot, king) \leftarrow \mathsf{ExtractIntervalOutput}(\mathbb{C}_{\mathsf{local}}, \mathtt{itvl})$
 2: Let $v$ denote the most frequent element in $\{val_i\}_{i\in[m]}$ and $c$ its frequency
 3: if $\mathtt{itvl} \mod 3 = 1$ then
 4:     if $c > m/2$ then set $\mathtt{val} \leftarrow v$
 5:     if $c > 3m/4$ then set $\mathtt{decide} \leftarrow \mathsf{true}, \mathtt{lock} \leftarrow \mathsf{true}$
 6: else if $\mathtt{itvl} \mod 3 = 2$ then
 7:     if $c > m/2$ then set $\mathtt{val} \leftarrow v$
 8:     if $c > 3m/4$ then set $\mathtt{lock} \leftarrow \mathsf{true}$
 9: else
10:     if $\mathtt{lock} = \mathsf{false}$ then set $\mathtt{val} \leftarrow king$
11:     if $\mathtt{decide} = \mathsf{true}$ then set $\mathtt{state} \leftarrow \mathtt{state} \parallel \mathtt{val}$ and $\mathtt{val} \leftarrow \bot$
12:     if $\mathtt{decide} = \mathsf{false}$ and $\mathtt{lock} = \mathsf{true}$ then set $\mathtt{lock} \leftarrow \mathsf{false}$
13: end if
```

Protocol 11: *The state update procedure for Chain-King Consensus.*

Note that in our SMR protocol, when a party P is about to finish local interval $\mathtt{itvl}$, she first stores her local view of the chains in this interval into $\mathtt{snapshot}$ and then run Chain-King Consensus with super-interval expansion and an additional lottery stage at the beginning which we detail below (cf. Section 5.3). We omit the details on super-phase expansion and use *itvl* to denote the super-interval index in a BA invocation.

Protocol $\mathsf{UpdateState}(\mathsf{P}, \mathrm{sid})$

```
    ▷ This algorithm is called once in each interval.
    ▷ Bookkeep local view of current interval
```

```
1: Set $\texttt{snapshot}[\texttt{itvl}] \leftarrow \varepsilon$
2: for $i$ from 1 to $m$ do
3:     Parse the $i$-th chain in $\mathbb{C}_{\mathsf{local}}$ as $\mathcal{C}$
4:     Parse hash of last block on $\mathcal{C}^{\lceil \texttt{itvl} \cdot R - R_{\mathsf{RC}} \rceil}$ as $h$
5:     $\texttt{snapshot}[\texttt{itvl}] \leftarrow \texttt{snapshot}[\texttt{itvl}] \parallel [h]_{i \sim m}$
6: end for
   ▷ Update internal ledger states
7: Let $\mathbf{V} = (v_1, \ldots, v_n)$ denote the output of $m$ chains respectively in super-interval $itvl$ and $king$
   the minimum hash from the first chain
8: Let $v$ denote the most frequent element in $\mathbf{V}$ and $c$ its frequency
9: if $\texttt{itvl} \mod 3 = 1$ then                                         ◁ Lottery first.
10:     if $\texttt{lock} = \mathsf{false}$ then set $\texttt{val} \leftarrow king$
11:     if $\texttt{decide} = \mathsf{true}$ then set $\leftarrow \texttt{state} \parallel \texttt{val}$ and $\texttt{val} \leftarrow \perp$
12:     if $\texttt{decide} = \mathsf{false}$ and $\texttt{lock} = \mathsf{true}$ then set $\texttt{lock} \leftarrow \mathsf{false}$
13: else if $\texttt{itvl} \mod 3 = 2$ then
14:     if $c > m/2$ then set $\texttt{val} \leftarrow v$
15:     if $c > 3m/4$ then set $\texttt{decide} \leftarrow \mathsf{true}, \texttt{lock} \leftarrow \mathsf{true}$
16: else
17:     if $c > m/2$ then set $\texttt{val} \leftarrow v$
18:     if $c > 3m/4$ then set $\texttt{lock} \leftarrow \mathsf{true}$
19: end if
```

Protocol 12: *The state update procedure for* Permissionless-SMR *with fast fairness.*

Synchronization procedure. Parties call SyncProcedure when their local clock enters the last round in an interval and adjusts their clock by computing shift based on their local parallel chains. Note that the "retorted" timestamps are marked explicitly with the next interval index. Thus, for each interval, this procedure is called only once.

Protocol SyncProcedure(P, sid)

```
   ▷ Only called when: P is alert, $\texttt{localTime} = \langle \texttt{itvl}, \texttt{itvl} \cdot R \rangle$ and $\texttt{itvl} > 0$
1: Initialize $\{\texttt{clockShift}_i\}_{i \in [m]}$ as an empty vector
2: for $i = 1$ to $m$ do
3:     Set $\mathcal{C}$ as $i$-th chain in $\mathbb{C}_{\mathsf{local}}$
4:     $B \leftarrow \{\mathcal{B} \mid (\mathcal{B} \in \mathcal{C}) \wedge \mathsf{isStageOG}(\mathsf{TS}(\mathcal{B})) = \mathsf{true}\}$
5:     $IB \leftarrow \{\texttt{IB} \mid (\texttt{IB} \in \mathcal{B} \in B) \wedge (\mathsf{TS}(\texttt{IB}) = \langle \texttt{itvl}, \cdot \rangle)\}$
       ▷ Find representative beacon and compute recommendation.
6:     for each $\texttt{IB} \in IB$ do
7:         Find unique $\texttt{IB}' \in \texttt{IBBuffer}$ s.t. $\texttt{IB}' = \texttt{IB}$. If inexistent, set $\texttt{IB}' \leftarrow \perp$.
8:         if $\texttt{IB}' \neq \perp$ then
9:             Set $\mathsf{arrivalTime}(\texttt{IB}) \leftarrow \mathsf{arrivalTime}(\texttt{IB}')$
10:            $\mathsf{recom}(\texttt{IB}, i) \leftarrow \mathsf{TS}(\texttt{IB}) - \mathsf{arrivalTime}(\texttt{IB})$
11:        else
12:            $IB \leftarrow IB \setminus \{\texttt{IB}\}$
13:        end if
14:    end for
15:    $\texttt{clockShift}_i \leftarrow \mathsf{med}\{\mathsf{recom}(\texttt{IB}, i) \mid \texttt{IB} \in IB\}$
```

```
16: end for
    ▷ Compute interval shift using Equation (5)
17: $\mathsf{shift}_{\mathtt{itvl}} \leftarrow \mathsf{avg}(\mathsf{select}(\mathsf{reduce}(\mathtt{clockShift}, \eta), \eta))$
    ▷ Update beacon registry
18: for each IB with $\mathsf{arrivalTime}(\mathtt{IB}) = (a, \mathsf{temp})$ do
19:     $\mathsf{arrivalTime}(\mathtt{IB}) \leftarrow (a + \mathsf{shift}_{\mathtt{itvl}}, \mathsf{final})$
20: end for
    ▷ Update local time
21: Set $\mathtt{localTime} \leftarrow (\mathtt{itvl} + 1, \mathtt{r} + \mathsf{shift}_{\mathtt{itvl}})$
22: Let $\mathcal{N}_0$ be the subsequence of futureChains s.t. $\mathcal{C} \in N_0 :\Leftrightarrow \forall \mathcal{B} \in \mathcal{C} : \mathsf{TS}(\mathcal{B}) < \mathtt{localTime}$
23: Remove each $\mathcal{C} \in \mathcal{N}_0$ from futureChains
24: Call $\mathsf{updateLocalChain}(\mathbb{C}_{\mathsf{local}}, \mathbb{T}_{\mathsf{local}}, \mathcal{N}_0)$ to update $\mathbb{C}_{\mathsf{local}}$ and $\mathbb{T}_{\mathsf{local}}$
25: Send $(\text{DIFFUSE}, \text{sid}, \mathbb{T}_{\mathsf{local}})$ to $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$ and proceed from here upon next activation of this procedure
OUTPUT: The protocol outputs ok to its caller (but not to $\mathcal{Z}$).
```

Protocol 13: *Parties update their local clocks.*

Finishing a round. Once a party P has done its actions in a round, P claims finishing current round by calling FinishRound and sending CLOCK-UPDATE to $\mathcal{G}_{\mathsf{DClock}}$.

Protocol FinishRound(P, sid)

```
1: while A (CLOCK-UPDATE, $\mathcal{Z}$) has not been received during the current round do
2:     Give up activation (set the anchor here)
3: end while
4: Send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{DClock}}$.
```

Protocol 14: *Finishing a round.*

The joining procedure. Honest yet un-synchronized parties run the JoiningProcedure to synchronize their internal state (i.e., their local clock and blockchain state). Parties run this procedure for constantly many rounds, by passively listen to the protocol execution, bootstrap the blockchain, keep track of the input-block local arrival time, and then adjust their local clock based on these information.

Protocol JoiningProcedure(P, sid)

```
   ▷ Phase A: state-reset
1: Call UpdateLocalTime(P, sid)                                  ◁ Align with newest round
2: if $\mathtt{localTime} > \langle 1, 1 \rangle$ then
3:     $\mathtt{localTime} \leftarrow \langle 1, 1 \rangle$
4:     $\mathtt{fetchCompleted} \leftarrow \mathsf{false}, \mathtt{futureChains} \leftarrow \emptyset, \mathtt{IBBuffer} \leftarrow \emptyset, \mathtt{buffer} \leftarrow \emptyset$
5:     Set input-block arrival timetable as empty array
6: end if
   ▷ Phase B: chain-convergence
```

```
7: while $\texttt{localTime} < \langle 1,1\rangle + t_{\mathsf{off}}$ do
8:     if $\texttt{fetchCompleted} = \mathsf{false}$ then
9:         Call $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote the fetched chains by $\mathcal{N} = (\mathcal{C}_1, \ldots, \mathcal{C}_N)$
10:        Call $\mathsf{UpdateLocalChain}(\mathbb{C}_{\mathsf{local}}, \mathbb{T}_{\mathsf{local}}, \mathcal{N})$ to update $\mathbb{C}_{\mathsf{local}}$ and $\mathbb{T}_{\mathsf{local}}$
11:        $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
12:        Call $\mathsf{FinishRound}(\mathsf{P}, \mathrm{sid})$
13:    end if
14:    Call $\mathsf{UpdateLocalTime}(\mathsf{P}, \mathrm{sid})$ to update $\texttt{localTime}$
15: end while
    ▷ Phase C: input-block gathering
16: while $\texttt{localTime} < \langle 1,1\rangle + t_{\mathsf{off}} + t_{\mathsf{gather}}$ do
17:     if $\texttt{fetchCompleted} = \mathsf{false}$ then
18:         Call $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote output by $(\mathcal{C}_1, \ldots, \mathcal{C}_N)$, $(\texttt{tx}_1, \ldots, \texttt{tx}_k)$
19:         Set $\texttt{buffer} \leftarrow \texttt{buffer} \,\|\, (\texttt{tx}_1, \ldots, \texttt{tx}_k)$
20:         Set $\texttt{futureChains} \leftarrow \texttt{futureChains} \,\|\, (\mathcal{C}_1, \ldots, \mathcal{C}_N)$
21:         Call $\mathsf{ProcessInputBlocks}(\mathsf{P}, \mathrm{sid}, \emptyset)$ and mark all arrival time with $\mathsf{temp}$
22:         Call $\mathsf{UpdateLocalChain}(\mathbb{C}_{\mathsf{local}}, \mathbb{T}_{\mathsf{local}}, \texttt{futureChains})$ to update $\mathbb{C}_{\mathsf{local}}$ and $\mathbb{T}_{\mathsf{local}}$
23:         Let $\mathbb{C}$ denotel the paralle chains after pruning $k_{\mathsf{bootstr}}$ blocks on all chains in $\mathbb{C}_{\mathsf{local}}$
24:         Set $\langle itvl^*, \cdot\rangle \leftarrow \mathsf{med}\{\mathsf{TS}(\mathcal{B}) \mid \mathcal{B}$ is the tip block of $\mathcal{C} \wedge \mathcal{C} \in \mathbb{C}\}$
25:         if $\texttt{snapshot}[itvl^*] = \perp$ then set $\texttt{snapshot}[itvl^*]$ using Protocol 12 Line 1 to 6
26:         $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
27:         Call $\mathsf{FinishRound}(\mathsf{P}, \mathrm{sid})$
28:     end if
29:     Call $\mathsf{UpdateLocalTime}(\mathsf{P}, \mathrm{sid})$ to update $\texttt{localTime}$
30: end while
    ▷ Phase D: synchronization and state-update
31: Set $i$ as the second minimum positive integer s.t. $\texttt{snapshot}[i] \neq \perp$
32: while $\texttt{snapshot}[i] \neq \perp$ do
33:     Initialize $\{\texttt{clockShift}_j\}_{j \in [m]}$ as an empty vector
34:     for $j$ from 1 to $m$ do
35:         Set $\mathcal{C}$ as $j$-th chain in $\mathbb{C}_{\mathsf{local}}$
36:         $B \leftarrow \{\mathcal{B} \mid (\mathcal{B} \in \mathcal{C}) \wedge \mathsf{isStageOG}(\mathsf{TS}(\mathcal{B})) = \mathsf{true}\}$
37:         $IB \leftarrow \{\texttt{IB} \mid (\texttt{IB} \in \mathcal{B} \in B) \wedge (\mathsf{TS}(\texttt{IB}) = \langle i, \cdot\rangle)\}$
            ▷ Find representative beacon and compute recommendation.
