Ouroboros Chronos: Permissionless Clock Synchronization via Proof-of-Stake

Christian Badertscher[1⋆], Peter Gaži[1], Aggelos Kiayias[1,2⋆⋆], Alexander Russell[1,3⋆⋆⋆], and Vassilis Zikas[2†]

[1] IOHK – firstname.lastname@iohk.io
[2] University of Edinburgh – firstname.lastname@ed.ac.uk
[3] University of Connecticut – acr@cse.uconn.edu

Abstract. Clock synchronization allows parties to establish a common notion of global time by leveraging a weaker synchrony assumption, i.e., local clocks with approximately the same speed. The problem has long been a prominent goal for fault-tolerant distributed computing with a number of ingenious solutions in various settings. However, despite intensive investigation, the existing solutions do not apply to common blockchain protocols, which are designed to tolerate variable—and potentially adversarial—participation patterns, e.g., sleepiness and dynamic availability. Furthermore, because such blockchain protocols rely on freshly joining (or re-joining) parties to have a common notion of time, e.g., a global clock which allows knowledge of the current protocol round, it is not clear if or how they can operate without such a strong synchrony assumption.

In this work, we show how to solve the global synchronization problem by leveraging proof of stake (PoS). Concretely, we design and analyze a PoS blockchain protocol in the above dynamic-participation setting, that does not require a global clock but merely assumes that parties have local clocks advancing at approximately the same speed. Central to our construction is a novel synchronization mechanism that can be thought as the blockchain-era analogue of classical synchronizers: It enables joining parties—even if upon joining their local time is off by an arbitrary amount—to quickly calibrate their local clocks so that they all show approximately the same time. As a direct implication of our blockchain construction—since the blockchain can be joined and observed by any interested party—we obtain a permissionless PoS implementation of a global clock that may be used by higher level protocols that need access to global time.

1 Introduction

Global clock synchronization [15,27,20] allows a set of mutually distrustful parties to approximate on a global notion of "time," in such a manner that if some party believes that the global time is τ then every party believes it to be $\tau \pm \epsilon$ for some small $\epsilon > 0$. This in terms, allows for an (approximately) synchronous (or partially synchronous) execution of distributed protocols which has placed the study of such *synchronizers* at a prominent prosition in the theoretical computer science research. A number of works investigating feasibility across the spectrum of security/adversary models—from perfect to computational security and for different types of network synchronisation assumption [15,27,20,17,16,2,37,28,32,36]. (For completeness, we include a description of the current landscape of feasibility in Appendix A.) The common assumption of such synchronizers is that the (honest) parties have local (initially desynchronized) clocks which advance at (roughly) the same speed.

Notwithstanding, existing synchronization techniques are inapplicable to the standard model used for the analysis of Nakamoto-style blockchain/consensus protocols. The reason is that the above traditional models assume accurate knowledge of the total number of parties present in the system, and their main

⋆ Work done while the author was at the University of Edinburgh, Scotland.
⋆⋆ Research partly supported by EU Project No. 780477, PRIVILEDGE.
⋆⋆⋆ This material is based upon work supported by the National Science Foundation under Grant No. 1717432.
† Work supported in part by IOHK and done in part while the author was visiting the Simons Institute for the Theory of Computing, UC Berkeley.

tool is smart counting of messages (or message chains) received—a standard technique in the distributed computing literature that is also adopted in "iterated" Byzantine Fault Tolerant (iBFT) protocols. In contrast, in the Nakamoto-style setting, Pass and Shi put forth the *sleepy model* of consensus as a natural model for the analysis of such protocols, which reduces the applicability of such counting-based approaches. The reason is that the sleepy model allows for parties to join the protocol at any time and then to temporarily *sleep*—i.e., drop out of the protocol[4]—according to an arbitrary (or even adversarial) sleep pattern. This model was later generalized—and adapted to the (G)UC setting [9,10]—in the *dynamic availability* model of [4], which captures sleepiness with respect to arbitrary resources available to the protocol, e.g, the clock (capturing temporary loss of time), the network (capturing temporary connectivity issues), or the random oracle (temporary unavailability of computing resources).

The above dynamic participation assumptions, however natural, limit the power of existing synchonization techniques, since the lack of agreement of participation patterns makes counting ineffective to take consistent decisions without additional synchrony assumptions. In fact, the cryptographic analysis of formally specified and analyzed proof-of-work (PoW) and proof-of-stake (PoS) blockchains, has typically assumed a (partially) synchronous model with a notion of global time. Indeed, standard references for the proven security of Bitcoin [18,19,33] implicitly use the fact that they can refer to a global round index in order to prove the desired properties of the protocol. For example, the common-prefix property is defined to require that if an honest party holds a chain at Round ρ, then the prefix of this chain—obtained by removing the k most recent blocks—will eventually become prefix of the chain of any honest party (at some round $\rho' \geq \rho$). The assumption was made explicit in [5] by assuming a global clock in the global UC setting [10]: this permits every party to query a common clock on demand and from that deduce the current round. A similar approach, assuming access to a global clock, was also adopted in the constructions of PoS blockchains, such as Sleepy Consensus [35], Snow White [6], and Ouroboros [26,14,4]. So a natural question arises: Do the above blockchain protocols preserve their security when the (implicit) agreement on the current round[5] is replaced by the assumption of local potentially unsynchronized clocks that proceed at roughly the same speed?

Interestingly, answering this question reveals a delicate distinction between proof-of-work (PoW) based protocols, in particular Bitcoin, and common Nakamoto-style proof-of-stake (PoS) based protocols. In particular, the description of the Bitcoin blockchain (without difficulty recalibration) can rely on a purely *execution-driven* notion of time: when a party engages in the protocol it need not be aware of the current global time. In fact, in the static difficulty setting, proving security in this way follows immediately from [18,33]. Moreover, a notion of global time may be inferred by the current blockchain length in each party's local state. In the PoS setting, however, this execution-driven notion of time has been achieved only by Algorand [12], requiring a highly relevant concession: that explicit participation bounds are known to the protocol participants, i.e., each protocol participant at any given time is aware of how many protocol parties are expected to engage in the protocol step. Such a rigid participation restriction is not necessary for the Bitcoin blockchain.

This state of affairs leaves an important gap between the PoW setting and those PoS protocols that require no explicit participation bounds such as [35,6,26,14,4]: these protocols use a notion of global time hardwired in the protocol logic, and the protocol is unspecified without such global knowledge of time/round. In fact, there does not seem to be a simple way to removing this dependence of global time, and replace it by local clocks (even, perfectly-coordination ones that advance at exactly the same speed) while preserving the security guarantees. (Of course, one could include such a notion of (approximate) global time in a trusted *checkpointing assumption* [13], but this defeats the purpose of decoupling the protocol from an explicitly assumed trusted source of global time when joining the protocol which, as discussed below, is the main challenge of our work.)

In a nutshell, the reason is that in these PoS blockchains a party's right to create a block is always associated with a concrete round[6] (also called "slot"), and in order to verify that a block is created by an eligible party, that party must include a proof explicitly referring to the slot number. This means that a

[4] Different degrees of sleepiness allow for queueing received messages which are read once the sleepy party re-joins (light sleepers) or even missing the messages sent while the party is asleep (heavy sleepers)

[5] Recall that this implicit agreement was abstracted as an explicit global clock in [5,4].

[6] One can view this as an implicit timestamp

new party that joins the blockchain—or one that has been sleeping for long—cannot prune-off chains with adversarial timestamps so that he eventually adopts the right chain. Thus if a new party with an incorrect local time joins the protocol and sees a chain that includes blocks which appear to be far in the future (according to her local time), she cannot decide whether the chain is adversarial—in which case she needs to ignore or truncate it—or her local time is far behind absolute time. It is worth adding that these are not merely theoretical considerations: in a real world deployment the dependency on a global clock is typically met by using a global time synchronization service, e.g., NTP [30] and hence the security of all these protocols becomes compromised if such service fails to deliver a truly reliable clock. This is a possibility that cannot be excluded [29].

Note that all previous PoS protocols which can operate in a participation-unrestricted setting [35,6,26,14,4] require an upper bound on the network delay Δ to be known to all participants; this in fact is a necessary assumption, see [34], due to the participation uncertainty. Unfortunately, knowing an upper bound on Δ does not help the parties in any direct way to assess the actual time (e.g., by locally counting time intervals of length Δ), as participation gaps can invalidate their local timer with respect to the implicit global execution-driven time.

Our contributions. We present the first provably secure approach to global clock synchronization in the dynamic participation setting from standard Nakamoto-style consensus assumptions. We focus on PoS—where we assume the same honest stake-majority underlying existing PoS protocols [4]—but we expect that our techniques can be adapted to the PoW setting. To this direction, we devise the first participation-unrestricted (i.e., without explicitly known participation bounds; see below) PoS blockchain that does not need a global clock and, instead, enables parties to implement and maintain a bounded-drift clock in the delayed-delivery network model [33,19]. Our new blockchain protocol avoids the dependency on an external service providing access to global time, which constitutes an improvement in resiliency compared to previous works. Additionally, our protocol can be used as a cryptographically secure synchronizer by any external application in this new era of dynamic participation.

In more detail, our construction assumes that the inaugural parties, i.e., those that initially commence the protocol execution (but might later-on sleep or drop out), have access to local clocks indicating their local times which might be off by a parameter Δ and which advance at (roughly) the same speed. This is similar in spirit to the guarantee offered by the timing model of Kalai et al. [22]; however, to guarantee full compatibility with past (G)UC statements—including universal composability—we introduce a new, imperfect version of the clock functionality [24] as a global setup, which captures that parties advance at roughly the same pace but, unlike previous attempts to capture synchrony in a composable framework [31,21,3], allows parties to advance to a next round even before every honest party has finished with his current round, as long as front-running parties do not drift too far ahead. We believe that this global-UC formulation of relaxed synchronization can be of independent interest.[7]

Our protocol guarantees that parties who later join the protocol, no matter how outdated their local clock value is, will synchronize themselves (up to a small drift depending on network delay) with the rest of the parties, and remain synchronized for as long as they faithfully execute the protocol.

Although we conduct our formal analysis in the UC framework (where we model certain setup functionalities as being globally accessible) [9,10], we also provide informal description and analysis of our protocols for readers not familiar with the framework. Note that the advantage of UC analysis is that it ensures that the resulting synchronizer can be used to enable the design of synchronous protocols—which enjoy the benefits sketched above—from our arguably weak assumptions of similar speed local clocks and a bounded-delivery network. In fact, as the protocol exports the clock it implements to all participants, a party wishing to only use it for synchonization (for a separate application) can simply join the protocol *in a passive mode*—without any need to have stake or participate in the lottery as a slot leader—and keep executing it for a sufficiently large number of (locally clocked) steps. By doing so, the party will compute a clock that, as we prove, is

[7] We note that there are UC realizations of other "clocks" in the literature, such as [11]. This is an unfortunate name-clash of different concepts: That clock is a much weaker functionality that does not guarantee even a relaxed (bounded-drift) round-based structure with guaranteed termination.

guaranteed to be in (loose, up to a small constant offset) synchronization with all honest parties running the protocol (in a passive or full-participation mode) as long as (available) honest stake majority is preserved.