38:         for each $\texttt{IB} \in IB$ do
39:             Find unique $\texttt{IB}' \in \texttt{IBBuffer}$ s.t. $\texttt{IB}' = \texttt{IB}$. If inexistent, set $\texttt{IB}' \leftarrow \perp$.
40:             if $\texttt{IB}' \neq \perp$ then
41:                 Set $\mathsf{arrivalTime}(\texttt{IB}) \leftarrow \mathsf{arrivalTime}(\texttt{IB}')$
42:                 $\mathsf{recom}(\texttt{IB}, j) \leftarrow \mathsf{TS}(\texttt{IB}) - \mathsf{arrivalTime}(\texttt{IB})$
43:             else
44:                 $IB \leftarrow IB \setminus \{\texttt{IB}\}$
45:             end if
46:         end for
47:         $\texttt{clockShift}_i \leftarrow \mathsf{med}\{\mathsf{recom}(\texttt{IB}, j) \mid \texttt{IB} \in IB\}$
48:     end for
        ▷ Compute interval shift using Equation (5)
49:     $\mathsf{shift}_i \leftarrow \mathsf{avg}(\mathsf{select}(\mathsf{reduce}(\texttt{clockShift})))$
50:     for each $\texttt{IB}$ with $\mathsf{arrivalTime}(\texttt{IB}) = (a, \mathsf{temp})$ do
51:         Set $\mathsf{arrivalTime}(\texttt{IB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{temp})$
52:     end for
```

```
53:     Set localTime ← localTime + $\mathsf{shift}_i$
54:     Set $i \leftarrow i + 1$
55: end while
56: Set isSync ← true and $t_{\mathsf{work}}$ ← localTime − 1
57: Run UpdateLocalChain to filter chains with future timestamps
58: for each beacon IB ∈ IBBuffer with TS(IBBuffer) $\leq (i+1) \cdot R$ do
59:     Parse arrivalTime(IB) as $(a, \mathsf{temp})$ and define arrivalTime(IB) $= (a, \mathsf{final})$
60: end for
OUTPUT: The protocol outputs ok to its caller (but not to $\mathcal{Z}$)
```

Protocol 15: *The joining procedure for a fresh new party.*

6 Full Protocol Analysis

Concentration bounds. In our analysis we apply a variety of large deviation bounds, assuming finite probability spaces and random variables with finite means.

Theorem 5 (Chernoff bounds). *Suppose $\{X_i : i \in [n]\}$ are mutually independent Boolean random variables, with $\mathbf{Pr}[X_i = 1] = p$, for all $i \in [n]$. Let $X = \sum_{i=1}^n X_i$ and $\mu = pn$. Then, for any $\delta \in (0, 1]$, it holds that*

$$\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\delta^2\mu/2} \text{ and } \mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\delta^2\mu/3}.$$

Also, for all $t > 0$,

$$\mathbf{Pr}[X \geq \mu + t] \leq e^{-2t^2 n}.$$

Definition 6. [DP09, Definition 5.3] *A sequence of random variables $(X_0, X_1, \ldots)$ is a martingale with respect to the sequence $(Y_0, Y_1, \ldots)$, if, for all $n \geq 0$, X_n is determined by $Y_0, \ldots, Y_n$ and $\mathbb{E}[X_{n+1} \mid Y_0, \ldots, Y_n] = X_n$.*

Theorem 6. [McD98, Theorem 3.15] *Let $(X_0, X_1, \ldots)$ be a martingale with respect to the sequence $(Y_0, Y_1, \ldots)$. Suppose an event G implies*

$$X_k - X_{k-1} \leq b \text{ (for all } k\text{) and } V = \sum_k \mathbf{Var}[X_k - X_{k-1} \mid Y_1, \ldots, Y_{k-1}] \leq v.$$

Then, for non-negative n and t,

$$\mathbf{Pr}[X_n \geq X_0 + t \wedge G] \leq \exp\Big\{-\frac{t^2}{2v + 2bt/3}\Big\}.$$

Other useful inequalities. Our analysis also employs the following inequalities.

Fact 1. *When $|r| \leq 1$, $\lim_{k=0}^{\infty} ar^k = a/(1-r)$.*

Fact 2. [BGK⁺21] *Let $(a_i)_{i=1}^n$ and $(b_i)_{i=1}^n$ be two sequences of n integers each, with the property that $\forall i \in [n], |a_i - b_i| \leq \Phi$. Then we have $|\mathsf{med}((a_i)_{i=1}^n) - \mathsf{med}((a_i)_{i=1}^n)| \leq \Phi$.*

6.1 Preliminary Notations, Definitions and Parameters

Our probability space is over all executions of length at most some polynomial in κ; we use **Pr** to denote the probability measure of this space. Furthermore, let $\mathcal{E}$ be a random variable taking values on this space and with a distribution induced by the random coins of all entities (adversary, environment, parties) and the random oracle.

For the sake of convenience, we define a *nominal time* that coincides with the internal variable τ_{sid} in $\mathcal{G}_{\mathsf{DClock}}$, indicating how many times the functionality "ticks" to all registered honest parties.

Definition 7 (Nominal time). *Given an execution of* Permissionless-SMR*, any prefix of the execution can be mapped deterministically to an integer r, which we call* nominal time*, as follows: r is the value of variable τ in the clock functionality at the final step of the execution prefix which is obtained by parsing the prefix from the genesis block and keeping track of the honest party set registered with the clock functionality (bootstrapped with the set of inaugural alert parties). In case no honest party exists in the execution, r is undefined.*

Recall that our parallel chain construction binds m independent mining procedures using $m\times 1$ PoW, we revisit the analytical frameworks in [GKL17, GKL20] and re-define all preliminary notations in the new context.

On the i-th single chain among m parallel chains, if at a nominal time r exactly h (alert) parties query the oracle with target T^i , the probability of at least one of them will succeed is

$$f(T^i, h) = 1 - (1 - pT^i)^h \leq pT^i h, \text{ where } p = 1/2^{\kappa/m}.$$

At nominal time r, alert parties might be querying the random oracle for various targets. We denote by $T_r^{i,\min}$ and $T_r^{i,\max}$ the minimum and maximum of those targets. Moreover, the initial target T_0 (which is the same for all chains) implies in our model an initial estimate of the number of honest RO queries h_0; specifically, $h_0 = 2^{\kappa/m}\Lambda_{\mathsf{epoch}}/(T_0 M)$, i.e., the number of parties it takes to produce Λ_{epoch} blocks of difficulty $1/T_0$ in time M. For convenience, we denote $f_0 = f(T_0, h_0)$ and simply refer to it as f. Also note that the ideal number of blocks $\Lambda_{\mathsf{epoch}} = M \cdot f$, so in the analysis we will use $M \cdot f$ to represent Λ_{epoch}.

Recall that our protocol runs in a (γ, s)-respecting requirement, for the ease of presentation, we introduce another parameter λ such that $\lambda + 2\Delta = \epsilon M/[2(1+\delta)\gamma^2]$.

"Good" properties. we present some definitions which will allow us to introduce a few ("good") properties, serving as an intermediate step towards proving the desired clock properties. Note that after switching to the new target recalculation function where blocks are counted to different epochs based on their timestamps, and parties forward epochs based on their local view, the original analytical framework [GKL17, GKL20] is infeasible to apply. Hence, following the treatment in [GKS22], we make necessary adaptions to consider the *target recalculation zone* — a sequence of consecutive nominal-time steps where alert parties may enter the next epoch in their local view.

- Nominal-time step r is *good* w.r.t. the i-th chain if $f/2\gamma^2 \leq ph_r T_r^{i,\min}$ and $ph_r T_r^{i,\max} \leq (1+\delta)\gamma^2 f$.
- Nominal-time step r is *good* if for any $i \in [m]$ it holds that $f/2\gamma^2 \leq ph_r T_r^{i,\min}$ and $ph_r T_r^{i,\max} \leq (1+\delta)\gamma^2 f$.
- A logical round $\langle \mathtt{itvl}, \mathtt{r}\rangle$ is a *target recalculation point* w.r.t. epoch *ep* if $(\mathtt{r} = \mathtt{itvl} \cdot R) \wedge [\mathtt{itvl} \bmod (M/R) = 0]$.
- A sequence of consecutive nominal-time steps $Z_{ep} = \{r_u, \ldots, r_v\}$ is a *target recalculation zone* w.r.t. target recalculation epoch *ep* if r_u (r_v resp.) is the smallest (largest resp.) time step s.t. some subset of synchronized parties are in the logical round that is a target recalculation point w.r.t. $ep - 1$.

- A target-recalculation zone Z_{ep} is *good* if for all $h_r, r \in Z_{ep}$ the target T_{ep} satisfies $f/2\gamma \leq ph_rT_{ep} \leq (1+\delta)\gamma f$.
- A chain is *good* if all its target recalculation zones are good.
- A chain is *ℓ-stale* for some nominal time u if it does not contain an honest block computed after nominal time $u - \ell - 2\Delta$.
- The *blocklength* of an epoch *ep* on i-th chain $\mathcal{C}$ is the number of blocks in $\mathcal{C}$ with timestamp $\langle itvl, \cdot\rangle$ such that $\lceil itvl/(M/R)\rceil = ep$.

Our goal is to prove that, at a certain nominal time r of the protocol execution, alert parties enjoy good properties on all parallel chains in their local view and reported timestamps. Thus, for each chain index i we extract all chains that either belong to alert parties at r or have accumulated sufficient difficulty and thus might be adopted in the future. We denote the chain set on i-th chain by $\mathcal{S}_r^i$:

$$\mathcal{S}_r^i \triangleq \left\{ \mathcal{C} \in E_r \;\middle|\; \begin{array}{l} \text{“}\mathcal{C} \text{ belongs to an alert party” or} \\ \text{“}\exists \mathcal{C}' \in E_r \text{ that belongs to an alert party and } \mathsf{diff}(\mathcal{C}) > \mathsf{diff}(\mathcal{C}') \text{ or} \\ \text{“}\exists \mathcal{C}' \in E_r \text{ that belongs to an alert party and } \mathsf{diff}(\mathcal{C}) = \mathsf{diff}(\mathcal{C}') \\ \qquad\qquad\qquad \text{and head}(\mathcal{C}) \text{ was computed no later than head}(\mathcal{C}') \end{array} \right\}.$$

Next, we define a series of useful predicates with respect to the potential chain set $\mathcal{S}_r$ and parties' local clocks at nominal time r. For a nominal-time step r, let:

- GoodChains(r) $\triangleq$ “For all $u \leq r$ and $i \in [m]$, every chain in $\mathcal{S}_u^i$ is good.”
- GoodTimesteps(r) $\triangleq$ “All time steps $u \leq r$ are good.”
- NoStaleChains(λ, r) $\triangleq$ “For all $u \leq r$ and $i \in [m]$, there is no λ-stale chains in $\mathcal{S}_u^i$.”
- NoStaleChains*(ℓ, i, r) $\triangleq$ “There is no ℓ-stale chains in $\mathcal{S}_r^i$.”
- BlockLength(r) $\triangleq$ “For all $u \leq r$ and $i \in [m]$, for all chain $\mathcal{C} \in \mathcal{S}_u^i$ the blocklength Λ of any epoch *ep* in $\mathcal{C}$ satisfies $\frac{1}{2(1+\delta)\gamma^2} \cdot Mf \leq \Lambda \leq 2(1+\delta)\gamma^2 \cdot Mf$.”