Our construction builds on Ouroboros Genesis [4], where to signify the connection, we refer to our protocol as *(Ouroboros) Chronos*. Similar to [4] we work in the dynamic-availability framework which implies that our solution works independently of the exact number of honest parties who are around, as long as they hold a majority of the system's (currently available) stake. The design and analysis of Chronos uses, in a white-box manner, elements from previous PoS protocols [35,6,26,14,4], but involves several new ideas which are crucial in achieving global synchronization in this participation-unrestricted setting. We discuss these ideas in detail below, where we give an overview of our design. (We include details and points of comparison especially with the most recent prior work [4] in Section C.)

Overview of our techniques. The heart of our protocol, namely its procedure for synchronization of newly joining parties, starts with these parties listening on the network for some time, collecting broadcasted chains and following a "densest chain" chain-selection rule. Informally, this rule mandates that if two chains $\mathcal{C}$ and $\mathcal{C}'$ start diverging at some time τ—according to the reported time-stamps in $\mathcal{C}$ and $\mathcal{C}'$—then prefer the chain which is denser in a sufficiently long interval after that time. Our first key observation is that this rule offers a useful (albeit in itself insufficient) guarantee in our setting: the joining party will end up with some blockchain that, although *arbitrarily long*, is at worst forking from a chain held by an honest and already synchronized party by a bounded number of blocks (equal to a security parameter) with overwhelming probability. This observation is the key to start building our synchronization mechanism. More concretely, we prove that the above process guarantees to eventually prune-off all chains with bad prefixes, i.e., prefixes that do not largely coincide with the prefixes of the other already synchronized honest parties' chains. In fact, as we show, the parties can compute an upper bound on the time (according to their local clocks) they need to remain in the above self-synchronization state before they build confidence in the above guarantee, i.e., before they know that their locally held chain is consistent with a long and stable prefix that already-synchronized honest parties adopt.

The second key observation is that once a joining party has converged to such a *fresh*—i.e., produced after the joining party was activated—prefix of an honest chain, it may use the difference between its current local time and the (local) time recorded when this chain (and other control information) was received to adjust its local clock so that its local time is consistent with the times reported on the prefix. The hope would be that a clever adjustment will bring this local time sufficiently close to that of an honest and already synchronized party.

Designing and analyzing such a natural updating process is unexpectedly challenging. To see why, consider the following naïve attempt: The party resets its local clock so that the time reported in, say, the last block of the prefix is the time this block was received. Before discussing the limitations of this proposal, let us first discuss an inherent property when dealing with clock synchronization in the setting with Δ-bounded (but adversarially controlled) delay networks. A message received by a party might have been sent up to Δ rounds before, hence the time that the party will set its clock to might be up to Δ rounds away from the clock of the sender (at the point of update). This delay-induced imprecision is unavoidable, so when we assess a given proposal we accept that clocks only need to be "loosely" synchronized; specifically, clocks of honest parties might differ by a bounded amount, where the bound is known and depends only on Δ. In fact, this relaxation is common and believed to be necessary even in the permissioned model [27,20].[8]

However, the above simple solution is problematic even when no delays are there: Although the chain that the newly joining party recovered is guaranteed to have a prefix consistent with the already synchronized honest parties, individual blocks might be originating from the adversary and therefore contain a time stamp very different from the true sending time of that block. To make matters worse, the rate of honestly generated blocks in a chain of an honest party can be quite low as implied by the known bounds of chain quality [19,14], and thus the time inaccuracy of any individual block can be significant.

A second attempt would be to have in every round (or at regular intervals) every party use the credentials of all the coins it owns to broadcast a signed timestamp, i.e., every party acts as a verifiable *synchronization*

[8] The model from [27] with honest clocks that report values differing by up to Δ is equivalent to a situation in which clocks report the right value, but parties might receive it with a difference of up to Δ rounds.

(or timestamping) beacon on behalf of all the coins it owns. The joining party receives all these broadcasted timestamps, and uses their majority to compute the value of its clock. Still, this solution has drawbacks: The first is scalability; this is not severe, as existing ideas can be employed such as using the protocol history as input to a verifiable random function (VRF) to identify eligible parties (or, as in the case of Algorand, by using Bracha-style committees [7]) to send timestamping beacons in every synchronization round. The second, harder problem is that in order to use the majority, the local clocks of the parties that report time need to be perfectly synchronized so that their majority agrees. If their clocks have any small drift, this fails. Furthermore, even with identical speed clocks, dynamic participation allows parties to drop off and rejoin, which means that, due to the network delay the honest parties will end up with only loosely synchronized local clocks. Using the average instead of the majority function does not help out here either since a single adversarial timestamp can throw off the average arbitrarily far. Hence, taking the median of the received timestamps promises to be more stable against extreme values. Observe that as long as synchronized honest parties' local clocks are not far apart, the times they report will be concentrated to a sufficiently small time interval, and the median will fall in this interval.

The above insight brings us closer, but is still insufficient: If the adversary can serve to, say, two different joining parties different and possibly disjoint sets of timestamps (on behalf of eligible corrupted synchronization-beacon parties) then he could force an opposing clock adjustment between the two that will increase their clock drift well beyond the drift of any pair of already synchronized parties. To resolve this issue, we need to ensure that the parties agree on the set of eligible timestamps (whether honest or corrupted) that they use for adjusting their local time. This is a classical consensus problem. Luckily, our synchronizer runs in tandem with a PoS-based blockchain which solves consensus with dynamic availability, and which can assist in reaching agreement on the synchronization-beacon values for recalibration. And thanks to the property discussed at the beginning of the section—namely that even joining parties (without accurate time) will eventually be able to bootstrap a sufficiently long prefix of the blockchain—the joining parties will agree on the set of beacons for recalibration.

Our technical solution follows the spirit of the final conclusion above. In a nutshell, we will use the VRF to assign timestamping-beacon parties to slots according to their state. Parties who are synchronized and active when their assigned slot is encountered will broadcast a timestamp and a VRF-proof of their eligibility for the current timeslot (together, we call this a *synchronization beacon*). And to agree on the set of eligible parties that will be used (including the dishonest ones) these beacons will also be included in the blockchain by the already synchronized parties, similarly to transactions. Any party who joins and tries to get synchronized will gather chains and record any broadcasted beacons (and keep track of the local time these were received). Once the party is confident it has a sufficiently long prefix of the honest chain, it will retrospectively use this gathered information to extract the agreed-upon set of beacons, compute a good approximation of the clocks parties had when they broadcasted these beacons and apply a median rule to set its local clock to at most a small distance from other honest and synchronized parties. In order to ensure that already synchronized parties adjust in tandem with joining parties we will have them also periodically execute the synchronization algorithm—but of course using their local blockchain, which they know is guaranteed to have a large common prefix with any other honest and synchronized party. Evidently, to turn this high-level idea of our solution into a provably secure protocol requires careful design choices that we present in Section 3. The analysis is given in Section 6. This constitutes the main technical contribution of this paper.

Outline. Section 2 sketches our model and Section 3 describes our main result, i.e., outlines the Chronos protocol and the functionality it realizes, along with the security analysis. These sections are written in a self-contained manner so that the reader can get a first idea of our contributions and techniques. Sections 4-6 include more details on our results and can help an interested reader to get an overview of the detailed treatment that is presented in the Appendix.

2 Our Model

Basic Notation. For $n \in \mathbb{N}$ we use the notation $[n]$ to refer to the set $\{1, \ldots, n\}$. For brevity, we often write $\{x_i\}_{i=1}^n$ and $(x_i)_{i=1}^n$ to denote the set $\{x_1, \ldots, x_n\}$ and the tuple $(x_1, \ldots, x_n)$, respectively. For a tuple

$(x_i)_{i=1}^n$, we denote by $\mathsf{med}((x_i)_{i=1}^n)$ the (lower) median of the tuple, i.e., $\mathsf{med}((x_i)_{i=1}^n) \triangleq x'_{\lceil n/2 \rceil}$, where $(x'_i)_{i=1}^n$ is a (non-decreasing) sorted permutation of $(x_i)_{i=1}^n$.

For a blockchain (or chain) $\mathcal{C}$, which is a sequence of blocks, we denote by $\mathcal{C}^{\lceil k}$ the chain that is obtained by removing the last k blocks; and by $\mathtt{head}(\mathcal{C})$ the last block of $\mathcal{C}$. We write $\mathcal{C}_1 \preceq \mathcal{C}_2$ if $\mathcal{C}_1$ is a prefix of $\mathcal{C}_2$.

Dynamic Availability. We adopt the dynamic availability framework from [4] which captures parties joining and leaving the protocol at (the environment's) will. This is done by equipping the functionalities, global setups, and the protocol with explicit registration/de-registration commands, thereby keeping track of when parties are joining and adjusting their guarantees depending based on this information. We refer the reader to [4] for a detailed discussion of this model.

Relaxed Synchrony. The synchrony assumption that parties advance at exactly the same pace can be captured by the global-setup variant of the clock functionality from [24]. This is a weaker version of the global clock used in previous composable analyses of blockchains [5,4] in that it does not keep a counter representing the global system time, but rather maintains for each party (resp. ideal functionality) an indicator-bit d_{P} (resp. $d_{(\mathcal{F},\mathrm{sid})}$) of whether or not a new round has started. Each party's indicator is accessible by a standard CLOCK-GET command. All indicators are set to 0 at the beginning of each round; once any party or functionality finishes its round it issues a CLOCK-UPDATE command that updates his indicator to 1. Once every party and functionality has updated its indicator, the clock resets all of them to 0; this switch allows the parties to detect that the previous round has ended and move on to the next round.

Arguably the above clock offers very strong synchronisation guarantees, since once a round switches, every party is informed about it in the next activation. In [24] a relaxed version of this clock was introduced which allowed the adversary to delay notifying the parties about a round switch by bounded amount of fetch-attempts. This behavior relaxes the perfect nature of the clock, but it still ensures that no party advances to a next round before all parties have completed their current round.