- GoodSkew(r) $\triangleq$ “For all alert parties in nominal time r, their local time differs by at most Φ if they are in the same interval.”

Random variables and Δ-isolated success. For the purpose of estimating the difficulty acquired by alert parties during a time window on each chain, we define the following random variables w.r.t. nominal time r and chain index i.

- D_r^i: the sum of the difficulties of all blocks computed by alert parties at nominal time r on the i-th chain.
- Y_r^i: the maximum difficulty of all blocks computed by alert parties at nominal time r on the i-th chain.
- Q_r^i: equal to Y_r^i when $D_u^i = 0$ for all $r < u < r + \Delta$ and 0 otherwise.

We call a nominal time r such that $D_r^i > 0$ *successful* w.r.t. the i-th chain and one wherein $Q_r > 0$ *isolated successful* w.r.t. the i-th chain. An isolated successful time step guarantees the irreversible progress of the honest parties on the corresponding chain (except for honest parties in different stages, see Section 6.3). We also note that isolated success is defined by not having any honest success in the future Δ nominal time steps, which is justified by our new timestamp scheme presented in Section 5.1.

Recall that the total number of hash queries alert parties (resp., the adversary) can make at nominal time r is denoted by h_r (resp., t_r). For a sequence of time steps S we write $h(S) = \sum_{r \in S} h_r$ and similarly $t(S)$, $D^i(S)$, $Q^i(S)$ where the superscript indicates the corresponding chain index.

Regarding the adversary $\mathcal{A}$, while $\mathcal{A}$ may query the random oracle for an arbitrarily low target and obtain blocks with arbitrarily high difficulty, we wish to upper-bound the difficulty it can accrue during a set of J queries. Consider, with respect to the i-th chain, a set of consecutive adversarial queries J and associate it with the target of the first query and denote it by $T^i(J)$. We define $A^i(J)$ and $B^i(J)$ to be equal to the sum of the difficulties of all blocks computed by the adversary on chain i during queries in J for target at least $T^i(J)/\tau$ and $T^i(J)$, respectively. That is, queries in J for targets less than $T^i(J)/\tau$ (resp. $T^i(J)$) do not contribute to $A^i(J)$ (resp. $B^i(J)$). While considering consecutive epochs of a particular chain, the target can either increase by at most τ (and $B^i(J)$ will be appropriate), or decrease by at most τ (and $A^i(J)$ will be useful).

Let $\mathcal{E}_{r-1}$ fix the execution just before time r. In particular, a value E_{r-1} of $\mathcal{E}_{r-1}$ determines the adversarial strategy and so determines the targets against which every party will query the oracle at time r and the number of parties h_r and t_r , but it does not determine D_r or Q_r. For an adversarial query j we will write E_{j-1} for the execution just before this query.

Fact 3 ([GKL20]). *Let U be a set of at most s consecutive time steps in a (γ, s)-respecting environment and $S \subseteq U$.*
(a) For any $h \in \{h_r : r \in U\}$, $\frac{h}{\gamma} \leq \frac{h(S)}{|S|} \leq \gamma h$.
(b) $h(U) \leq \big(1 + \frac{\gamma|U \setminus S|}{|S|}\big)h(S)$.

Protocol parameters and their conditions. We summary all Permissionless-SMR parameters in Table 2 in Appendix A. Especially, in our analysis we consider a consecutive set of ℓ time steps where ℓ is set as a constant and holds the following relation with other protocol parameters (which are also constants in our analysis).

$$\ell = \Theta\bigg(\frac{4(1+5\epsilon)}{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}} \cdot \gamma^3 \cdot \mathsf{max}\{\Delta, \tau\}\bigg) \tag{6}$$

Next, we consider intervals and epochs with durations that are set relatively large with respect to ℓ. Regarding the duration of an interval and three stages VC, OG and RC inside, they are all constant parameters independent of the security parameter and we require that

$$R_{\mathsf{VC}} = R_{\mathsf{RC}} \geq 2(\ell + 2\Delta) \text{ and } R_{\mathsf{OG}} = (\frac{\gamma}{\epsilon} - 4) \cdot (\ell + 2\Delta) \text{ and } R \geq \frac{\gamma}{\epsilon} \cdot (\ell + 2\Delta). \tag{7}$$

We also require that an epoch consists of poly-logarithmically many intervals $M = \Theta(R \log^2 \kappa)$.

We assume the following conditions on protocol parameters to be satisfied: First, the advantage of the honest parties δ is large enough to absorb error from the convergence quality ϵ. Specifically,

$$\epsilon \leq \delta/24 \leq 1/24. \tag{C1}$$

Next, network delay Δ, party fluctuation ratio γ are well set with respect to ϵ.

$$[1 - (1+\delta)\gamma^2 f]^{\Delta} \geq 1 - \epsilon. \tag{C2}$$

We also require that the bound on clock drift rate ρ is set relatively small.

$$\rho < \frac{f}{8\gamma^4} \cdot \epsilon^2(1-\epsilon)^2. \tag{C3}$$

Note that, the clock drift Φ and interval duration R also gives us the maximum skew that parties may deviate in an interval with initial skew Φ_{init}, and we write $2\rho R + (2 + \frac{1}{1+\rho}) \cdot \Delta = \Phi_{\mathsf{init}}/2 = \Phi/3$.

6.2 Properties of Protocol Executions

Honest convergence probability. We first show that the probability that good concentrations on random variables (specifically, lower bound on $Q(S)$ and upper/lower bound on $D(S)$) will happen except with error that decreases exponentially with respect to the length of time step sequences that we concern.

Definition 8 (Ideal-convergence events). *We define the following ideal-convergence events with respect to a chain index $i \in [m]$.*

- *For a set S of consecutive time steps, let $\mathsf{goodConv}^i_{\mathcal{H}}(S)$ denote the event that $Q^i(S)$ and $D^i(S)$ stays close to their expected value.*

$$\mathsf{goodConv}^i_{\mathcal{H}}(S) \triangleq (1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta} ph(S) < Q^i(S) \wedge (1-\epsilon)ph(S) < D^i(S) < (1+\epsilon)ph(S).$$

- *For a set J of consecutive adversarial queries and*

$$\alpha(J,k) = \frac{\epsilon f[1-(1+\delta)\gamma^2 f]^{\Delta}}{2(1+4\epsilon)\gamma^3\tau} \cdot \frac{k}{T(J)}, \tag{8}$$

let $\mathsf{goodConv}_{\mathcal{A}}(J,k)$ denote the event that $A^i(J)$ and $B^i(J)$ are well upper bounded.

$$\mathsf{goodConv}^i_{\mathcal{A}}(J) \triangleq A^i(J) < p|J| + \mathsf{max}\{\epsilon p|J|, \tau\alpha(J)\} \wedge B^i(J) < p|J| + \mathsf{max}\{\epsilon p|J|, \alpha(J)\}.$$

- *For nominal time r and $S = \{1, \ldots r\}$, let $\mathsf{goodConv}^i(r,k)$ denote the event where, for any time $u \le r-k$, $\mathsf{goodConv}^i_{\mathcal{H}}$ holds for $S[u:r]$ and $\mathsf{goodConv}_{\mathcal{A}}$ holds for $J[u:r]$ and $k = r-u$. Precisely,*

$$\mathsf{goodConv}^i(r,k) \triangleq \bigwedge_{u\in[r-k]} \Big(\mathsf{goodConv}^i_{\mathcal{H}}(S[u:r]) \wedge \mathsf{goodConv}^i_{\mathcal{A}}(J[u:r], r-u)\Big),$$

where $S = \{u, \ldots, r\}$ and J the set of adversarial queries in S.

The following proposition, considers the basic bounds on the expectation and variance of random variables D_r, Y_r (due to [GKL20]), and apply them per chain index.

Proposition 1. *For any time step r and any chain index $i \in [m]$, it holds that*
(a) $[1 - f(T^{i,\max}_r, h_r)]ph_r \le \mathbb{E}[Y^i_r|\mathcal{E}_{r-1} = E_{r-1}] \le \mathbb{E}[D^i_r|\mathcal{E}_{r-1} = E_{r-1}] = ph_r$.
(b) $\mathbb{E}[(Y^i_r)^2|\mathcal{E}_{r-1} = E_{r-1}] \le ph_r/T^{i,\min}_r$.
(c) $\mathbf{Var}[D^i_r|\mathcal{E}_{r-1} = E_{r-1}] \le ph_r/T^{i,\min}_r$.

Proof. Let us drop the superscript i on targets and random variables, and subscript r on number of parties for convenience.

(a) Suppose that the h honest parties at time r query for targets $T_1, \ldots, T_n$. Observe that all these variables are determined by $\mathcal{E}_{r-1}$. We have

$$\begin{aligned}\mathbb{E}[Y_r|\mathcal{E}_{r-1} = E_{r-1}] &= \sum_{i\in[h]} \frac{1}{T_i} \cdot \frac{T_i}{2^{\kappa/m}} \prod_{i<j}[1-f(T_j,1)] \ge \sum_{i\in[h]} p \prod_{j\in[h]} [1-f(T_j,1)] \\ &\overset{(*)}{\ge} \sum_{i\in[h]} p \prod_{j\in[h]} \big[1-f(T^{\max},1)\big] = \sum_{i\in[h]} p\big[1-f(T^{\max},h)\big] = ph\big[1-f(T^{\max},h)\big]\end{aligned}$$

where inequality $(*)$ holds because $f(T, n)$ is increasing in T.

For (b) and (c), it holds that

$$\mathbf{Var}[D_r | \mathcal{E}_{r-1} = E_{r-1}] \le \sum_{i \in [h]} \frac{1}{T_i^2} \cdot \frac{T_i}{2^{\kappa/m}} = \sum_{i \in [h]} \frac{p}{T_i} \le \frac{pn}{T^{\min}},$$

and $\mathbb{E}[Y_r^2 | \mathcal{E}_{r-1} = E_{r-1}]$ is upper-bounded alike. □

The following theorem gives an upper bound, as a function of k, on the event that $\mathsf{goodConv}_{\mathcal{H}}(S)$ does not hold for a consecutive S time steps with $|S| = k$.

Theorem 7. *In a (γ, σ)-respecting environment, for any chain i and any set S of at least $k \ge \ell$ consecutive* good *time steps, $\mathsf{goodConv}^i_{\mathcal{H}}(S)$ holds except with probability no more than $\epsilon_{\mathcal{H}}(k)$ where*

$$\epsilon_{\mathcal{H}}(k) \triangleq \exp\left\{ \ln(\Delta + 2) - \frac{\epsilon^2 f [1 - (1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3 \Delta (1 + \epsilon/3)} \cdot \min\{k, s\} \right\}.$$

Proof. Fix a chain index and drop all related superscripts on random variables. Fix an execution E_0 just before the beginning of S.

We first consider the lower bound on $Q(S)$. For each nominal time $i \in S$, define a Boolean random variable F_i equal to 1 exactly when all h_i queries of the honest parties yield evaluations above $\min\{T : f(T, h_i) \ge (1+\delta)\gamma^2 f\}$; define $Z_i = Y_i \cdot F_{i+1} \cdots F_{i+\Delta-1}$. Let G denote the event that the time steps in S are good. Given G, for any $i \in S, (F_i = 1) \implies (D_i = 0)$ and so $Q_i \ge Z_i$. For any d, it holds that

$$\mathbf{Pr}\Big[G \wedge \sum_{i \in [k]} Q_i \le d\Big] \le \mathbf{Pr}\Big[G \wedge \sum_{i \in [k]} Z_i \le d\Big]$$

thus we now work on Z_i.