In this work we consider parties that advance at roughly the same speed, which means that a party might advance its round even before another party has finished with its current round, and even multiple times, as long as its is ensured that no honest party is left too far behind. For this purpose we introduce an even more relaxed version of the (global-setup variant) of the clock from [24] which, intuitively, allows a party to advance to its next round multiple times *before* some honest parties have completed their current round, as long as the relative pace of advancement for any two honest parties stays below a drift parameter Δ_{clock}. We note in passing that a similar guarantee was formulated in the timing model [23]; however, the solution there notified the underlying model of computation which creates complications with the (G)UC composition theorem which would need to be reproved. To avoid such complications, in this work we capture the above relaxed synchrony assumption as a global functionality.[9]

The above is captured as follows: Similar to the perfect clock above, the imperfect clock stores an indicator-bit d_{P} which is used to keep track of when everyone has completed a round (not necessarily the same round)—one can think of this indicator as corresponding to a baseline round-switch, which is however hidden from the parties and might only be observed by ideal functionalities. Additionally, for every party the imperfect clock keeps an imperfect version of the indicator bit d_{P}^{Imp} (corresponding to switches P's *local*, e.g., hardware, clock switches) which is what is exported when the party attempts to check his clock.

This local indicator is used similarly to how synchronous protocols would use the perfect indicator in [24]; but we allow the adversary to control when this local indicator is updated under the restrictions that (a) d_{P}^{Imp} cannot advance in the middle of P's round, (b) it cannot fall behind the baseline induced by the indicator d_{P}, and (c) it cannot advance ahead of the baseline by more than Δ_{clock}. This is achieved by the imperfect clock keeping track of the relative difference/distance $\mathtt{drift}_{\mathsf{P}}$ between the number of local advances of each registered P from the baseline updates; this distance is increased whenever d_{P}^{Imp} is reset (by the adversary) to 0 and decreased whenever the baseline indicator $d_{\mathsf{P}} \in \{0,1\}$ is reset to 0; if the distance of some party

[9] In [24] a functionality corresponding to the timing-model assumptions [23] was proposed along with a reduction to the (local) clock functionality. However, both the fact that their clock functionality is local and that their reduction uses a complete network of (known) bounded-delay authenticated channels—which we do not assume here—makes that result incompatible with our model and goals.

from the baseline falls below 0 (i.e., the adversary attempts to stall a party when the baseline advances[10]) then the local indicator is reset to $d_{\mathsf{P}}^{Imp} = 0$ (which allows P to advance his round) and the corresponding distance is also reset to 0.

Modeling Peer-to-Peer Communication. We assume a diffusion network in which all messages sent by honest parties are guaranteed to be fetched by protocol participants after a specific delay Δ_{net}. Additionally, the network guarantees that once a message has been fetched by an honest party, this message is fetched by any other honest party within a delay of at most Δ_{net}, even if the sender of the message is corrupted. We note that this network model is not substantially stronger than in previous works [5,4], which use a network functionality providing bounded-delay message delivery. Our model is equivalent via an unconditional reduction: echoing received messages. In practice, this reduction of course needs to be applied prudently to avoid saturating the network. This is exactly done by the relevant networking protocols: e.g. in Bitcoin, when a new block is received its hash is advertized and then propagated and validated by the network as needed. Chronos can use the same mechanism. We detail the corresponding functionality in Section B.

Genesis Block Distribution; the Weak Start Agreement. Our model allows parties' local time-stamps to drift apart over the course of an execution; additionally the model makes no assumption that the initialization of the initial stakeholders is completed in the same round, i.e., honest parties might start producing blocks for logical slot 1 in different rounds of the (global) execution. To this aim, we weaken the functionality $\mathcal{F}_{\mathsf{INIT}}$ adopted by [4] to allow for bounded delays when initial stakeholders receive the genesis blocks. Namely, our $\mathcal{F}_{\mathsf{INIT}}^{\Delta_{\text{net}}}$ functionality merely guarantees genesis block delivery to initial stakeholder not more than Δ_{net} rounds apart from each other; the offsets are under adversarial control. The details of the $\mathcal{F}_{\mathsf{INIT}}^{\Delta_{\text{net}}}$ functionality appear in Section B.

Further Hybrids. The protocol makes use of a VRF (verifiable random function) functionality $\mathcal{F}_{\mathsf{VRF}}$, a KES (key-evolving signature) functionality $\mathcal{F}_{\mathsf{KES}}$, and a (global) random oracle functionality $\mathcal{G}_{\text{RO}}$ (to model ideal hash functions). Full model details are provided in Section B for reference.

3 The Synchronizing Blockchain

In this section we provide our main result: a concrete blockchain protocol, called *Chronos* (meaning "time" in greek), implementing a new functionality that extends the ledger from [4] with a robust notion of time. Informally, in addition to the standard ledger guarantees, this new ledger exports a notion of global time with strong accuracy guarantees.

In a nutshell, our new "timed" ledger functionality maintains an immutable ledger state denoted by `state` which encodes a sequence of transaction blocks. How fast this state grows and which transactions it includes are specified in a ledger policy description, which is a ledger parameter. Each registered party can request to see the state, and is guaranteed to receive a sufficiently long prefix of it—the length of each party's visible prefix of the state at any given point is captured by (monotonically) increasing pointers that point to the last block this party's current prefix includes. The adversary is given policy-limited control to set these pointers within a sliding of width `windowSize`, which starts at the head of the state and slides as the state increases. The above mechanism captures the guarantees of the *common-prefix (CP)* property of blockchains (where the common-prefix parameter relates to the width of the window).

Parties advance the ledger when they are instructed to (activated with specific maintain-ledger input) by their UC environment. The ledger uses these queries plus the knowledge of the baseline speed to ensure the advancement of the state (which encapsulates a property analogous to *chain growth* (CG) [18,33]). Per the ledger policy, any party can input a transaction to the ledger, which are validated and guaranteed to be included eventually whenever they stay valid (this will ensure transaction liveness [18]). The ledger gives different guarantees to honest parties depending on whether they are considered synchronized or not. Roughly speaking, synchronized parties are the parties for which one can provide the best security guarantees (for example, they are inaugural, or part of the system for a long time and correctly bootstrapped).

[10] Note that, by definition the baseline advances when all parties have completed their current round.

De-synchronized honest parties receive reduced security guarantees and could, until they become synchronized, have unreliable values.

In our ledger description, the above policy is specified by means of a ledger-parameter/procedure which is denoted as ExtendPolicy. The basic mode of operation of ExtendPolicy is as follows: it takes, as an input, a proposal from the adversary for extending the state, i.e., a sequence of new blocks to be added, and can then decide to follow this proposal if it satisfies its policy sketched above; if it does not, ExtendPolicy can ignore the proposal (and enforce a default extension). It will enforce minimal chain growth, a certain fraction of "good blocks," (a.k.a. *chain quality* (CQ) [18,33]) and transaction liveness guarantees for old and (still) valid transactions. In fact, as discussed below, in the security proof of Chronos, we will demonstrate that it satisfies the above properties, CP, CG, and CQ for a set of appropriate parameters, and will use this to derive a UC simulator that respects the restrictions of our ledger's ExtendPolicy with the same parameters.

The above policy of the new ledger is similar to the one from [4,5]. What differentiates the new Chronos ledger is an *exportable approximate-time extension*: Every party P has an associated timestamp t—which will correspond to this party's local time in Chronos. t is adjusted in two ways: (1) in a monotonically increasing manner whenever the local (baseline) clock (of the party) advances, and (2) in a non-monotonic manner (discussed below), at epoch boundaries. (The beginning and the end of epochs is according to the party's local time t—an epoch switch occurs whenever the local time increases by R_L from the (local) timestamp of the previous epoch.) In fact, our ledger records an *extended* local timestamp $\mathtt{time}_{\mathsf{P}}$ for each party P, which is the pair $\mathtt{time}_{\mathsf{P}} = (e, t)$, where t is as above, and e is the number of non-monotone adjustments to t, i.e., the number of epoch switches that P has observed. Note that due to the potential adversarial clock and network influence combined with dynamic participation, we cannot realize a ledger with perfect, monotonically and consistently increasing timestamps. Therefore we weaken the timestamp updates guarantee to allow the adversary to apply a limited shift s, within some bounding parameter $\mathsf{shiftLB} \leq s \leq \mathsf{shiftUB}$, to the non-monotone adjustments that are triggered by epoch-switches. Nevertheless, the ledger enforces that any two alert parties with respective timestamps (e, t) and (e', t'), satisfy the constraints $|t - t'| \leq \mathtt{timeSlack}_{\mathrm{total}}$ and $|t - t'| \leq \mathtt{timeSlack}_{\mathrm{ep}}$ if $e = e'$, and $|e - e'| \leq 1$ for the respective clock parameters $\mathtt{timeSlack}_{\mathrm{ep}}$, $\mathtt{timeSlack}_{\mathrm{total}}$ that define the maximally allowed skewness of parties. Note that this gives the possibility that within an epoch the slack could be potentially different (i.e., much better) than when comparing two timestamps across epochs.

Our main result is stated in the following. We stress that this is a simplified version of the concrete theorem we actually prove, with some of the parameters left implicit. Our proof actually shows that we can obtain security for a much more concrete set of parameters, reflected fully in Theorem 7. Note that such precision concrete-security statements are crucial for the analysis of Nakamoto-style blockchains as one needs to take decisions such as "how many confirmations are sufficient to ensure a given block cannot be inverted?"

Theorem. *Our blockchain synchronizer realizes the ledger functionality described above and achieves the clock parameters*

$$\mathsf{shiftLB} = -2\Delta; \qquad \mathsf{shiftUB} = \Delta;$$
$$\mathtt{timeSlack}_{\mathrm{total}} = 2\Delta; \qquad \mathtt{timeSlack}_{\mathrm{ep}} = \Delta,$$

where the parameter R_L is chosen sufficiently large w.r.t. to the security parameter and $\Delta = \Delta_{\mathrm{net}} + \Delta_{\mathrm{clock}}$. The parameters of the ledger and of ExtendPolicy are instantiated based on the concrete *CP, CG, and CQ guarantees of our blockchain synchronizer.*

The ledger functionality is fully specified in Section D. A glossary of all parameters of the ledger is provided in Section M.2. The full UC-realization statement with all parameters concretely instantiated (which is the last step in our sequence of proofs) is given in Section L.

Using Chronos as a Black-box Synchronizer. Cryptographic protocols can use the abstracted clock and make use of the provided timestamps. When $\mathtt{timeSlack}_{\mathrm{ep}} = \mathtt{timeSlack}_{\mathrm{total}} = 0$, and $\mathsf{shiftLB} = \mathsf{shiftUB} = 0$ we obtain an equivalent formulation of the global time/clock of previous works. If shifts are to be expected but slack is still zero, then a protocol must anticipate irregularities at exactly the epoch boundaries. If in addition the skew is not zero, then a higher-level protocol must be further resilient to parties working with

slightly inaccurate time values. Since the irregularities are very predictable and bounded, it is possible to propose generic countermeasures for protocols that need more accurate clocks. For example, a protocol can stall the operation at epoch boundaries and resume at a fixed later time in the new epoch. Additionally, the duration of one logical round of the higher-level protocol can be enlarged to absorb the skew. By the limited shift, the bounded skew, and the guaranteed advancement of the timestamps one can derive concrete guarantees for liveness. Finally, when working in more optimistic (i.e., less adversarial) network models, the time-stamps can be further adjusted to improve accuracy assuming the network parameters (such as expectation and deviation of delay) are known. We give more details in Section E.