Identify S with $\{1, \ldots, |S|\}$ and partition it with sets of the form $S_j = \{j, j + \Delta, j + 2\Delta, \ldots\}$ for $j \in \{0, 1, \ldots, \Delta - 1\}$. Fix a set $S_j = \{s_1, s_2, \ldots, s_\nu\}$, with $\nu \ge \lceil |S|/\Delta \rceil$, and define the event G_t as the conjunction of the events G and $t = \epsilon(1 - 2\gamma^2 f)^\Delta p h(S_j)$. We consider the following event:

$$G_t \wedge \sum_{i \in S_j} Z_i \le [1 - (1+\delta)\gamma^2 f]^\Delta p \sum_{i \in S_j} h_i - t.$$

To that end, consider the sequence of random variables

$$X_0 = 0; X_u = \sum_{i \in [u]} Z_{s_i} - \sum_{i \in [u]} \mathbb{E}[Z_{s_i} | \mathcal{E}_{s_i - 1}], u \in [\nu].$$

This is a martingale with respect to the sequence $\mathcal{E}_{s_1 - 1}(\mathcal{E}_0 = E_0), \ldots, \mathcal{E}_{s_\nu - 1}, \mathcal{E}$ because, following the linearity of conditional expectation and the fact that X_{u-1} is a deterministic function of $\mathcal{E}_{s_{u-1} + \Delta - 1} = \mathcal{E}_{s_u - 1}$, it holds that

$$\mathbb{E}[X_u | \mathcal{E}_{s_u - 1}] = \mathbb{E}\big[Z_{s_u} - \mathbb{E}[Z_{s_u} | \mathcal{E}_{s_u - 1}] \big| \mathcal{E}_{s_u - 1}\big] + \mathbb{E}[X_{u-1} | \mathcal{E}_{s_u - 1}] = X_{u-1}.$$

In addition, given an execution E satisfying G_t,

$$\epsilon \sum_{i \in S_j} \mathbb{E}[Z_i | \mathcal{E}_{s_u - 1} = E_{s_u - 1}] \ge \epsilon \sum_{i \in S_j} [1 - (1+\delta)\gamma^2 f]^\Delta p h_i = t.$$

Now, consider the details relevant to Theorem 6. For an execution E satisfying G_t, let B denote the event $\mathcal{E}_{s_u-1} = E_{s_u-1}$. Note that $Z^2_{s_u} = Y^2_{s_u} \cdot F_{s_u+1} \cdots F_{s_u+\Delta-1}$ and all these random variables are independent given B. Since $X_u - X_{u-1} = Z_{s_u} - \mathbb{E}[Z_{s_u}|\mathcal{E}_{s_u-1}]$, let $S_j^{(u)}$ denote a sub sequence of S_j of length $\mathsf{min}\{\nu, s/\Delta\}$ such that $s_u \in S_j^{(u)}$. It holds that

$$Z_{s_u} - \mathbb{E}[Z_{s_u}|B] \leq \frac{1}{T^{\min}_{s_u}} \leq \frac{ph_{s_u}}{ph_{s_u}T^{\min}_{s_u}} \leq \frac{\gamma ph(S_j^{(u)})}{ph_{s_u}T^{\min}_{s_u}|S_j^{(u)}|} \leq \frac{2\gamma^3 t}{\epsilon(1-2\gamma^2 f)^{\Delta} f \cdot \mathsf{min}\{\nu, s/\Delta\}} \overset{\text{def}}{=} b. \quad (9)$$

The third inequality holds due to Fact 3(a); and the next one is because s_u is a good time step. We see that the event G implies $X_u - X_{u-1} \leq b$.

With respect to $V = \sum_u \mathbf{Var}[X_u - X_{u-1}|\mathcal{E}_{s_u-1}] \leq \sum_u \mathbb{E}[Z^2_{s_u}|\mathcal{E}_{s_u-1}]$, first recall Fact 3(a), we have

$$\sum_{u\in[\nu]} \big(ph_{s_u}\big)^2 \leq \sum_{u\in[\nu]} ph_{s_u}\frac{p\gamma h(S_j^{(u)})}{|S_j^{(u)}|} \leq \frac{\gamma}{\mathsf{min}\{\nu, s/\Delta\}}\big(ph(S_j)\big)^2.$$

Then, using the independence of the random variables it holds that

$$\begin{aligned}\sum_{u\in[\nu]} \mathbb{E}[Z^2_{s_u-1}|B] &\leq [1-(1+\delta)\gamma^2 f]^{\Delta-1}\sum_{u\in[\nu]}\frac{\big(ph_{s_u}\big)^2}{ph_{s_u}T^{\min}_{s_u}} \leq \frac{2\gamma^3[1-(1+\delta)\gamma^2 f]^{\Delta-1}}{f\cdot \mathsf{min}\{\nu, s/\Delta\}}\cdot\big(ph(S_j)\big)^2 \\ &\leq \frac{2\gamma^3 t^2}{\epsilon^2 f(1-2\gamma^2 f)^{\Delta+1}\cdot \mathsf{min}\{\nu, s/\Delta\}} \overset{\text{def}}{=} v. \quad (10)\end{aligned}$$

The first inequality holds due to Proposition 1(b); the second one is because the inequality above and that all time steps in S_j are good; and the last one is acquired by substituting t.

After applying Theorem 6, we have

$$\mathbf{Pr}[-X_\nu \geq t \wedge G_t] \leq \exp\Big\{-\frac{t^2}{2v(1+\epsilon/3)}\Big\} \leq \exp\Big\{-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3\Delta(1+\epsilon/3)}\cdot \mathsf{min}\{k, s\}\Big\}.$$

Note that the first inequality follows $bt < \epsilon v$; and the next one holds by substituting b, v as in Equations (9) and (10) and $\nu \leq k/\Delta$. Finally, we apply the union probability to all $j \in \{0, 1, \ldots, \Delta-1\}$ thus

$$\mathbf{Pr}\Big[G\wedge\sum_{i\in[k]} Q_i \leq (1-\epsilon)(1+\delta\gamma^2 f)^{\Delta} ph(S)\Big] \leq \exp\Big\{\ln\Delta - \frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3\Delta(1+\epsilon/3)}\cdot \mathsf{min}\{k, s\}\Big\} \overset{\text{def}}{=} \epsilon_Q(k).$$

Regarding the bounds on $D(S)$, we consider per honest query. Let J denote the queries in S ($\nu = |J|$), and Z_i the difficulty of any block obtained from query $i \in J$. For the lower bound, define the martingale sequence

$$X_0 = 0; X_u = \sum_{i\in[u]} Z_i - \sum_{i\in[u]}\mathbb{E}[Z_i|\mathcal{E}_{i-1}], u \in [\nu]$$

and let $t = \epsilon p\nu$. Analogously, by considering a sub sequence of S of length $\mathsf{min}\{\nu, s\}$, we have

$$X_u - X_{u-1} \leq \frac{2\gamma^3 t}{\epsilon f\cdot \mathsf{min}\{\nu, s\}} \overset{\text{def}}{=} b \text{ and } V \leq \frac{2\gamma^3 t^2}{\epsilon^2 f\cdot \mathsf{min}\{\nu, s\}} \overset{\text{def}}{=} v.$$

After applying Theorem 6, it holds that

$$\mathbf{Pr}\Big[G\wedge\sum_{i\in[k]} D_i \le (1-\epsilon)ph(S)\Big] \le \exp\Big\{-\frac{t^2}{2v(1+\epsilon/3)}\Big\} \le \exp\Big\{-\frac{\epsilon^2 f}{4\gamma^3(1+\epsilon/3)}\cdot\mathsf{min}\{k,s\}\Big\} \overset{\text{def}}{=} \epsilon_D(k).$$

The error probability on violating the upper bound on $D(S)$ can be computed in the same way, and yields the same as $\epsilon_D(k)$. To compute $\epsilon_{\mathcal{H}}$, we consider the union bound that $Q(S)$ and $D(S)$ yields good concentration. Specifically, let $G_{Q,D}(S)$ denote the event $(1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta} ph(S) < Q^i(S) \wedge (1-\epsilon)ph(S) < D^i(S) < (1+\epsilon)ph(S)$, it holds that

$$\begin{aligned}\mathbf{Pr}[\neg G_{Q,D}(S)] &= 1-[1-\epsilon_Q(k)][1-\epsilon_D(k)]^2 \le \epsilon_Q(k)+2\epsilon_D(k)\\ &\le \exp\Bigg\{\ln(\Delta+2)-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3\Delta(1+\epsilon/3)}\cdot\mathsf{min}\{k,s\}\Bigg\},\end{aligned}$$

which defines $\epsilon_{\mathcal{H}}(k)$. □

Next, consider a sequence of adversarial queries J. We show that the probability of the event regarding the violation of $\mathsf{goodConv}_{\mathcal{A}}(J,k)$ is also upper-bounded as a function of k.

Theorem 8. *For any chain i and any set J of consecutive adversarial queries and $\alpha(J,k)$ as defined in Equation (8), it holds that*

$$A^i(J) < p|J| + \mathsf{max}\{\epsilon p|J|, \tau\alpha(J)\} \quad\text{and}\quad B^i(J) < p|J| + \mathsf{max}\{\epsilon p|J|, \alpha(J)\}$$

except with probability no more than $\epsilon_{\mathcal{A}}(k)$ where

$$\epsilon_{\mathcal{A}}(k) \triangleq \exp\Bigg\{-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta}}{4\gamma^3\tau(1+5\epsilon)}\cdot k\Bigg\}.$$

Proof. For each $j\in J$, let A_j be equal to the difficulty of the block obtained with the j-th query as long as the target was at least $T(J)/\tau$; thus, $A(J)=\sum_{j\in J} A_j$. If $|J|=\nu$, identify J with $[\nu]$ and define the martingale

$$X_0=0;\ X_u=\sum_{j\in[u]} A_j - \sum_{j\in[u]}\mathbb{E}[A_i|\mathcal{E}_{j-1}], u\in[\nu].$$

For all $u\in[\nu]$ we have $X_u-X_{u-1}\le \tau/T(J)$, $\mathbf{Var}[X_u-X_{u-1}|\mathcal{E}_{u-1}]\le p\tau/T(J)$, and $E[A_j|\mathcal{E}_{j-1}]\le p$.

We now apply Theorem 6 with $b=\tau/T(J)$, $v=bp\nu\le bt/\epsilon$; and since $\epsilon<1/2$ due to Condition (C1), we have $(1+4\epsilon)(1+\frac{\epsilon}{3})<(1+5\epsilon)$ hence set

$$t=\mathsf{max}\Bigg\{\epsilon p\nu, 2(\frac{1}{\epsilon}+\frac{1}{3})b\cdot\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta}}{4\gamma^3\tau(1+5\epsilon)}\cdot k\Bigg\}.$$

We have

$$\mathbf{Pr}\Big[\sum_{j\in J} A_j \ge p\nu + t\Big] \le \exp\Big\{-\frac{t}{2b(\frac{1}{3}+\frac{1}{\epsilon})}\Big\} \le \exp\Bigg\{-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta}}{4\gamma^3\tau(1+5\epsilon)}\cdot k\Bigg\},$$

which defines $\epsilon_{\mathcal{A}}(k)$. □

Given the convergence error $\epsilon_{\mathcal{H}}(k)$ and $\epsilon_{\mathcal{A}}(k)$, we now consider the event $\mathsf{goodConv}(r,k)$. Note that, since $s=\Theta(\log^2\kappa)$ thus $\epsilon_{\mathcal{H}}(s)$ and $\epsilon_{\mathcal{A}}(s)$ are both negligible, we may consider $\mathsf{goodConv}(r,k)$ for $k<s$.

Lemma 4. *Let E be a typical execution in a (γ, s)-respecting environment. If the execution E_{r-1} is good with respect to chain i, then $\mathsf{goodConv}^i(r,k)$ holds except with probability no more than $\epsilon_{\mathsf{conv}}(k)$ where*

$$\epsilon_{\mathsf{conv}}(k) \triangleq \frac{1}{1-c}\cdot\exp\left\{\ln(\Delta+3)-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3(1+5\epsilon)\cdot\max\{\Delta,\tau\}}\cdot k\right\} \text{ and } c=\exp\left\{-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3(1+5\epsilon)\cdot\max\{\Delta,\tau\}}\right\}.$$

Proof. Fix a time step $u \leq r-k$ and $S=\{u,\ldots,r\}$ and J the adversarial queries in S. It holds that $\mathbf{Pr}[\neg(\mathsf{goodConv}_{\mathcal{H}}(S)\wedge \mathsf{goodConv}_{\mathcal{A}}(J))] \leq 1-(1-\epsilon_{\mathcal{A}}(k))(1-\epsilon_{\mathcal{H}}(k)) \leq \epsilon_{\mathcal{A}}(k)+\epsilon_{\mathcal{H}}(k)$.

Consider a new function $\epsilon^*(k)$ such that

$$\epsilon_{\mathcal{A}}(k)+\epsilon_{\mathcal{H}}(k) \leq \exp\left\{\ln(\Delta+3)-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3(1+5\epsilon)\cdot\max\{\Delta,\tau\}}\cdot\min\{k,s\}\right\} \overset{\text{def}}{=} \epsilon^*(k).$$

Since $\epsilon^*(k)$ is an exponential function when $k\leq s$, it holds that for any $k \leq s$, there exist a constant $c\in(0,1)$ such that

$$\frac{\epsilon^*(k+1)}{\epsilon^*(k)} = \exp\left\{-\frac{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\Delta+1}}{4\gamma^3(1+5\epsilon)\cdot\max\{\Delta,\tau\}}\right\} = c$$

Therefore, consider the union probability for all time steps in $\{\max\{1,r-s\},\ldots,r-k\}$ and using Fact 1, it holds that

$$\begin{aligned}\mathbf{Pr}[\neg\mathsf{goodConv}(r,\ell)] &= 1-\prod_{1}^{r-\ell}(1-\mathbf{Pr}[\neg(\mathsf{goodConv}_{\mathcal{H}}(S)\wedge\mathsf{goodConv}_{\mathcal{A}}(J))]) \\ &\leq \sum_{1}^{r-\ell}\mathbf{Pr}[\neg(\mathsf{goodConv}_{\mathcal{H}}(S)\wedge\mathsf{goodConv}_{\mathcal{A}}(J))] \\ &\leq \epsilon^*(k)\cdot(1+c+c^2+\ldots) \leq \frac{\epsilon^*(k)}{1-c},\end{aligned}$$

which defines $\epsilon_{\mathsf{conv}}(k)$. □

Corollary 1. *For any $c\in(0,1)$, there exists $\ell\in\mathbb{N}^+$ such that for any sequence S of consecutive time steps with $|S| = \mathsf{poly}(\ell)$, $\mathsf{goodConv}^i(r,\ell)$ holds for all time steps in S except with probability no more than c.*

Proof. We apply the union probability to all time steps $r\in S$ which yields

$$1-\prod_{r\in S}\big(1-\epsilon_{\mathsf{conv}}(\ell)\big) \leq \mathsf{poly}(\ell)\cdot\epsilon_{\mathsf{conv}}(\ell).$$ □

Corollary 2. *Let E be a typical execution in a (γ, s)-respecting environment. If the execution E_{r-1} is good with respect to chain i, then there exist a subset $\mathcal{I}\subseteq[m]$ of size at least $(1-\epsilon)(1-\epsilon_{\mathsf{conv}}(k))m$ such that for all $i\in\mathcal{I}$, $\mathsf{goodConv}^i(r,k)$ holds.*

Proof. Given that the executions on each chain are mutually independent and the number of chains $m=\Theta(\log^2\kappa)$, we prove this corollary using Chernoff bound (Theorem 5). In more details, let X_i denote the random variable such that if $\mathsf{goodConv}^i(r,k)$ holds then $X_i = 1$; and otherwise $X_i=0$. Also let $X=\sum_{i\in[m]}X_i$. Due to Lemma 4, $\mathbf{Pr}[X_i=1]\geq 1-\epsilon_{\mathsf{conv}}(k)$ We have $\mathbb{E}[X] \geq (1-\epsilon_{\mathsf{conv}}(k))m$. Hence, it holds that

$$\mathbf{Pr}\big[X\leq(1-\epsilon)(1-\epsilon_{\mathsf{conv}}(k))m\big] \leq \mathbf{Pr}\big[X\leq(1-\epsilon)\mathbb{E}[X]\big] \leq \exp\big\{-\epsilon^2(1-\epsilon_{\mathsf{conv}}(k))m/2\big\}.$$

Note that $m=\Theta(\log^2\kappa)$, the event such that good convergence does not hold on sufficiently many chains happens with probability negligible in terms of the security parameter. □

Bad events with respect to the random oracle. Following [GKL14], we consider three bad events — namely, block insertions, copies and predictions — with respect to the random oracle. An *insertion* happens when a block $\mathcal{B}^*$, created after two consecutive blocks $\mathcal{B}$ and $\mathcal{B}'$ on chain $\mathcal{C}$, yields $\mathcal{B}, \mathcal{B}^*, \mathcal{B}'$ three consecutive blocks of a valid chain; a *copy* occurs if the same block exists in two different positions; a *prediction* occurs when a block extends one with later creation time.

Note that, with parallel chains and $m\times 1$ PoW, these bad events ought to be reasoned per chain index (while in previous works the entire RO output is assigned to the only single chain). For example, an insertion with respect to chain index $i \in [m]$ happens if the i-th segment of hash of block $\mathcal{B}^*$ is identical to the i-th segment of hash of block $\mathcal{B}$, implying a "partial" collision on two different RO queries. We show in the following theorem that for all parallel chains, bad events happen with only negligible probability.

Theorem 9. *Consider an execution E of $L = \mathsf{poly}(\kappa)$ time steps. No insertions, no copies, and no predictions occurred in E, except with probability negligibly small in κ.*

Proof. Let Q denote the total number of random oracle queries all parties made in $L = \mathsf{poly}\kappa$ time steps. We consider the partial collision which happens when there exist two RO outputs h, h' and an integer $i \in [m]$ such that $[h]_{i\sim m} = [h']_{i\sim m}$. Fix i, a partial collision with respect to i happens with probability $Q^2/2^{\kappa/m} = \exp(2\log Q - \omega(\log\kappa))$ which is negligible in κ. Since $m = \Theta(\log^2\kappa)$, the probability that no partial collision happens for any $i \in [m]$ yields $1-(1-\exp(2\log Q-\omega(\log\kappa)))^m \leq m\cdot\exp(2\log Q - \omega(\log\kappa)) = \exp(2\log Q - \omega(\log\kappa))$. □

6.3 Averaged Chain Growth Lemma

We consider in this section the Chain Growth lemma, which provides a lower bound on the progression of accumulated difficulty of the honest parties that holds irrespective of the adversary. Note that this lemma has been proved in various settings: In [GKL14] with fixed number of parties and synchronous network; in [GKL17] with dynamic participation and synchronous networks; in [GKL20] with dynamic participation and bounded-delay networks; and in [GKS22] with additionally the imperfect local clocks.

Recall that we are in a dynamic, bounded-delay setting with drifting clocks and our timestamp scheme in Section 5.1 does not ask for monotonically increasing timestamps for blocks in the same stages. This implies that during any period of time, as long as all honest local clocks are in the same stage, an isolated success guarantees the progression on honest chains regardless of the honest block difficulty and adversarial behavior. Hence, Chain-Growth lemma still applies *unconditionally* in the same stage which we state as Same-stage Chain Growth lemma as follows.

Lemma 5 (Same-stage Chain Growth). *Let u, v be two nominal time steps of an execution E such that all honest local clocks stay in the same stage during time $[u, v]$. If at time u an honest party diffuses a chain of difficulty d with chain index i, then by time v every honest party has received a chain of difficulty at least $d + Q^i(S)$ where $S = \{r : u + \Delta \leq r \leq v - \Delta\}$.*

In our analysis, we will concern sequence of time steps that span among different stages. Yet, we highlight that the unconditional Chain-Growth lemma does not apply when honest local clocks split in different stages. This is because timestamp monotonicity is required for blocks in different stages, which implies that when honest parties stay in different stages, an isolated success does not necessarily contribute to the progress of honest chains. For example, suppose that party P at nominal time r produces a block $\mathcal{B}$ extending a chain $\mathcal{C}$, with timestamp of the second stage of the first interval, followed by Δ time steps that no honest block being produced. Then, another party P' at time $r' = r + \Delta'$ ($\Delta' > \Delta$) produces another block $\mathcal{B}'$, extending the same chain $\mathcal{C}$ with timestamp of the first stage of the same interval, also followed by Δ time steps that no honest block being produced. Such event happens in that P' defers the processing of block $\mathcal{B}$ due to clock drifts. As a result, while each time r and r' qualifies for an isolated success, honest chains progresses by only $\max\{Q(r), Q(r')\}$.

To address this issue and simplify our analysis, we now prove a *conditional* variant of the Chain-Growth lemma. Specifically, we show the lower bound on honest progression after dropping all isolated successes during nominal time steps that honest parties split in different stages. Such approach works so long as (i) the number of nominal time steps that parties stay in different stages (and therefore the number of isolated success) is bounded with respect to clock drift rate ρ and the length of an interval; and (ii) the difference of honest block difficulty is bounded in neighboring stages. Note that the above two aspects implicitly requires certain pre-conditions. First, the length of time-step sequences that we concern should be relatively large compared with the upper bound on time steps that parties split, otherwise the impact of trimming those time steps might be unbounded. Next, in a sequence of fixed number of time steps honest parties should always query blocks with targets close to each other.

To this end, we revise the Chain Growth lemma and apply these two pre-conditions. Intuitively, this can be viewed as counting all isolated success for steps that parties do not split and then "averaging" them to all time steps. We now state this "averaged" variant of Chain Growth as follows.

Lemma 6 (Averaged Chain Growth). *In a (γ, s)-respecting environment, let u, v be two nominal time steps of an execution E such that all time steps in $[u, v]$ are good. If at time u an honest party diffuses a chain of difficulty d with chain index i, then by time v every honest party has received a chain of difficulty at least*

$$d + Q^i(S) - \epsilon(1-\epsilon)(1-(1+\delta)\gamma^2 f)^{\Delta} ph(S),$$

where $S = \{r : u + \Delta \le r \le v - \Delta\}$. If additionally $v - u \ge \ell$ and $Q^i(S) > (1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta} ph(S)$, then by time v every honest party has received a chain of difficulty at least

$$d + Q^{i,\text{avg}}(S),$$

where $Q^{i,\text{avg}}(S) \triangleq (1-\epsilon)Q^i(S)$.

Proof. Let us drop the chain-index superscript i for convenience. If two blocks are obtained at time steps which are at distance at least Δ and all parties are in the same stage in between, then we are certain that the later block increased the accumulated difficulty. Consider $S' \subseteq S$ such that for all $i, j \in S'$, $|i - j| \ge \Delta$ and $Y_i > 0$ and $S'' \subseteq S'$ such that for all $i \in S''$ all parties stay in the same stage. We argue that, by time v, every honest party has a chain of difficulty at least $d + Y(S'') \le d + Q(S'') = d + Q(S) - Y(S' \backslash S'')$. This is because every honest party will receive the chain of difficulty d by time $u + \Delta$ and so the first block obtained in S'' extends a chain of weight at least d. Next, note that if a block obtained in S'' is the head of a chain of weight at least d', then the next block in S'' extends a chain of weight at least d'.

Regarding $Y(S' \backslash S'')$, we may consider S' of size between $\ell/2$ and $R/2$ with at most one subsequence of consecutive time steps that parties split in different stages. Such partition always works in that the duration of each stage is larger than ℓ but small than R.

Fix a partition S' and denote $S' \backslash S'' = \{s_1, \ldots, s_n\}$ $(n < \Phi)$. It holds that

$$Q(S' \backslash S'') \le \sum_{i \in [n]} \frac{1}{T^{\min}_{s_i}} \le \Phi \cdot \frac{2\gamma^3 ph(S)}{f|S|} \le \frac{4\gamma^3 \Phi}{f\ell} ph(S) \le \epsilon(1-\epsilon)(1-(1+\delta)\gamma^2 f)^{\Delta} ph(S).$$

The last inequality follows Condition (C3). $\square$

Lemma 7. *Let $S = \{r : u \le r \le v\}$ be a set of consecutive at least ℓ time steps and J the set of adversarial queries in $U = \{r : u - \Delta \le r \le v + \Delta\}$.*

- $\mathsf{goodConv}^i_{\mathcal{H}}(U) \implies (1+\epsilon)p|J| \le Q^{i,\mathrm{avg}}(S) \le Q^i(S) \le D^i(U)$.
- *If* $\mathsf{goodConv}^i_{\mathcal{A}}(J,k)$ *holds, then either* $A(J) < (1+\epsilon)p|J|$ *or*

$$T(J)A(J) < (1-\epsilon)^2 \cdot \frac{[1-(1+\delta)\gamma^2 f]^{\Delta} f}{2\gamma^3} \cdot k.$$

Proof. (a) The middle inequality follows directly with the definitions. For the other two, let us first verify the following inequalities:

$$h(U) = h(S) + h(U \backslash S) \le \Big(1 + \frac{2\gamma\Delta}{\ell}\Big)h(S) < (1 + \frac{\epsilon^2}{2})h(S).$$

The first inequality comes from Fact 3(b); and the next one

(b) Either $\epsilon p|J| \ge \tau\alpha(J)$ and Definition 8 applies directly, or $p|J| < \tau\alpha(J)/\epsilon$ thus we get the inequalities by substituting $\alpha(J,k)$ as defined in Equation (8).