3.1 Our Protocol

The protocol we present is a Nakamoto-style proof-of-stake based protocol for the so-called semi-synchronous setting; this is the same model used for standard analyses of Bitcoin. In this model, parties have a somewhat accurate common notion of elapsed time (rather than absolute time information) and the network has an upper bound on the delay which is not known to the parties. At a very high-level the protocol attempts to imitate a process which resembles a situation in which state (including time) is continuously passed on to currently alert stakeholders. The honest majority of active stake assumption that is explicit in [14,4] will then ensure that the adversary cannot destroy this state by using his ability to tune participation.

To ease into the main protocol ideas it is useful to imagine a situation in which there is a core of parties with sufficient stake that has been around from the onset of the blockchain. (These parties have a common, albeit somewhat imperfect, understanding of how much time has passed since the protocol started and can contribute this information to the synchronization procedure.) We stress that the continuous or indefinite presence of such parties is not needed in our final protocol which will ensure that the information that these parties would safeguard is passed on to new parties if/when such inaugural parties go to sleep or deregister.

Here is how such an inaugural participant (i.e., a participant who is assigned stake at the outset of the computation by $\mathcal{F}_{\mathsf{INIT}}$) executes the protocol. With access to the provided genesis block, which reveals an initial record $\mathbb{S}_1 = \big((\mathsf{P}_1, v_1^{\mathrm{vrf}}, v_1^{\mathrm{kes}}, s_1), \ldots, (\mathsf{P}_n, v_n^{\mathrm{vrf}}, v_n^{\mathrm{kes}}, s_n)\big)$ that associates each participant P_i[11] to its chosen public keys used for verification purposes of the staking process and its initial stake s_i, each party begins the so-called first epoch of the staking procedure and sets its local clock `localTime` to the value 1. The party has to execute a certain set of tasks per round. Note that two inaugural parties have only a somewhat accurate notion of elapsed time and receiving the genesis block might be delayed, it might very well be that a party P_1 has executed three rounds, while P_2 has only executed one so far, or has not even received the genesis block. The bounds on the clock drifts and the network delay however ensure that the difference of the number of completed protocol rounds does not drift too far apart.

A participant's main task (per round) is to evaluate whether it is elected to produce a block for the current local time, which we refer to as a *slot*. For this, it evaluates a verifiable random function (VRF) on input $x := \eta_1 \,\|\, \texttt{localTime} \,\|\, \texttt{TEST}$, where η_1 is a truly random seed provided by $\mathcal{F}_{\mathsf{INIT}}$. If the returned value y is smaller than a threshold value $T_{\mathsf{P}}^{\mathsf{ep}}$, which is derived from the stake associated with P, then the participant is called a slot leader. The threshold is computed to yield a higher probability of slot leadership the higher the stake of the party. The main task of the slot leader is to create a valid block for this slot that contains, as control information (alongside the transactions), the VRF proof of slot leadership, an additional random nonce, and the hash to the head of the chain it connects to. Each block is signed using a key-evolving signature scheme.[12] As typical in these systems, the block is made to extend (essentially) the longest valid chain known to the party. Due to the slightly shifted local clocks, some care has to be taken to not disregard entirely chains that contain blocks in the logical future of a party. However, the chain a party adopts (and computes the ledger state from) at slot `localTime` shall never contain a block with a higher time-stamp.[13]

[11] More precisely, P_i denotes just a bitstring in the model that formally identifies a machine and is used to identify which keys (and hence stake) are controlled by corrupted machines. Note that we write participant or party instead of machine.

[12] The KES ensures that if a participants gets corrupted, no blocks can be created in retrospect.

[13] Some further care has to be taken in proof of stake to detect chains that try to perform a long-range attack. We describe this in the next section in more detail when we recall the Genesis chain-selection rule.

In addition to the above actions, or if a party is not slot leader, it must play the lottery once more on input $x' := \eta_1 \,\|\, \texttt{localTime} \,\|\, \texttt{SYNC}$. If the party is lucky this time and receives a return value smaller than the threshold (defined shortly), it must emit a so-called synchronization beacon containing the VRF proof and the current time `localTime`. Synchronization beacons are treated similarly to transactions and are contained into blocks if valid. If a party has done all its tasks, it increments `localTime` and waits until the round is over. Except for the generation of synchronization beacons, which is only done in a first fraction of an epoch, the above round procedure iterates over the entire first epoch, where the length of an epoch is R, a parameter of the protocol. Our security proof shows that this first epoch does result in a blockchain satisfying common prefix, chain growth, and chain quality properties for specific parameters, as long as the leader-election per slot is to the advantage of honest protocol participants.[14]

At the epoch boundary to the second epoch, two important things happen. First the stake-distribution and the epoch randomness change: they are derived from specific blocks contained in the guaranteed common prefix established by the first epoch. In particular, we must ensure that at the time the stake distribution is fixed, the epoch randomness cannot be predicted to ensure the freshness of the slot leader election lottery for the second epoch. The second critical update at the epoch boundary is the local time: each party performs a local-clock adjustment, outlined in Section 5.1, which ensures that after the adjustment parties are still close together, where "close" means within $\Delta = \Delta_{\text{net}} + \Delta_{\text{clock}}$ (two sources of bounded variance contribute to this: delay and drift) and that performed shifts of the local clock remain small (which is crucial for security). The desired property follows from the common-prefix guarantee (enabling an agreement on beacons), the honest majority assumption (enabling small clock shifts), and the network properties and clock properties (which ensure correlated arrival times). With some additional considerations detailed in Section 5.1, the protocol proceeds executing the above round tasks for the entire second epoch until the next boundary is met. This iterated process, where one epoch bootstraps the next, is backed by an inductive security argument, following previous works [4,14,26], that shows how the overall security is a consequence of the first epoch achieving the desired blockchain properties to serve as a good basis for the second, etc.

The reason to perform a local-clock adjustment is to enable the main goal of our construction: to enable new parties to safely join the system and to determine, just by observing the network and without any further help, an accurate and up-to-date local-clock value and ledger state with respect to the existing honest parties in the system, i.e., being within a Δ interval of their clock values and obtaining the same common-prefix, chain-quality and chain-growth guarantees. After this, newly joining parties can start contributing to the security of the system.

The bootstrapping procedure for newcomers is quite involved due to a combination of obstacles: First, the joining party needs to obtain a blockchain that shares some common prefix with the common prefix established by the existing parties. This is achieved by having the joining party listen to the network for some rounds, and picking the "best" chain $\mathcal{C}$ it sees in the following sense: when compared with any other seen valid chain $\mathcal{C}'$, $\mathcal{C}$ contains more blocks in an interval of slots of size s starting from the forking point of $\mathcal{C}$ and $\mathcal{C}'$. We prove that based on the honest-majority assumption, such a densest chain must share a large common prefix with the chains honest parties currently hold. However, $\mathcal{C}$ could still be adversarially crafted and for example be much longer than what honest parties agreed on by extending into the future, hence a reliable ledger state cannot yet be computed. However, it will become possible once the joining party succeeds in bootstrapping also an accurate time-stamp in the Δ interval of honest participants' timestamps, which is the second obstacle to overcome. After the party is guaranteed to be hooked to a large prefix of the honest parties' common-prefix, it begins recording all synchronization beacons it receives on the network for a long enough period of time, a parameter of the system. The length of the waiting time is set in order to ensure that, after the newly joining party started listening to the network, the parties at least once seeded the slot-leadership lottery with a fresh nonce that was unpredictable at the time of joining the system. After an additional waiting time, the agreed-upon set of beacons (with proofs referring to the fresh lottery) will be part of the common prefix and eventually be part of what is known to the joining party. We prove that based on this agreement on beacons found in the blockchain, the clock-adjustments procedure by the current

[14] We note that the leader election is per logical slot and honest parties will all pass through the same logical not at the same time, but at related times.

participants in the system can be retraced and will yield a clock adjustment to the newly joining party's local clock that will directly push it into the interval of existing honest participants' local clock. At this point, the party runs the normal chain-selection mechanism, essentially cutting off blocks in its logical future and obtains a reliable ledger state as well.

3.2 Outline of the Security Arguments

To handle the proof complexity, we divide the proof logically into a sequence of steps. Here we give a very high-level motivation of this modular approach with pointers to the technical sections. Later in Section 6, we give an in-depth introduction to the more concrete security statements.

We first analyze the protocol only for a single epoch in a rather static world, where parties do not disappear. We focus on proving the traditional CP, CG, and CQ guarantees of the protocol for the first epoch. This is needed to establish that the underlying blockchain protocol, Ouroboros Genesis, is able to tolerate small inaccuracies in terms of time-stamps. To be more precise, the only modification that would be needed at this point is that parties buffer future chains and consider them later. Once we can rely on the blockchain properties, we can as a second step analyze the synchronization procedure and prove that no matter what the adversary does, the parties will always stay close together when transitioning from one epoch (i.e., the first) to the next and the logical clock-adjustments are very small. This establishes the base case for a greater inductive argument to establish the full security over the lifetime of the protocol in a static world. The exact sequence of arguments is given in Section 6.4 and the formal theorems and the proofs are given in the technical supplement, Sections F to H.

The first step in moving to the more dynamic world we are aiming at is the analysis of the joining procedure. A party joining the network acts like an observer of the network (i.e., it does not interfere with the protocol) and becomes synchronized after extracting enough information from the network, at which point it can start to be an active protocol participant. This step of the security proof can hence be conducted based on the previous analysis. This is detailed in Section 6.5 and in the technical supplement, Section I provides the formal theorems and proofs.

At this point, we are ready to derive the CP, CG, and CQ guarantees for the entire protocol in a fully dynamic world, where parties join any time, might be temporarily stalled, and disappear unannounced. This can be argued based on a case distinction on different party types (cf. Section 4) and quantify their impact on the security guarantees established above. As it turns out, the most crucial aspect is the joining of new parties, whose security we already understand at this point. This theorem and its proof are given in the technical supplement, Section J.

Moving towards the UC proof, we first observe that the proven chain-growth guarantees (coupled with CP and CQ) do not directly imply liveness of the ledger as in previous works. The reason is that the parties, due to the time-adjustments at the end of an epoch, could (from the viewpoint of an outside observer) move slower than what the observer perceives as the *nominal time* advancement derived from the baseline speed, i.e., a more objective notion of time. For this reason, we prove a concrete relationship between the reported time-stamps by parties and such a reference time. The definition of the reference time and how to connect it to liveness is discussed in Sections 6.3 and 6.7 respectively, the formal statements with proofs are given in technical Section K.