$$T(J)A(J) < (1 + \frac{1}{\epsilon}) \cdot \frac{\epsilon[1-(1+\delta)\gamma^2 f]^{\Delta} fk}{2(1+4\epsilon)\gamma^3} < (1-\epsilon)^2 \cdot \frac{[1-(1+\delta)\gamma^2 f]^{\Delta} f}{2\gamma^3} \cdot k.$$

The second inequality follows $(1+4\epsilon)(1-\epsilon)^2 > 1+\epsilon$ and Condition (C1). □

6.4 Steady Block Generation Rate and Bounded Skews

All statements in this section assume a (γ, s)-respecting environment, and the Conditions are assumed to hold for the initialization parameters n_0 and T_0.

We first prove that long forks appears on only a bounded number of parallel chains.

Lemma 8. *Assuming* GoodTimesteps$(r-1)$*, for any* $i \in [m]$*,* NoStaleChains$^*(\ell, i, r-1)$ *holds except with probability no more than* $\epsilon_{\mathsf{conv}}(\ell)$*.*

Proof. Recall Lemma 4 where $\mathsf{goodConv}^i(r)$ holds with probability no more than $\epsilon_{\mathsf{conv}}(\ell)$, it suffices to show that when $\mathsf{goodConv}^i(r)$ holds, $\mathcal{S}^i_r$ contains no stale chains with probability 1.

Suppose $\mathsf{goodConv}^i(r)$ holds and, towards a contradiction, $\mathcal{C} \in \mathcal{S}^i_r$ and has not been extended by an honest party for at least $\ell + 2\Delta$ time steps and r is the least (nominal) time with this property. Let $\mathcal{B}$ be the last honestly-generated block of $\mathcal{C}$ (possibly the genesis) and let w be the (nominal) time it was computed. We consider $S = \{u : w + \Delta \le u \le r - \Delta\}$ and $U = \{u : w \le u \le r\}$ ($|S| \ge \ell$ by assumption). Suppose that the blocks of $\mathcal{C}$ after $\mathcal{B}$ (we denote these blocks by B) span k epochs. For $i \in [k]$ let m_i be the number of blocks in epoch i. We denote the target in the i-th epoch with T_i and set $M = m_1 + \ldots + m_k$ and $d = \sum_{i\in[k]} T_i$. Our plan is to contradict the assumption that $\mathcal{C} \in \mathcal{S}^i_r$ by showing that all chains in $\mathcal{S}^i_r$ have more difficulty than $\mathcal{C}$. By Averaged Chain-Growth Lemma 6, all the honest parties have advanced (in difficulty) during the time steps in U by $Q^{\mathrm{avg}}(S)$. Therefore, to reach a contradiction it suffices to show that $d < Q^{\mathrm{avg}}(S)$.

Consider the following partition on B: we partition B into u sections $B_v, v \in [u]$ and associate each section B_v with the target of its first block T_v. Section B_v starts with either the block after $\mathcal{B}$ (if $v = 1$) or the $\lceil m_i/2 \rceil$-th block in an epoch (if $v > 1$); it ends at either the last block of the chain (if $v = u$) or the $(\lceil m_i/2 \rceil + 1)$-th block such that in epoch $i+1$ the target is less than T_v/τ. Under such partition, the next block after partition B_v is exactly the first block of partition B_{v+1}.

For $u \ge 2$, we claim that for partition B_v, it has the following properties: (1) for all blocks in B_v, their target is at least T_v/τ; and (2) the number of blocks in B_v is at least $Mf/2$. Property

(1) holds because of the strategy of our partition that will stop before it exceeds the lower bound for the targets and thanks to Equation (4) we need to pass at least two boundaries of epochs so the circumstance that no blocks exist in such partition will never happen. To reason why property (2) stands, consider those epochs that are split into two different sections. For an epoch ep whose blocks are split into two sections B_v, B_{v+1}, since in epoch $ep+1$ the target is larger than that in ep (if not, it does not satisfy the criteria of the partition), there are at least Mf blocks in epoch ep. Otherwise, Equation (4) will raise the target. By the rule of partition, at least $Mf/2$ blocks are in each sections. Hence for every partition, either its head or tail has at least $Mf/2$ blocks in the same epoch, and this implies the lower bound of the total number of blocks.

For each $v \in [u]$, let $j_v \in J$ denote the index of the query during which the first block of the v-th section was computed and set $J_v = \{j : j_v \leq j < j_{i+1}\}$ (Theorem 9) assures $j_i < j_{i+1}$. We have

$$d = \sum_{i \in [k]} T_i < \sum_{v \in [u]} (1+\epsilon)|J_v| \leq (1+\epsilon)|J| \leq Q^{\text{avg}}(S).$$

The first inequality holds because by setting $k = \lambda$ and due to Lemma 7(b) we get either $A(J) < (1+\epsilon)p|J|$ or $T(J)A(J) \leq Rf/[2(1+\delta)\gamma^3]$ with overwhelming probability (and note that $Rf/[2(1+\delta)\gamma^3] \leq Rf/2$). The last inequality is due to Lemma 7(a).

If $u = 1$, let J denote the queries in U starting from the first adversarial query attempting to extend $\mathcal{B}$. Then, $T_1 = T(J)$ and $T_2 \geq T(J)/\tau$ thus $d < A(J)$. If $A(J) < (1+\epsilon)p|J|$, then $A(J) < Q^{\text{avg}}(S)$ is obtained by Lemma 7(a). Otherwise, first observe that $pn(S) \geq pT_w h_w |S|/(\gamma T_w) \geq f|S|/(2\gamma^3 T_w)$; by considering the first $\mathsf{min}\{k, \lambda\}$ time steps, it holds that

$$Q^{\text{avg}}(S) \geq (1-\epsilon)^2 (1-(1+\delta)\gamma^2 f)^{\Delta} \cdot \frac{f|S|}{2\gamma^3 T_w} > A(J).$$

The last inequality holds because we apply Lemma 7(b) by setting $k = |S|$. $\square$

Note that, when Lemma 8 holds for a chain i throughout an interval, it implies that by pruning k_{bootstr} blocks from each honest party's local chain i we achieve common prefix [GKL15]. Due to the similar argument in [GKL20] we set $k_{\mathsf{bootstr}} = 2\ell f$.

Additionally, if we replace ℓ in Lemma 8 with any integer larger than $\lambda = \Theta(\log^2 \kappa)$, we get the following corollary. Note that now $\epsilon_{\mathsf{conv}}(\lambda)$ is negligibly small in the security parameter hence no λ-stale chains holds throughout the execution.

Corollary 3. $\textsc{GoodTimesteps}(r-1) \implies \textsc{NoStaleChains}(\lambda, r)$.

The two lemmas below shows that by adopting the target recalculation function in Equation (4) independently on each chain, they all maintain good block length in each epoch and good block generation rate in each target recalculation zone. Note that these the proof of these two lemmas (on each chain) can be viewed as a reminiscent of the proofs in [GKS22] where they use the same function to adjust mining difficulty on a single chain, hence we omit them and refer to [GKS22] for more details.

Lemma 9. $\textsc{GoodTimesteps}(r-1) \wedge \textsc{GoodChains}(r-1) \wedge \textsc{GoodSkew}(r-1) \implies \textsc{BlockLength}(r)$.

Lemma 10. $\textsc{GoodTimesteps}(r-1) \implies \textsc{GoodChains}(r)$.

Given that parties maintain good block generation rates in each target recalculation zone and the fact that $s \geq (1+\rho)R$, we show that good block generation rate is maintained throughout the execution.

Corollary 4. $\textsc{GoodTimesteps}(r-1) \implies \textsc{GoodTimesteps}(r)$.

Proof. Consider any $i \in [m]$ thus any chain $\mathcal{C} \in \mathcal{S}_r^i$. Let Z_{ep} be its last target recalculation zone before r. If $r \in Z_{ep}$, it follows directly by Lemma 10 that it is good. Otherwise, consider a time $w \in Z_{ep}$ (recall that $f/2\gamma \le ph_w T_{ep} \le (1+\delta)\gamma f$). Since the duration of an epoch implies $r - w < s$, we have $h_r/\gamma \le h_w \le \gamma h_r$. Combining these two bounds we obtain the desired inequality. □

Given that long forks could only happen in a bounded fraction of parallel chains, we then show that for any honest party, their view of current interval contains sufficiently many chains with two good properties at the end of an interval — first, parties share a common view on that chain, and second the majority of input blocks included in that chain are generated by honest parties. Also note that we can parameterize our protocol to make η an arbitrarily small constant.

Theorem 10. *Let $\mathbb{C}$ and $\mathbb{C}'$ denote the parallel chains held by two honest parties P and P' at the end of an interval itvl, respectively. There exists a subset $S \subseteq \{1, 2, \ldots, m\}$, $|S| > m - \eta$, such that for all $i \in S$, the following properties hold on chains $\mathcal{C} = \mathbb{C}_i$ and $\mathcal{C}' = \mathbb{C}'_i$:*
- **Agreement**: $\mathcal{C}^{\lceil R\cdot itvl - R_{\mathsf{RC}}} = \mathcal{C}'^{\lceil R\cdot itvl - R_{\mathsf{RC}}}$.
- **Honest input-block majority**: *More than half of the PoW transactions included in $\mathcal{C}^{\lceil R\cdot itvl - R_{\mathsf{RC}}}$ and $\mathcal{C}'^{\lceil R\cdot itvl - R_{\mathsf{RC}}}$ are produced by honest parties.*

Proof. Recall Corollary 1 and Lemma 8, agreement holds due to the fact that $\textsc{NoStaleChains}^*(\ell, i, r)$ holds for any $i \in S$ and time r that parties stay in interval *itvl*.

We now prove that honestly generated input blocks that are included on chain account for the majority. Let u denote the first (nominal) time such that all alert parties are mining input blocks w.r.t. interval *itvl*. Consider a set of consecutive nominal time steps $S = \{i : u \le i \le v\}$ where where all the honest queries in S are doing 2-for-1 PoW w.r.t. interval *itvl* and hence contribute to the honest beacon set. Let $\mathcal{B}$ be the last block produced by honest parties before time v and denote its production time by w (in terms of the nominal time index). Since $\mathcal{C}$ will become stale if there is no honest block since w for $\ell + 2\Delta$ time steps, we get that $w < v - (\ell + 2\Delta)$.

Let $S_1 = \{i : u \le i \le w - \Delta\}$ and $S_2 = \{i : u - (2\ell + 4\Delta) \le i \le w + (\ell + 2\Delta)\}$. S_1 is the time interval that honest success can contribute to the beacon set w.r.t. interval *itvl*; and S_2 is for the adversary. The earliest time step of S_1 is derived from the definition of u and the largest time is because it will take up to Δ time steps for all beacons to be diffused to and accepted by all alert parties. The earliest time step of S_2 is acquired due to the unpredictability of an honest block (cf. Lemma 8). Regarding the largest time step of S_2, it is achieved by considering the first honest block $\mathcal{B}'$ after v, which is produced no later than $w' = w + \ell + 2\Delta$ (otherwise it violates "no stale chains"). The adversary can no longer include beacons to the output generation stage after w' as it can no longer revert $\mathcal{B}'$ before the end of the interval, so all the subsequent beacons produced after w' are invalid w.r.t. the current chain. Note that $|S_1| \ge [R - (5\ell + 11\Delta)]/(1+\rho)$ and $|S_2 \backslash S_1| \le (1+\rho)(3\ell + 7\Delta)$.