Putting everything together, we can finally instantiate all ledger parameters and describe the simulator. The clue here is that writing down the simulator at this point is conceptually simple: the simulator can in principle emulate the real-world execution perfectly. As long as the ledger parameters are chosen correctly based on the CP, CQ, and CG parameters established above in the fully dynamic world, only a violation of the properties can prevent a correct simulation. This is the topic of the technical Section L.

4 Dynamic Participation Model

To support a fine-grained dynamic participation model, we follow the approach of [4] and categorize the parties into *party types*. Recall that the dynamic participation model allows to capture the security of the

Resource	Basic types of *honest* parties Resource unavailable	Resource available
random oracle $\mathcal{G}_{\text{RO}}$	*stalled*	*operational*
network $\mathcal{F}_{\text{N-MC}}$	*offline*	*online*
clock $\mathcal{G}_{\text{PerfLClock}}$	*time-unaware*	*time-aware*
synchronized state, local time	*desynchronized*	*synchronized*
KES capable of signing (w.r.t. local time)	*sign-capable*	*sign-uncapable*

Derived types:

$$\textit{alert} :\Leftrightarrow \textit{operational} \wedge \textit{online} \wedge \textit{time-aware} \wedge \textit{synchronized} \wedge \textit{sign-capable}$$

$$\textit{active} :\Leftrightarrow \textit{alert} \vee \textit{adversarial} \vee \textit{time-unaware}$$

Note: *alert* parties are honest, *active* parties also contain all adversarial parties.

Fig. 1. Party types.

protocol in a realistic fashion, by considering that some parties might be stalling their computation, some might accidentally lose network access and hence disappear unannounced, and others might lose track of the passage of time due to some failure. In our model, we formally let the environment be in charge of connecting and disconnecting to its resources: The various basic and derived types of parties are summarized in Figure 1.

For a given point in execution, a party is considered *offline* if it is not registered with the network, otherwise it is considered *online*. A party is *time-aware* if it is registered with the clock, otherwise we call it *time-unaware*. We say that a party is *operational* if it is registered with the random oracle, otherwise we call it *stalled*. Finally, we say that a party is *sign-capable* if the counter in $\mathcal{F}_{\mathsf{KES}}$ is less or equal to its local time-stamp.

Additionally, an honest party is called *synchronized* if it has been continuously connected to all its resources for a sufficiently long interval to make sure that, roughly speaking, (i) it holds a chain that shares a common prefix with other synchronized parties (synchronized state) and (ii) its local time does not differ by much from other synchronized parties (synchronized time). Our protocol's resynchronization procedure JoinProc will guarantee the party that after executing it for the prescribed number of rounds, it will achieve both properties (i) and (ii) above. In addition, such a party will eventually become sign-capable in future rounds (in case the KES is "evolved" too far into the future due to a de-synchronized time-stamp before joining). We note that an honest party always knows whether it is synchronized or sign-capable and (in contrast to the treatment in [4]), it maintains its synchronization state in a local variable `isSync` and makes its actions depend on it.

Based on these four basic attributes, we define *alert* and *active* parties similarly to [4]. Alert parties are considered the core set of honest parties that have access to all necessary resources, are synchronized and sign-capable. On the other hand, *potentially active* parties (or *active* for short) are those (honest or corrupted) parties that can potentially act (propose a block, send a synchronization beacon) in its current status; in other words, we cannot guarantee their inactivity. Formally, it includes alert parties, corrupted (i.e., adversarial) parties, and moreover any party that is time-unaware (independently of the other attributes; this is because those parties are in particular not capable of evolving their signing keys reliably and hence it cannot be excluded that if they later get corrupted, they might retroactively perform protocol operations in a malicious way).

The definition of a party type is extended from a single point in an execution to a logical slot in a natural way as follows: a party P is counted as alert (resp. operational, online, time-aware, synchronized, sign-capable) for a slot `sl` if the first time its local clock passes through the (logical) slot `sl`, it maintains this state *throughout the whole slot*, otherwise it is considered not alert (resp. stalled, offline, time-unaware, desynchronized, sign-uncapable) for `sl`. It is considered corrupted (i.e., adversarial) for `sl` if it was corrupted by the adversary $\mathcal{A}$ when its local clock satisfied $\texttt{localTime} \leq \texttt{sl}$. Finally, it is active for `sl` if it is either corrupted for that slot, or it is alert or time-unaware *at any point* during the interval when its local clock for the first time passes through slot `sl`.

5 Protocol Details

We provide a more in-depth overview of our new protocol (Ouroboros Chronos) in this section and focus on the actions of an alert party (i.e., synchronized and with access to all required resources). Such a party runs the following standard round instructions:

1. Fetch information from the network (procedure FetchInformation) over which transactions, beacons, and blocks are sent. The local time is updated via a call to UpdateTime. The party locally advances its time-stamp whenever it realizes that a new (local) round has started by a call to $\mathcal{G}_{\text{ImperfLClock}}$ and comparison to `lastTick`. The code is found in Section C.6 of the technical supplement.
2. Record the arrival times of the synchronization beacons the protocol sends out (call to ProcessBeacons). This is discussed in detail in Section 5.1.
3. Process the received chains: as some chains might have been created by parties whose time-stamps are ahead of local time, the future chains are stored in the buffer `futureChains` for later usage. Among the remaining chains, the protocol will decide whether any chain is more preferable than the local chain using a chain-selection rule inspired by Ouroboros Genesis [4] (procedure SelectChain) which we thus refer to as the Genesis rule. An important property of the Genesis rule is that chain selection is secure without requiring a moving checkpoint: roughly speaking, a chain $\mathcal{C}_1$ is preferred over $\mathcal{C}_2$ if they have a large common history, except possibly the last k blocks (where k is some parameter) and $\mathcal{C}_1$ is longer. If however, they fork even before, chain $\mathcal{C}_1$ is preferred if it is block density is higher compared to $\mathcal{C}_2$ in a carefully selected interval of size s slots after the forking point. The details of all the above are described in Sections C.5, C.8 and C.9,
4. Run the main staking procedure (StakingProcedure) to evaluate slot leadership, and potentially create and emit a new block or synchronization beacon. Before the main staking procedure is executed, the local state is updated including the current stake distribution (call to UpdateStakeDist). The procedures are specified in Sections C.10 and C.6.
5. If the end of the round coincides with the end of an epoch, the synchronization procedure is executed described in Section 5.1.

The code of the entire protocol is given in the technical supplement, Section C. The code of the basic round structure is given in Section C.3 that steers the above tasks. Below we provide details on the most important aspects of the above process.

Stake distribution and leader election. A party P is an eligible slot-leader for a particular slot `sl` in an epoch `ep` if its VRF-output (for an input dependent on `sl`) is smaller than a threshold value $T_{\mathsf{P}}^{\mathsf{ep}}$. The threshold is derived from the (local) stake distribution $\mathbb{S}_{\mathsf{ep}}$ assigned to an `ep` which in turn is defined by the (local) blockchain $\mathcal{C}_{\mathsf{loc}}$, that is we assume an abstract mapping that assigns to a party (identified by an encoding of its public keys) its stake derived as a function of the transactions in $\mathcal{C}_{\mathsf{loc}}$, the genesis block, and the epoch the party is currently in. As described above, the stake distribution is only updated once a party enters a new epoch, i.e., once $\texttt{localTime} \mod R = 1$. Say a party enters in epoch $\texttt{ep}+1$, then the distribution is defined by the state contained in the block sequence up to and including the last block in epoch $\texttt{ep}-1$ (or the genesis block for the first two epochs). Furthermore, the epoch randomness for epoch $\texttt{ep}+1$ (to refresh the lottery) is extracted from the previous randomness and the seeds defined by the first two-thirds of the blocks in epoch `ep` (for the first epoch, the randomness is defined by the genesis block). Both of these updates thus derived based on the (supposedly) established common prefix among participants.

The relative stake of P in the stake distribution $\mathbb{S}_{\mathsf{ep}}$ is denoted as $\alpha_p^{\mathsf{ep}} \in [0,1]$. The mapping $\phi_f(\cdot)$ is defined as

$$\phi_f(\alpha) \triangleq 1-(1-f)^{\alpha} \tag{1}$$

and is parametrized by a quantity $f \in (0,1]$ called the *active slots coefficient* [14].

Finally, the threshold T_p^{ep} is determined as

$$T_p^{\mathsf{ep}} = 2^{\ell_{\mathsf{VRF}}}\phi_f(\alpha_p^{\mathsf{ep}}), \tag{2}$$

where ℓ_{VRF} denotes the output length of the VRF (in bits).

Note that by (2), a party with relative stake $\alpha \in (0,1]$ becomes a slot leader in a particular slot with probability $\phi_f(\alpha)$, independently of all other parties. We clearly have $\phi_f(1) = f$, hence f is the probability that a hypothetical party controlling all 100% of the stake would be elected leader for a particular slot. Furthermore, the function ϕ has an important property called "independent aggregation" [14]:

$$1 - \phi\left(\sum_i \alpha_i\right) = \prod_i (1 - \phi(\alpha_i)) \ . \tag{3}$$

In particular, when leadership is determined according to ϕ_f, the probability of a stakeholder becoming a slot leader in a particular slot is independent of whether this stakeholder acts as a single party in the protocol, or splits its stake among several "virtual" parties.

The technical description of the staking procedure appears in Section C.10. It starts by two calls evaluating the VRF in two different points, using constants `NONCE` and `TEST` to provide domain separation, and receiving (y_ρ, π_ρ) and (y, π), respectively. The value y is used to evaluate slot leadership: if $y < T_p^{\mathsf{ep}}$ then the party is a slot leader and continues by processing its current transaction buffer to form a new block B. Aside of this application data, each block contains control information. The information includes the proof of leadership (y, π), additional VRF-output (y_ρ, π_ρ) that influences the epoch-randomness for the next epoch, and the block signature σ produced using $\mathcal{F}_{\mathsf{KES}}$. Finally, an updated blockchain $\mathcal{C}_{\mathsf{loc}}$ containing the new block B is multicast over the network (note that in practice, the protocol would only diffuse the new block B). A slot leader embeds a sequence of valid transactions into a block. As in [4], we abstract block formation and transaction validity into predicates $\mathsf{blockify}_{\mathsf{OC}}$ and $\mathsf{ValidTx}_{\mathsf{OC}}$. The function $\mathsf{blockify}_{\mathsf{OC}}$ takes as input a plain sequence of transactions and outputs a block, whereas $\mathsf{ValidTx}_{\mathsf{OC}}$ takes as input a single transaction and the ledger state. A transaction is said to be valid with respect to the ledger state if and only if it fulfills the predicate. The transaction validity predicate $\mathsf{ValidTx}_{\mathsf{OC}}$ induces a natural transaction validity on blockchain-states that we succinctly denote by the predicate $\mathsf{isvalidstate}(\mathtt{st})$ that decides that a state is valid if it can be constructed sequentially by adding one transaction at a time and viewing the already added transactions as part of the state.