Let J denote the adversarial queries associated with S_2. In order to prove that alert parties can produce at least half of synchronization beacons, it suffices to show that

$$D(S_1) > d/2.$$

We first show that the number of RO queries alert parties can make during S_2 is at most 10ϵ more than those in S_1. We have

$$h(S_2) \le \left(1 + \frac{\gamma|S_2 \backslash S_1|}{S_1}\right) h(S_1) \le \left(1 + \frac{\gamma(1+\rho)^2(3\ell + 7\Delta)}{R - (5\ell + 11\Delta)}\right) \le \left(1 + \frac{4(1+\epsilon)\epsilon}{1 - 6\epsilon/\gamma}\right) < (1 + 10\epsilon) h(S_1).$$

The first inequality follows from Fact 3(b); the third one holds due to Equation (7) and Condition (C1); and the last one is by Condition (C1) ($\epsilon < 1/12$). Next,

$$\begin{aligned}D(S_1) \geq (1-\epsilon)ph(S_1) > (1-11\epsilon)ph(S_2) &> \frac{1-11\epsilon}{2-\delta}p[h(S_2)+|J|] \\ &> \frac{1-12\epsilon}{2-\delta}[D(S_2)+A(J)] > \frac{1}{2}[D(S_1)+A(J)] = \frac{d}{2}.\end{aligned}$$

The first inequality follows Theorem 7; the second one is achieved by substituting $h(S1)$ with $h(S2)$; the next inequality follows from the honest majority assumption; the last inequality holds due to Condition (C1) ($\delta \geq 24\epsilon$). ☐

Bounded skews. We now show that the synchronization procedure run at the end of each interval helps parties tighten their clock skews and stay in a good linear envelope with respect to real time.

Lemma 11. $\textsc{GoodSkew}(r-1) \wedge \textsc{GoodChains}(r-1) \implies \textsc{GoodSkew}(r)$.

Proof. For nominal-time steps that all alert parties stay in the first interval, their local clock can differ with each other for up to $(1+\rho)R - (1+\rho)^{-1}R + \Phi_{\mathsf{init}} \leq 2\rho R + \Phi_{\mathsf{init}} \leq \Phi$ time steps in that no clock synchronization happened.

Next, suppose that honest local clocks deviate from each other for at most Φ in interval $itvl$. We consider the nominal time r such that at least one alert party enters the next interval $itvl+1$. We show that for those parties that have finished adjusting their clock, the logical time that they report can deviate from each other for up to Φ.

Let shift_i denote the the shift computed by a party P on its i-th chain at the end of interval $itvl$. We first show that if good properties holds on chain j, then all honest parties (if they adjust time based on chain j) will adjust their clock back to $\Phi_{\mathsf{init}}/2$ after they enter interval $itvl+1$. Consider the first nominal time step such that at least one honest party enters interval $itvl+1$. Let μ denote the timestamp of P after adding the adjusted shift computed from Equation (5) to its local time in interval $itvl$; and let $(\mu_1, \ldots, \mu_m)$ denote the set of timestamps held by a party P acquired by adding $(\mathsf{shift}_1, \ldots, \mathsf{shift}_m)$ to its local time respectively. Consider the following claim.

Claim 1. *For any two honest parties P and P', there exist a set $S \subseteq [m]$ with $|S| \geq m - \eta$ such that for any $k \in S$, we have $|\mu_k - \mu'_k| \leq \Phi_{\mathsf{init}}/2$.*

Proof. Fix $i \in S$ such that on the i-th chain, $\mathbb{C}_i^{itvl,\mathsf{P}} = \mathbb{C}_i^{itvl,\mathsf{P}'}$ and the majority of the input-blocks in $\mathbb{C}_i^{itvl,\mathsf{P}}$ are produced by honest parties. I.e., honest parties share a unanimous view of the input-block set IB. For each $\mathtt{IB} \in IB$, let $\mu_i(\mathtt{IB})$ denote the time after using beacon $\mathtt{IB}$ to update local time; we have $\mu_i(\mathtt{IB}) = \mathtt{r}_i + \mathsf{TS}(\mathtt{IB}) - \mathsf{P}.\mathsf{arrivalTime}(\mathtt{IB})$. We are going to show that for any two honest parties P and P', $|\mu_i(\mathtt{IB}) - \mu'_i(\mathtt{IB})| \leq \Phi_{\mathsf{init}} - (1+\rho)\Delta$.

Notice that the arrival time of $\mathtt{IB}$ in the view of party P can be represented as

$$\mathsf{P}.\mathsf{arrivalTime}(\mathtt{IB}) = \mathtt{r}_i - \rho^{\mathsf{P}}(r - r_{\mathtt{IB}}) + \Delta_{\mathsf{P},\mathtt{IB}},$$

where $r_{\mathtt{IB}}$ is the nominal time that $\mathtt{IB}$ is emitted to the network if $\mathtt{IB}$ is honest, and is the first nominal time step such that at least one honest party receives $\mathtt{IB}$ if $\mathtt{IB}$ is adversarial. And ρ^{P} is the clock speed of party P during nominal time r and $r_{\mathtt{IB}}$. The quantity $\Delta_{\mathsf{P},\mathtt{IB}} \in [\Delta]$ is the time elapsed (counted by nominal-time steps) for $\mathtt{IB}$ to be delivered to P. By substituting we get

$$\mu_i(\mathtt{IB}) = \mathsf{TS}(\mathtt{IB}) + \rho^{\mathsf{P}}(r - r_{\mathtt{IB}}) - \Delta_{\mathsf{P},\mathtt{IB}}.$$

Note that for different parties, $|\mu_i(\texttt{IB}) - \mu'_i(\texttt{IB})| \le 2\rho R + (1+\rho)\Delta$.

Now consider the tuples $(\mu_i(\texttt{IB}))_{\texttt{IB}\in IB}$ and $(\mu'_i(\texttt{IB}))_{\texttt{IB}\in IB}$. By applying Fact 2 and Condition (C3), we get

$$\left|\mathsf{med}\Big((\mu_i(\texttt{IB}))_{\texttt{IB}\in IB}\Big) - \mathsf{med}\Big((\mu'_i(\texttt{IB}))_{\texttt{IB}\in IB}\Big)\right| \le 2\rho R + (1+\rho)\Delta \le \Phi_{\mathsf{init}}/2.$$

The similar argument works for any $i \in S$ which concludes the proof. □

Now for each party P we construct a new array $(\hat{\mu}_1, \dots \hat{\mu}_m)$ such that if $i \in S$, $\hat{\mu}_i = \mathsf{max}_{\mathsf{P}\in\mathcal{H}}\mu_i^{\mathsf{P}}$ is the maximum of μ_i among all honest parties; and $\hat{\mu}_i = \mu_i$ otherwise. $\hat{\mu}_i$ shares a subset of size at least $m-\eta$. Hence, the following inequality holds for any (possibly the same) $\mathsf{P}_1, \dots, \mathsf{P}_k \in \mathcal{H}$.

$$\mathsf{min}\hat{\mu} \le \hat{\mu}_{\eta+1}^{\mathsf{P}_1} \le \hat{\mu}_{2\eta+1}^{\mathsf{P}_2} \le \dots \le \hat{\mu}_{m-\eta}^{\mathsf{P}_k} \le \mathsf{max}\hat{\mu}.$$

Notice that the shift calculation algorithm can be re-written by first compute the time after adding the shifts on each chain, and then apply the array operations, we have

$$\mu \triangleq \mathsf{avg}(\mathsf{select}(\mathsf{reduce}(\{\mu_i\}_{i\in[m]}, \eta), \eta)) \text{ and } \hat{\mu} \triangleq \mathsf{avg}(\mathsf{select}(\mathsf{reduce}(\{\hat{\mu}_i\}_{i\in[m]}, \eta), \eta)).$$

When $m \ge 5\eta$, it holds that for any two parties P and P', $|\hat{\mu} - \hat{\mu}'| < (1/3)(\mathsf{max}\hat{\mu}_{m-\eta} - \mathsf{min}\hat{\mu}'_{\eta+1}) = \Phi_{\mathsf{init}}/2$. Also notice that for a party P, $\mu \ge \hat{\mu} \ge \mu - \Phi_{\mathsf{init}}/2$. Combining them together we get for any two parties P and P', $|\mu - \mu'| < \Phi_{\mathsf{init}}$. □

Lemma 12. GoodSkew(r) $\implies$ *for any alert party* P *and its local time* $\texttt{r}$, *it holds that* $\frac{1}{1+\Gamma} \cdot r \le \texttt{r} \le (1+\Gamma)\cdot r$ *for* $\Gamma = \rho + \epsilon$.

Proof. Consider two "virtual" parties $\mathsf{P}_{\mathsf{fast}}^{\mathsf{virt}}$ and $\mathsf{P}_{\mathsf{slow}}^{\mathsf{virt}}$. They are parties that passively listen to the protocol execution and update their logical clock, yet they do not perform any mining operations (thus an execution with virtual parties is indistinguishable from another one without them).

Fix an interval i and let $\mathsf{P}_{\mathsf{fast}}^{\mathsf{virt}}$ start the interval at the same time as the first alert party enters this interval, and her clock always runs at rate $1+\rho$ (i.e., the fastest speed); and $\mathsf{P}_{\mathsf{slow}}^{\mathsf{virt}}$ starts the i-th interval with the last alert party that enters this interval, and her clock always runs at rate $(1+\rho)^{-1}$ (the slowest speed).

Let $\{\mathcal{C}_j\}_{j\in S}(S \subseteq [m])$ denote the chains where parties share the common view and acquires majority of input blocks at the end of interval i. For any chain $\mathcal{C}_j$ and any honestly generated beacon $\texttt{IB} \in \mathcal{C}_j$, the following holds due to the honest majority of input blocks and the fact that $\mathsf{P}_{\mathsf{fast}}^{\mathsf{virt}}$ ($\mathsf{P}_{\mathsf{slow}}^{\mathsf{virt}}$ resp.) maintains the largest (smallest resp.) local time throughout the interval.

$$-\Delta \le \mathsf{med}\{\mathsf{TS}(\texttt{IB}) - \mathsf{arrivalTime}_{\mathsf{P}_{\mathsf{slow}}^{\mathsf{virt}}}(\texttt{IB})\} \text{ and } \mathsf{med}\{\mathsf{TS}(\texttt{IB}) - \mathsf{arrivalTime}_{\mathsf{P}_{\mathsf{fast}}^{\mathsf{virt}}}(\texttt{IB})\} \le 0$$

Recall Equation (5), we have $-\Delta \le \mathsf{shift}_{\mathsf{P}_{\mathsf{slow}}^{\mathsf{virt}}}$ and $\mathsf{shift}_{\mathsf{P}_{\mathsf{fast}}^{\mathsf{virt}}} \le 0$. Thus, for any party $\mathsf{P} \in \mathcal{H}$ at the end of interval i, their local clock stays in the Γ-linear envelope where $\Gamma = \rho + (\Delta/R) \le \rho + \epsilon$. □

6.5 State Machine Replication

Regarding the fast fairness property, we show that any party that controls α fraction of the computational power has probability $(1-\epsilon')\alpha$ to get selected as the output for any constant $\epsilon' > 0$. Note that ϵ is the protocol parameter on ideal convergence quality, and $\epsilon' = 12\epsilon$ can be tuned to be an arbitrarily small constant.

Lemma 13. *Consider a BA invocation that outputs* $(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$ *at time* r *and let* W *denote the time window of the first interval where parties replace their input with a block with minimum hash. Assume a party* P *makes* h *random oracle queries during* W *and let* n *denote the number of total random oracle queries made by all parties in* W*. The probability that* tx_1 *contains* P*'s public key is* $(1-\epsilon')(h/n)$ *for* $\epsilon' = 12\epsilon$*.*

Proof(sketch). First note that due to Theorem 9 that collision on RO queries happen with only negligible probability. Suppose the king chain in the first interval is a good chain - i.e., honest parties share common view on the chain. Following a similar argument in Theorem 10, we learn that honest parties can contribute to the probability that one of h queries yield the minimum hash among n queries is $(1-\epsilon)(h/n)/(1+10\epsilon) \geq (1-11\epsilon)(h/n)$. Now, under the protocol parametrization that king chain yields a bad chain with ϵ probability, it holds that tx_1 contains P's public key is at least $(1-12\epsilon)(h/n)$. □

In Theorem 1 we conclude that when the clock drift, network latency and all other protocol parameters satisfy certain conditions, our protocol Permissionless-SMR solves the clock synchronization problem achieves all desired properties.

Theorem 1. *In a* (γ, s)*-respecting environment, assume that honest majority holds throughout the execution, all parties are equipped with physical clocks with* ρ*-bounded rate and the diffusion network is* Δ*-bounded. If all protocol parameters are well-set under Condition* (C1)*,* (C2) *and* (C3)*, then* Permissionless-SMR *(Protocol 3) solves the state machine replication problem and achieves consistency, expected-constant liveness,* Φ*-bounded skews and* $(\rho+\epsilon)$*-accuracy; and fast fairness, except with probability negligibly small with respect to the security parameter.*

Proof. Regarding consistency and expected-constant liveness, our proof is a reminiscent of the Chain-King Consensus in [GKS24]. For the clock skew and accuracy part, it follows directly with Lemmas 11 and 12. The fast fairness property follows Lemma 13. □

Acknowledgements

Juan Garay's research has been supported in part by NSF grants no. 2001082 and 2055694. Yu Shen's research has been supported by Input Output (`iohk.io`) through their funding of the University of Edinburgh Blockchain Technology Lab.