Emitting synchronization beacons. An alert party emits so-called *synchronization beacons* in the first $R/6$ slots of an epoch ep. To be admissible to emit a beacon, the party evaluates the VRF again as for slot-leadership. To obtain an independent evaluation, we use a new constant called `SYNC` to obtain domain separation. If the returned value $y \leq T_{\mathsf{P}}^{\mathsf{ep,bc}}$, where in this case we can simply use a linear scaling of the domain, i.e., we define the threshold

$$T_p^{\mathsf{ep,bc}} := 2^{\ell_{\mathsf{VRF}}} \cdot \alpha_p^{\mathsf{ep}}, \tag{4}$$

then the party will create a block header and send it on the broadcast network.[15]

Embedding synchronization beacons. Part of the staking procedure is to embed synchronization beacons in the first $2R/3$ slots of an epoch ep. A synchronization beacon is embedded if the creator of the beacon was elected to emit a beacon (according to the current stake distribution in epoch ep) in the first $R/6$ slots of this epoch, and if no other beacon in the chain already specifies the same slot and party identifiers. Like this, an alert party is assured to produce a valid chain according to the validity predicate $\mathsf{IsValidChain}$ in Section C.5. Note that for a slot leader, we provide for simplicity an extra-predicate $\mathsf{ValidSB}$ in Section C.5 that allows ensuring that the extension block is valid with respect to beacon inclusion.

Running the synchronization procedure. At the end of an epoch, parties run the synchronization procedure based on the beacons recorded in this epoch. We will elaborate on this core procedure of the new protocol in Section 5.1.

[15] Note that there is no need to additionally sign a beacon. Looking ahead, for the synchronization procedure to achieve its goal, we only need agreement on the reported slot numbers (by the respectively elected parties), which is derived from the blockchain, and the guarantees provided by the broadcast functionality. Furthermore, to bound the shift that alert parties experience, it is sufficient that slot numbers reported by alert (and thus synchronized parties) are dominating and are delivered within a reasonable number of rounds after first being emitted.

5.1 The Synchronization Procedure

Our main synchronization procedure is based on several logical building blocks. We describe each of them in detail and provide the rationale behind the choices.

1.) *Synchronization* slots: Once a party's local time-stamp reaches a defined synchronization slot for the first time, it will adjust its local time-stamp before moving to the next slot. The protocol will specify the necessary actions for the cases where the local time-stamp is shifted forward or backward. We define the synchronization slots to be the slots with numbers $i \cdot R$ for $i \geq 1$ and hence they coincide with the end of an epoch. In a real-word execution (which is a random experiment with discrete steps), we say that a party P *has passed its synchronization slot* $i \cdot R$ (e.g., at step x of the experiment) if it has already concluded its operations in a round where $\mathsf{P}.\texttt{localTime} = i \cdot R$ holds for the first time. In the code, the synchronization procedure is invoked as the final step in a synchronization slot in Section C.3.

2.) *Synchronization Beacons:* In addition to the other messages, the parties in Ouroboros Chronos generate synchronization messages or "beacons" as follows: an alert party P evaluates the VRF functionality by sending $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \parallel \mathsf{P}.\texttt{localTime} \parallel \texttt{SYNC})$ to $\mathcal{F}_{\mathsf{VRF}}$ to receive the response $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$. The beacon message is then defined as the meta-data

$$\texttt{SB} \triangleq (\mathsf{P}.\texttt{localTime}, \mathsf{P}, y, \pi),$$

where $\mathsf{P}.\texttt{localTime}$ is the current slot number party P reports and the triple $(\mathsf{P}, y_\rho, \pi)$ is the usual attestation of slot leadership by party (or stakeholder) P. In the code, synchronization beacons are created in the main staking procedure in Section C.10.

3.) *Arrival times bookkeeping:* Every party P maintains an array $\mathsf{P}.\mathsf{Timestamp}_{SB}(\cdot)$ that assigns to each synchronization beacon $\texttt{SB}$ a pair $(n, \mathrm{flag}) \in \mathbb{N} \times \{\mathsf{final}, \mathsf{temp}\}$. Assume a beacon $\texttt{SB}$ with $\mathsf{slotnum}(\texttt{SB}) \in [j \cdot R + 1, \ldots, j \cdot R + R/6]$, $j \in \mathbb{N}$ and party P' is fetched by party P (for the first time). If the pair $(\mathsf{slotnum}(\texttt{SB}), \mathsf{P}')$ is new, the recorded arrival time is defined as follows:

- If P has already passed synchronization slot $j \cdot R$ but not yet passed synchronization slot $(j+1) \cdot R$, $\mathsf{Timestamp}_{SB}(\texttt{SB})$ is defined as the current slot number and the value is considered final, i.e., $\mathsf{Timestamp}_{SB}(\texttt{SB}) \triangleq (\mathsf{P}.\texttt{localTime}, \mathsf{final})$.
- If party P has not yet passed synchronization slot $j \cdot R$ (and thus the beacon belongs logically to this party's next epoch), $\mathsf{Timestamp}_{SB}(\texttt{SB})$ is defined as the current slot number $\mathsf{P}.\texttt{localTime}$ and the decision is marked as temporary, i.e., $\mathsf{Timestamp}_{SB}(\texttt{SB}) \triangleq (\mathsf{P}.\texttt{localTime}, \mathsf{temp})$. This value will be adjusted once this party adjusts its local time-stamp for the next epoch (when arriving at the next synchronization slot $j \cdot R$).

If a party has already received a beacon for the same slot and creator, it will set the arrival time equal to the first one received among those. The process to record arrival times is described in its own algorithm in Section C.7.

4.) *The synchronization interval*: the interval based on which the adjustment of the local time-stamp is computed. For a synchronization slot $i \cdot R$ $(i \geq 1)$, its associated synchronization interval is the interval $I_{\mathrm{sync}}(i) \triangleq [(i-1) \cdot R + 1, \ldots (i-1) \cdot R + R/6]$ and hence encompasses the first sixth of the epoch that is now ending.

5.) *Emitting Beacons and inclusion into the chain:* An alert party sends out a synchronization beacon during a synchronization interval (i.e., if the current local time reports a slot number that falls into a synchronization interval) if and only if the VRF evaluation $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \parallel \mathsf{P}.\texttt{localTime} \parallel \texttt{SYNC})$ to $\mathcal{F}_{\mathsf{VRF}}$ returned $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$ with $y < T_{\mathsf{P}}^{\mathsf{ep}}$ where $T_{\mathsf{P}}^{\mathsf{ep,bc}}$ is the beacon threshold in the current epoch as defined in equation 4. An alert slot leader P' on the other hand will include any valid synchronization beacon in its new block as long as $\mathsf{P}'.\texttt{localTime}$ reports a slot number within the first two-thirds of an epoch (and if the beacon has not been included yet). This process is part of the main staking procedure in Section C.10.

The remaining three steps are implemented as part of the core synchronization procedure in Section C.11.

6.) *Computing the adjustment evidence:* The adjustment will be computed based on evidence from the set $\mathcal{S}_i^{\mathsf{P}}$ that is defined with respect to the current view of P in the execution: Let $\mathcal{S}_i^{\mathsf{P}}$ contain all beacons $\mathtt{SB}$ that report a slot number $\mathsf{slotnum}(\mathtt{SB}) \in [(i-1)\cdot R+1,\ldots,(i-1)\cdot R+R/6]$ (of the synchronization interval) and which are included in a block B of $\mathsf{P}.\mathcal{C}_{\mathsf{loc}}$ that reports a slot number $\mathsf{slotnum}(B) \leq (i-1)\cdot R+2R/3$. Based on these beacons and their recorded arrival times, the shift will be computed. More precisely, if a beacon $\mathtt{SB}$ is recorded in $\mathsf{P}.\mathcal{C}_{\mathsf{loc}}$, then the arrival time used in the computation will be based on a the valid[16] beacon $\mathtt{SB}'$ that reports the same slot number and party identity as $\mathtt{SB}$ and which has arrived first—either as part of some blockchain block or as a standalone message. By our choice of parameters, parties will have assigned an arrival value to any such beacon with overwhelming probability.

7.) *Adjusting the local clock:* The shift $\mathsf{shift}_i^{\mathsf{P}}$ a party P computes to adjust its clock in synchronization slot $i\cdot R$ is defined by

$$\mathsf{shift}_i^{\mathsf{P}} \triangleq \mathsf{med}\left\{\mathsf{slotnum}(\mathtt{SB}) - \mathsf{Timestamp}(\mathtt{SB}) \mid \mathtt{SB} \in \mathcal{S}_i^{\mathsf{P}}\right\}.$$

Recall that $\mathsf{Timestamp}(\mathtt{SB})$ is shorthand for the first element of the pair $\mathsf{Timestamp}_{SB}(\mathtt{SB})$. As we will show, this adjustment ensures that the local time stamps of alert parties report values in a sufficiently narrow interval (depending on the network delay) to provide all protocol properties we need. Furthermore, for each beacon $\mathtt{SB}$ with $\mathsf{P}.\mathsf{Timestamp}_{SB}(\mathtt{SB}) = (a, \mathsf{temp})$ and slot number $\mathsf{slotnum}(\mathtt{SB}) > i\cdot R$ the arrival time is adjusted by $\mathsf{P}.\mathsf{Timestamp}_{SB}(\mathtt{SB}) \triangleq (a+\mathsf{shift}_i^{\mathsf{P}}, \mathsf{final})$. This ensures that eventually the arrival times of all beacons that logically belong to epoch $i+1$ will be expressed in terms of the newly adjusted local time-stamp computed at synchronization slot $i\cdot R$. At this point, the party is further capable of excluding invalid beacons.

8.) At the beginning of the next round the party will report a local time equal to $i\cdot R+\mathsf{shift}+1$. If $\mathsf{shift}\geq 0$, the party proceeds by emulating its actions for shift rounds. If $\mathsf{shift}<0$, the party remains a silent observer (recording arrival times for example) until its local time has advanced to slot $i\cdot R+1$ and resumes normally at that round. Note that in this time, an alert party will not revert any previously reported ledger state with overwhelming probability. The reason is that the party will stick to $\mathcal{C}_{\mathsf{loc}}$ during this waiting time and only replace it by longer chains that do not fork by more than k blocks from $\mathcal{C}_{\mathsf{loc}}$ which is a direct consequence of the security guarantees implied by the Genesis chain-selection rule. (An alert party reverting a previously reported state implies a common-prefix violation.)

5.2 The Joining Procedure

De-Registration and Re-Joining. If a party is alert, it can lose in several ways its status of being alert. If a party loses access to the random oracle only, then it will still be able to observe the protocol execution and record message arrivals as seen in Section C.1. The main issue is that such a party—when re-joining—will have to retrace what it missed. This is slightly complicated due to the adjustments to the local clock in the course of the execution. However, the party has all reliable information to actually retrace the actions as if it was present as a passive observer all the time. This special procedure SimulateClockAdjustments is described in Section C.13. It is invoked as part of procedure LedgerMaintenance before preforming as an alert party again.

On the other hand, if any alert party loses access to $\mathcal{G}_{\text{ImperfLClock}}$ or $\mathcal{F}_{\text{N-MC}}$ by the respective de-registration queries, or if it joins anew only late in the execution, then it considers itself as de-synchronized. Parties are aware of their synchronization status, and any party that is de-synchronized will have to run through the main joining procedure JoinProc.

Description of JoinProc. Introducing synchronization slots into the protocol serves the main purpose of enabling a novel joining procedure that newly joining (or resynchronizing) parties can execute to bootstrap an actual reliable time-stamp and ledger state, where a reliable time-stamp is one that lies in the interval of time stamps reported by alert parties. The joining procedure is divided into several phases where the party gathers reliable information, identifies a good synchronization interval and finally applies the shift(s) that will allow it to report a local time-stamp that is sufficiently close to the alert parties in the system. Below

[16] Evaluated using this epoch's stake distribution.

we give an overview and rationale behind our procedure and formally prove its security in Section 6.5. The code for this procedure is given in Figure 2 containing the procedure JoinProc which is invoked as part of LedgerMaintenance for newly joining parties. The procedures refer to parameters that are summarized in Table 1 along with their default values.

Phase A: A joining party with all resources available invokes the main round procedure triggering the join procedure that first resets the local variables.

Phase B: In the second activation upon a MAINTAIN-LEDGER command, the party will jump to phase B and continue to do so until and including round t_{off}. During this interval, the party applies the Genesis chain selection rule maxvalid-bg to filter its incoming chains. It will apply the chain selection rule to all valid chains it receives. Since the party does not have reliable time, it will consider also future chains as valid, as long as they satisfy all remaining validity predicates (cf. Section C.5). As we prove in Lemma 6, at the end of this phase, the party adopts chain $\mathcal{C}$ that stands in a particularly useful relation to any chain $\mathcal{C}'$ an alert party adopts. Roughly, the relation says that the point at which the two chains fork is about k blocks behind the tip of $\mathcal{C}'$. This follows from the Genesis chain selection rule and the fact that $\mathcal{C}'$ is more dense than $\mathcal{C}$ shortly after the fork. However, this also means that P could still hold an extremely long chain served by the adversary (namely, an adversarial extension of an alert party's chain at some point less than k blocks behind the tip into the future). On the positive side, the stake distribution used for general validation of blocks and beacons logically associated to the time before the fork are reliable.

Phase C: If a party arrives at local time $t_{\text{off}} + 1$, it starts with phase C, the gathering phase. The party still filters chains as before, but now processes the arrival times of beacons from the network (or indirectly via the received chains). This phase is parameterized by two quantities: the sum of t_{minSync} and t_{stable} define the total duration of this round, where intuitively, t_{minSync} guarantees that enough arrival times are recorded to compute a reliable estimate of the time-shift, and t_{stable} ensures that the blockchain reaches agreement on which (valid) synchronization beacons to use. After this phase, a party can reliably judge valid arrival times.

Phase D: The party collects the valid evidence and computes the adjustment based on the first synchronization interval $I = [(i-1)R, \ldots, (i-1)R + R/6]$ identified on the blockchain that reports beacons that arrived sufficiently later than the start of phase C (parameter t_{pre}). Party P computes the adjustment value that alert parties would do at synchronization slot $i \cdot R$ based on the recorded beacon arrival times associated with interval I. The party P is done if its adjusted time does not indicate that it should have passed another synchronization slot (and otherwise, the above is repeated with adjusted arrival times of already recorded beacons).

6 Analysis Details

6.1 Security Assumptions: Alert and Participating Stake Ratio

We begin by setting down notation and defining the conventions we adopt for measuring stake ratios. The following definition is adapted from [4]; the crucial difference is that it refers to the types of parties with respect to a *logical slot* as defined in Section 4.

Definition 1 (Classes of parties and their relative stake). *Let $\mathcal{P}[\mathtt{sl}]$ denote the set of all parties in a logical slot* sl *and let $\mathcal{P}_{type}[\mathtt{sl}]$, for any* type *of party described in Figure 1 (e.g. alert, active), denote the set of all parties of the respective type in the slot* sl*. For a set of parties $\mathcal{P}_{type}[\mathtt{sl}]$, let $\mathcal{S}^{-}(\mathcal{P}_{type}[\mathtt{sl}]) \in [0,1]$ (resp. $\mathcal{S}^{+}(\mathcal{P}_{type}[\mathtt{sl}]) \in [0,1]$) denote the minimum (resp., maximum), taken over the views of all alert parties, of the total relative stake of all the parties in $\mathcal{P}_{type}[\mathtt{sl}]$ in the stake distribution used for sampling the slot leaders for slot* sl*.*

Looking ahead, we remark that even though we give the general definition above, our protocol will have the desirable property that for all party types and all time slots, $\mathcal{S}^{-}(\mathcal{P}_{\mathsf{type}}[\mathtt{sl}]) = \mathcal{S}^{+}(\mathcal{P}_{\mathsf{type}}[\mathtt{sl}])$ with overwhelming probability, as all the alert parties will agree on the distribution used for sampling slot leaders with overwhelming probability.

Definition 2 (Alert ratio, participating ratio). *For any logical slot* $\mathtt{sl}$ *during the execution, we let: (i.) the* alert stake ratio *be the fraction* $\mathcal{S}^{-}(\mathcal{P}_{alert}[\mathtt{sl}])/\mathcal{S}^{+}(\mathcal{P}_{active}[\mathtt{sl}])$*; and (ii.) the (potentially)* participating stake ratio *be* $\mathcal{S}^{-}(\mathcal{P}_{active}[\mathtt{sl}])$*.*

It is instructive to see that the potentially participating stake ratio allows us to infer the ratio of stake belonging to parties that cannot participate in slot $\mathtt{sl}$. Intuitively speaking, we will prove the security of our protocol under the assumption that both stake ratios from Definition 2 are sufficiently lower-bounded (the former one by $1/2 + \varepsilon$, the latter one by a constant). We remark that it is easy to verify that in particular, such an assumption also implies the existence of alert parties at any point in the execution.

6.2 Blockchain Security Properties

We now define the standard security properties of blockchain protocols: *common prefix*, *chain growth* and *chain quality*. These will later be useful as an intermediate step in establishing the UC-security guarantees.

Similarly to [4], we only grant these guarantees to *alert* parties. More importantly for this work, the definitions from [4] need to be adjusted to take into account the fact that the local clocks of the parties are not synchronized. To this end, we choose now to define the properties below with respect to the *logical* timestamps (i.e., slot numbers) contained in blocks, and the local clocks of the parties. Namely, we refer to logical slots below, and a party is considered to *be on the onset* of slot $\mathtt{sl}$ (or *enter* slot $\mathtt{sl}$) if her local clock just switched to $\mathtt{sl}$.

Common Prefix (CP); with parameters $k \in \mathbb{N}$**.** The chains $\mathcal{C}_1, \mathcal{C}_2$ possessed by two alert parties at the onset of the slots $\mathtt{sl}_1 < \mathtt{sl}_2$ are such that $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$, where $\mathcal{C}_1^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}_1$, and $\preceq$ denotes the prefix relation.

Chain Growth (CG); with parameters $\tau \in (0, 1], s \in \mathbb{N}$**.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a slot $\mathtt{sl}$. Let $\mathtt{sl}_1$ and $\mathtt{sl}_2$ be two previous slots for which $\mathtt{sl}_1 + s \leq \mathtt{sl}_2 \leq \mathtt{sl}$, so $\mathtt{sl}_2$ is at least s slots ahead of $\mathtt{sl}_1$. Then $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \geq \tau \cdot s$. We call τ the *speed coefficient*.

Chain Quality (CQ); with parameters $\mu \in (0, 1]$ **and** $k \in \mathbb{N}$**.** Consider any portion of length at least k of the chain possessed by an alert party at the onset of a slot; the ratio of blocks originating from alert parties is at least μ. We call μ the chain quality coefficient.

Existential Chain Quality ($\exists\mathsf{CQ}$); with parameter $s \in \mathbb{N}$**.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a slot $\mathtt{sl}$. Let $\mathtt{sl}_1$ and $\mathtt{sl}_2$ be two previous slots for which $\mathtt{sl}_1 + s \leq \mathtt{sl}_2 \leq \mathtt{sl}$. Then $\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]$ contains at least one alertly generated block (i.e., block generated by an alert party).

The first 3 properties are standard, the last one is a slight variant of chain quality fitting better our analysis. For brevity we sometimes write $\mathsf{CP}(k)$ (resp., $\mathsf{CG}(\tau, s)$, $\mathsf{CQ}(\mu, k)$, $\exists\mathsf{CQ}(s)$) to refer to these properties.

While these definitions based on the logical time allow us to talk about the logical structure of the forks created by the parties and reuse parts of the technical machinery given in [26,14,4] to analyze it, providing only guarantees based on the logical time would be unsatisfactory, as the parties running Ouroboros Chronos desire persistence and liveness with respect to a more "real-time" notion (that we define in a moment). We will address this translation from logical-time to real-time guarantees later in Section 6.7.

6.3 Formal Definitions of Nominal Time and Skew

As discussed earlier, for many of the security arguments, it is very convenient to define a *nominal time* notion:

Definition 3 (Nominal Time). *Given an execution of Ouroboros Chronos, any prefix of the execution can be mapped deterministically to an integer* t*, which we call nominal time, as follows: parsing the prefix from genesis and keeping track of the honest party set registered with the imperfect clock functionality (bootstrapped with the set of inaugural alert parties),* t *is the number of times the functionality internally switches all flags* $d_{\mathsf{P}}, \mathsf{P} \in \mathcal{P}$ *from* 1 *to* 0 *until the final step of the execution prefix. (In case no honest party exists in the execution* t *is undefined).*

Nominal time is a technical definition useful for the analysis. It naturally coincides with the idea of defining a baseline that runs at a certain speed, but where parties have some varying (but bounded) lead ahead of the baseline. For example, if a set of alert parties execute Chronos from the beginning, then nominal time lower bounds the *number* of rounds completed by any of them. Furthermore, by the bounded (absolute) drift enforced by $\mathcal{G}_{\text{IMPERFLCLOCK}}^{\Delta_{\text{clock}}}$, the number of locally completed rounds by these alert parties can each be decomposed to be $t + \delta$ (nominal) rounds, where t is the baseline, and δ is bounded by Δ_{clock}.