References

[AMN+20] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy*, pages 106–118, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press. `https://doi.org/10.1109/SP40000.2020.00044`.

[BG89] Piotr Berman and Juan A. Garay. Asymptotically optimal distributed consensus. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *Automata, Languages and Programming*, pages 80–94, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press. `https://doi.org/10.1145/3243734.3243848`.

[BGK$^+$21] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Dynamic ad hoc clock synchronization. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part III*, volume 12698 of *Lecture Notes in Computer Science*, pages 399–428, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-77883-5_14.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science*, pages 410–415, Research Triangle Park, NC, USA, October 30 – November 1, 1989. IEEE Computer Society Press. https://doi.org/10.1109/SFCS.1989.63511.

[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, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 585–602, London, UK, November 11–15, 2019. ACM Press. https://doi.org/10.1145/3319535.3363213.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-63688-7_11.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press. https://doi.org/10.1145/168588.168596.

[Bra84] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *3rd ACM Symposium Annual on Principles of Distributed Computing*, pages 154–162, Vancouver, BC, Canada, August 27–29, 1984. Association for Computing Machinery. https://doi.org/10.1145/800222.806743.

[Can00a] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000. https://doi.org/10.1007/s001459910006.

[Can00b] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. URL: https://eprint.iacr.org/2000/067.

[CCGZ19] Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. Probabilistic termination and composability of cryptographic protocols. *Journal of Cryptology*, 32(3):690–741, July 2019. https://doi.org/10.1007/s00145-018-9279-y.

[CEM$^+$20] T-H. Hubert Chan, Naomi Ephraim, Antonio Marcedone, Andrew Morgan, Rafael Pass, and Elaine Shi. Blockchain with varying number of players. Cryptology ePrint Archive, Report 2020/677, 2020. URL: https://eprint.iacr.org/2020/677.

[CHMV17] Ran Canetti, Kyle Hogan, Aanchal Malhotra, and Mayank Varia. A universally composable treatment of network time. In Boris Köpf and Steve Chong, editors, *CSF 2017: IEEE 30th Computer Security Foundations Symposium*, pages 360–375, Santa Barbara, CA, USA, August 21–25, 2017. IEEE Computer Society Press. https://doi.org/10.1109/CSF.2017.38.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[CPS19] T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. Consensus through herding. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 720–749, Darmstadt, Germany, May 19–23, 2019. Springer, Cham, Switzerland. `https://doi.org/10.1007/978-3-030-17653-2_24`.

[DHS86] Danny Dolev, Joseph Y Halpern, and Raymond H Strong. On the possibility and impossibility of achieving clock synchronization. *J. Comput. Syst. Sci.*, 32(2):230–250, apr 1986.

[DKSS22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022. `https://doi.org/10.1145/3492321.3519594`.

[DLP$^+$86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986. `https://doi.org/10.1145/5925.5931`.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988. `https://doi.org/10.1145/42282.42283`.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147, Santa Barbara, CA, USA, August 16–20, 1993. Springer Berlin Heidelberg, Germany. `https://doi.org/10.1007/3-540-48071-4_10`.

[DNS98] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent zero-knowledge. In *30th Annual ACM Symposium on Theory of Computing*, pages 409–418, Dallas, TX, USA, May 23–26, 1998. ACM Press. `https://doi.org/10.1145/276698.276853`.

[Dol82] Danny Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982. `https://doi.org/10.1016/0196-6774(82)90004-9`.

[DP09] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms.* Cambridge University Press, 2009. `https://doi.org/10.1017/CBO9780511581274`.

[DS98] Cynthia Dwork and Amit Sahai. Concurrent zero-knowledge: Reducing the need for timing constraints. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 442–457, Santa Barbara, CA, USA, August 23–27, 1998. Springer Berlin Heidelberg, Germany. `https://doi.org/10.1007/BFb0055746`.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014: 18th International Conference on Financial Cryptography and Data Security*, volume 8437 of *Lecture Notes in Computer Science*, pages 436–454, Christ Church, Barbados, March 3–7, 2014. Springer Berlin Heidelberg, Germany. `https://doi.org/10.1007/978-3-662-45472-5_28`.

[FG03] Matthias Fitzi and Juan A. Garay. Efficient player-optimal protocols for strong and differential consensus. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *22nd ACM Symposium Annual on Principles of Distributed Computing*, pages 211–220, Boston, MA, USA, July 13–16, 2003. Association for Computing Machinery. `https://doi.org/10.1145/872035.872066`.

[FGKR20] Matthias Fitzi, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ledger combiners for fast settlement. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 322–352, Durham, NC, USA, November 16–19, 2020. Springer, Cham, Switzerland. `https://doi.org/10.1007/978-3-030-64375-1_12`.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. `https://doi.org/10.1145/3149.214121`.

[GK20] Juan A. Garay and Aggelos Kiayias. SoK: A consensus taxonomy in the blockchain era. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 284–318, San Francisco, CA, USA, February 24–28, 2020. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-40186-3_13.

[GKL14] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. Cryptology ePrint Archive, Report 2014/765, 2014. URL: https://eprint.iacr.org/2014/765.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Sofia, Bulgaria, April 26–30, 2015. Springer Berlin Heidelberg, Germany. https://doi.org/10.1007/978-3-662-46803-6_10.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-63688-7_10.

[GKL20] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. Full analysis of nakamoto consensus in bounded-delay networks. Cryptology ePrint Archive, Report 2020/277, 2020. URL: https://eprint.iacr.org/2020/277.

[GKS22] Juan A. Garay, Aggelos Kiayias, and Yu Shen. Permissionless clock synchronization with public setup. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022: 20th Theory of Cryptography Conference, Part III*, volume 13749 of *Lecture Notes in Computer Science*, pages 181–211, Chicago, IL, USA, November 7–10, 2022. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-22368-6_7.

[GKS24] Juan A. Garay, Aggelos Kiayias, and Yu Shen. Proof-of-work-based consensus in expected-constant time. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part III*, volume 14653 of *Lecture Notes in Computer Science*, pages 96–125, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-58734-4_4.

[HMM85] Joseph Y Halpern, Nimrod Megiddo, and Ashfaq A Munshi. Optimal precision in the presence of uncertainty. *Journal of Complexity*, 1(2):170–196, 1985. https://doi.org/10.1016/0885-064X(85)90010-X.

[HSSD84] Joseph Y. Halpern, Barbara Simons, H. Raymond Strong, and Danny Dolev. Fault-tolerant clock synchronization. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *3rd ACM Symposium Annual on Principles of Distributed Computing*, pages 89–102, Vancouver, BC, Canada, August 27–29, 1984. Association for Computing Machinery. https://doi.org/10.1145/800222.806739.

[KK06] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 445–462, Santa Barbara, CA, USA, August 20–24, 2006. Springer Berlin Heidelberg, Germany. https://doi.org/10.1007/11818175_27.

[KLP05] Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. Concurrent general composition of secure protocols in the timing model. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 644–653, Baltimore, MA, USA, May 22–24, 2005. ACM Press. https://doi.org/10.1145/1060590.1060687.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498, Tokyo, Japan, March 3–6, 2013. Springer Berlin Heidelberg, Germany. `https://doi.org/10.1007/978-3-642-36594-2_27`.

[LL84a] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2):190–204, 1984. `https://doi.org/10.1016/S0019-9958(84)80033-9`.

[LL84b] Jennifer Lundelius and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *3rd ACM Symposium Annual on Principles of Distributed Computing*, pages 75–88, Vancouver, BC, Canada, August 27–29, 1984. Association for Computing Machinery. `https://doi.org/10.1145/800222.806738`.

[LL22] Christoph Lenzen and Julian Loss. Optimal clock synchronization with signatures. In Alessia Milani and Philipp Woelfel, editors, *41st ACM Symposium Annual on Principles of Distributed Computing*, pages 440–449, Salerno, Italy, July 25–29, 2022. Association for Computing Machinery. `https://doi.org/10.1145/3519270.3538444`.

[LMS85] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, jan 1985. `https://doi.org/10.1145/2455.2457`.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, Jul 1982. `https://doi.org/10.1145/357172.357176`.

[McD98] Colin McDiarmid. Concentration. In Michel Habib, Colin McDiarmid, Jorge Ramirez-Alfonsin, and Bruce Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, pages 195–248, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. `https://doi.org/10.1007/978-3-662-12788-9_6`.

[MS85] Stephen R. Mahaney and Fred B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In Michael A. Malcolm and H. Raymond Strong, editors, *4th ACM Symposium Annual on Principles of Distributed Computing*, pages 237–249, Minaki, Ontario, Canada, August 5–7, 1985. Association for Computing Machinery. `https://doi.org/10.1145/323596.323618`.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

[PS17a] Rafael Pass and Elaine Shi. FruitChains: A fair blockchain. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *36th ACM Symposium Annual on Principles of Distributed Computing*, pages 315–324, Washington, DC, USA, July 25–27, 2017. Association for Computing Machinery. `https://doi.org/10.1145/3087801.3087809`.

[PS17b] Rafael Pass and Elaine Shi. Rethinking large-scale consensus. In Boris Köpf and Steve Chong, editors, *CSF 2017: IEEE 30th Computer Security Foundations Symposium*, pages 115–129, Santa Barbara, CA, USA, August 21–25, 2017. IEEE Computer Society Press. `https://doi.org/10.1109/CSF.2017.37`.

[PS17c] Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 380–409, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland. `https://doi.org/10.1007/978-3-319-70697-9_14`.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, apr 1980. `https://doi.org/10.1145/322186.322188`.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, Paris, France, April 30 – May 4, 2017. Springer, Cham, Switzerland. `https://doi.org/10.1007/978-3-319-56614-6_22`.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990. `https://doi.org/10.1145/98163.98167`.

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, jul 1987. `https://doi.org/10.1145/28869.28876`.

[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 Symposium Annual on Principles of Distributed Computing*, pages 347–356, Toronto, ON, Canada, July 29 – August 2, 2019. Association for Computing Machinery. `https://doi.org/10.1145/3293611.3331591`.

A Glossary

Permissionless-SMR **parameters.** In Table 2, we list all the protocol parameters and their explanation.

Variable	Description
κ	Security parameter; length of the random oracle output.
m	The number of parallel chains ($m = \Theta(\log^2 \kappa)$).
R	The length of a synchronization interval in number of rounds; each interval consists of three stages: (i) view convergence of length R_{VC} rounds; (ii) input-block generation of length R_{OG} rounds; and (iii) reference convergence of length R_{RC} rounds.
M	The length of a target recalculation epoch in number of rounds.
T	The target to successfully solve a PoW.
h_r	Number of RO queries made by honest parties at (nominal) time r.
t_r	Number of RO queries made by corrupted parties at (nominal) time r.
δ	Advantage of honest parties ($t \leq (1-\delta)h$).
f	Initial (and ideal) block generation rate.
ϵ	Ideal quality of concentration of random variables.
Φ_{init}	The initial skew (i.e., difference in nominal time) that parties may start at the beginning of each interval.

Table 2: *Main parameters of* Permissionless-SMR.

Main state variables of Permissionless-SMR **participants.** We summarize the internal state variables of a protocol participant as follows.

Variable	Description
`localTime`	Party P's local time in the form of interval index and round index `localTime` $= \langle$`itvl`, `r`$\rangle$.
`ep`	Party P's local epoch index.

$\mathbb{C}_{\mathsf{local}}$	Party P's local working parallel chains (which P tries to extend).
$\mathbb{T}_{\mathsf{local}}$	Party P's local parallel tree (all valid chains that P has seen so far).
`futureChains`	The buffer that stores all incoming new parallel-chains.
`buffer`	The buffer that stores all incoming transactions.
`IBBuffer`	The buffer that stores all incoming input blocks.
`isSync`	A bit variable to store the synchronization status
`fetchCompleted`	A variable to store whether the round messages have been fetched.
$\mathsf{arrivalTime}(\cdot)$	A mapping from input-blocks to their corresponding local arrival time.
`state`	The ledger state.
`snapshot`	A mapping from interval index to party's local view (represented by m block hashes) at the end of that interval.

Table 3: *Main state variables in* Permissionless-SMR*.*