We next state a definition that will help us quantify how much parties' (local) timestamps deviate from the nominal time and from each other.

Definition 4 (Clock skew and Skew_Δ). *Given an honest party* P*, we define its* skew *in slot* $\mathtt{sl}$ *(denoted* $\mathit{Skew}^{\mathsf{P}}[\mathtt{sl}]$*) as the difference between* $\mathtt{sl}$ *and the nominal time* t *when* P *enters slot* $\mathtt{sl}$*. For any* $\Delta \geq 0$ *and a slot* $\mathtt{sl}$*, we denote by* $\mathit{Skew}_\Delta[\mathtt{sl}]$ *the predicate that for all parties that are synchronized in slot* $\mathtt{sl}$*, their skew in this slot differs by at most* Δ*; formally*

$$\mathit{Skew}_\Delta[\mathtt{sl}] :\Leftrightarrow \left(\forall \mathsf{P}_1, \mathsf{P}_2 \in \mathcal{P}_{alert}[\mathtt{sl}]\colon \left|\mathit{Skew}^{\mathsf{P}_1}[\mathtt{sl}] - \mathit{Skew}^{\mathsf{P}_2}[\mathtt{sl}]\right| \leq \Delta\right) .$$

Note that in the static-registration setting, all honest parties are synchronized (and hence are considered for $\mathsf{Skew}_\Delta[\mathtt{sl}]$); the difference will become important in later sections.

6.4 Setting with Static Registration

As outlined in Section 3, our first goal is to establish that the properties of (logical-time) common prefix, chain growth, and chain quality are achieved by Ouroboros Chronos when executed in a restricted environment where all parties participate in the protocol run from the beginning and never get deregistered from any of their resources (i.e., from $\mathcal{G}_{\text{RO}}$, $\mathcal{F}_{\text{N-MC}}$ or $\mathcal{G}_{\text{IMPERFLCLOCK}}$). Similarly to [4], we refer to this setting as the *setting with static registration*; we will drop this assumption later.

6.4.1 Single-Epoch Analysis with Δ-Bounded Skew

Before we can analyze the resynchronization procedure $\mathsf{SyncProc}$, we first need to establish the blockchain security properties given in Section 6.2 for Ouroboros Chronos during a single-epoch execution, as the proper functioning of $\mathsf{SyncProc}$ will inductively depend on these properties being satisfied in the epochs preceding it. Having this inductive structure of the proof in mind, we actually need a security statement for the single-epoch setting in the regime where the predicate $\mathsf{Skew}_\Delta[\mathtt{sl}]$ is satisfied for all slots in that epoch, we refer to this as the setting *with Δ-bounded skew*. Looking ahead, this property will be guaranteed in the first epoch thanks to $\mathcal{F}_{\mathsf{INIT}}$, and preserved by induction.

The desired properties are established in the following theorem; we give its proof in Appendix F. Recall that R denotes the epoch length in slots, f is the active-slot coefficient, let $\Delta \geq \Delta_{\text{net}} + \Delta_{\text{clock}}$ be the upper bound on the sum of the network delay and clock drift and let $\widetilde{\Delta} \triangleq 2\Delta$.

Theorem 1. *Consider the single-epoch execution of the protocol* Ouroboros-Chronos *in the setting with static registration and Δ-bounded skew. Let $\alpha, \beta \in [0,1]$ denote a lower bound on the alert ratio and participating ratio throughout this epoch, respectively. If for some $\epsilon \in (0,1)$ we have*

$$\alpha \cdot (1-f)^{\widetilde{\Delta}+1} \geq (1+\epsilon)/2\,, \tag{5}$$

and the maxvalid-bg *parameters, k and s, satisfy*

$$k > 192\widetilde{\Delta}/(\epsilon\beta) \qquad \text{and} \qquad R/6 \geq s = k/(4f) \geq 48\widetilde{\Delta}/(\epsilon\beta f) \tag{6}$$

then Ouroboros-Chronos *achieves the following guarantees:*

Common prefix. *The probability that it violates the common prefix property with parameter k' is no more than $\bar{\epsilon}_{\mathsf{CP}}(k'; R, \Delta, \epsilon) + \bar{\epsilon}_{\mathsf{mv}}$ where*

$$\bar{\epsilon}_{\mathsf{CP}}(k'; R, \Delta, \epsilon) \triangleq \frac{19R}{\epsilon^4} \exp(\widetilde{\Delta} - \epsilon^4 k'/18)\,;$$

Chain growth. *The probability that it violates the chain growth property with parameters* $s' \geq 48\widetilde{\Delta}/(\epsilon\beta f)$ *and* $\tau_{\mathsf{CG}} = \beta f/16$ *is no more than* $\bar{\epsilon}_{\mathsf{CG}}(\tau_{\mathsf{CG}}, s'; R, \epsilon) + \bar{\epsilon}_{\mathsf{mv}}$ *where*

$$\bar{\epsilon}_{\mathsf{CG}}(\tau_{\mathsf{CG}}, s'; R, \epsilon) \triangleq \frac{s'R^2}{2} \exp\left(-(\epsilon\beta f)^2 s'/256\right) \ ;$$

Existential chain quality. *The probability that it violates the existential chain quality property with parameter* $s' \geq 12\widetilde{\Delta}/(\epsilon\beta f)$ *is no more than* $\bar{\epsilon}_{\exists\mathsf{CQ}}(s'; R, \epsilon) + \bar{\epsilon}_{\mathsf{mv}}$ *where*

$$\bar{\epsilon}_{\exists\mathsf{CQ}}(s'; R, \epsilon) \triangleq (s'+1)R^2 \exp\left(-(\epsilon\beta f)^2 s'/64\right) \ ;$$

Chain quality. *The probability that it violates the chain quality property with parameters* $k' \geq 48\widetilde{\Delta}/(\epsilon\beta f)$ *and* $\mu = \epsilon\beta f/16$ *is no more than* $\bar{\epsilon}_{\mathsf{CQ}}(\mu, k'; R, \epsilon) + \bar{\epsilon}_{\mathsf{mv}}$ *where*

$$\bar{\epsilon}_{\mathsf{CQ}}(\mu, k'; R, \epsilon) \triangleq \frac{kR^2}{2} \exp\left(-(\epsilon\beta f)^2 k'/256\right) \ ;$$

and where $\bar{\epsilon}_{\mathsf{mv}}$ *is a shorthand for the quantity*

$$\bar{\epsilon}_{\mathsf{mv}} \triangleq \exp\left(\ln R - \Omega(k)\right) + \bar{\epsilon}_{\mathsf{CG}}(\beta f/16, k/(4f)) + \bar{\epsilon}_{\exists\mathsf{CQ}}(k/(4f)) + \bar{\epsilon}_{\mathsf{CP}}(k\beta/64) \ .$$

In this work, we analyze the concrete security of the protocol. In asymptotic terms, if κ denotes the security parameter, we can treat the chain selection parameters k and s, as well as the parameters k' and s' of CP, CG, and CQ, as functions in $\omega(\log k)$ to conclude that the error terms are negligible such that all chain properties hold (for the concrete coefficients τ_{CG} and μ) except with negligible probability. Note that the bound on s implies that the epoch length R has a lower bound in $\omega(\log k)$, too.

6.4.2 Properties of SyncProc

Here we establish two key properties of the resynchronization procedure SyncProc given in Section 5.1 that is being executed by all alert parties on the edge of any two epochs. Consider the following fact, proven in Appendix G for completeness.

Fact 1 *Let* $(a_i)_{i=1}^n$ *and* $(b_i)_{i=1}^n$ *be two sequences of* n *integers each, with the property that* $|a_i - b_i| \leq \Delta$ *for all* $i \in [n]$*. Then we also have* $|\mathsf{med}\left((a_i)_{i=1}^n\right) - \mathsf{med}\left((b_i)_{i=1}^n\right)| \leq \Delta$.

The above simple statement is at the heart of our analysis of the synchronization procedure SyncProc. Informally, consider an execution of Ouroboros-Chronos over its full lifetime consisting of several epochs, and focus on the edge between epochs i and $i+1$, where SyncProc is executed. We show that it satisfies two properties:

SyncProc **maintains** Skew_Δ. If (some parametrizations of) CG and CP are not violated up to the end of epoch i, then Skew_Δ is satisfied in the first slot of epoch $i+1$.

Bounded shift**.** If a lower bound on α, some parametrization of $\exists$CQ, and Skew_Δ are not violated up to epoch i, then the shift by which an alert party updates its local clock in SyncProc right before epoch $i+1$ satisfies $|\mathsf{shift}| \leq 2\Delta$.

We state each of these properties in a formal manner separately as Lemmas 4 and 5 and give their full proofs in Appendix G.

Here we only briefly comment on the proof of the first property, which relies on two intermediate claims: The first is that all alert parties use the same set of synchronization beacons in their execution of the procedure SyncProc between epochs ep and ep + 1; the second is that for any fixed beacon $\mathtt{SB} \in \mathcal{S}_j^{\mathsf{P}_1} = \mathcal{S}_j^{\mathsf{P}_2}$ (in the jth synchronization slot), the quantity $\mu(\mathsf{P}_i, \mathtt{SB}) \triangleq \mathsf{Skew}^{\mathsf{P}_i}[\mathtt{sl}] + \mathsf{slotnum}(\mathtt{SB}) - \mathsf{P}_i.\mathsf{Timestamp}(\mathtt{SB})$ will differ by at most Δ between any two alert parties P_1 and P_2.

6.4.3 Lifting to Multiple Epochs

Theorem 1 gives us security guarantees achieved by Ouroboros Chronos in a single-epoch setting with static stake distribution and perfect randomness. These guarantees can be extended throughout the whole lifetime of the system consisting of many epochs by an inductive argument over epochs, using the properties of $\mathcal{F}_{\mathsf{INIT}}$ and Theorem 1 for the base case, and the epoch-randomness analysis of [14] together with the properties of SyncProc from Section 6.4.2 (again together with Theorem 1) for the inductive step. We give an informal theorem here and defer the formal statement (Theorem 5) and its proof to Appendix H.

Theorem 2 (Full-execution security with static registration; informal). *Consider the setting with static registration. If the assumptions (5) and (6) are satisfied, then the full execution of Ouroboros-Chronos achieves the same guarantees for* CP, CG, CQ, ∃CQ *as given in Theorem 1 except with additional error probability roughly equal to the one induced by the lifting argument in [4].*

While we always target concrete security, an asymptotic summary of this theorem would be that under the same dependencies of the parameters on a security parameter as explained after Theorem 1, the additional error term incurred by the lifting is just negligible.

6.5 Newly Joining Parties

In this section we prove that the guarantees on common prefix, chain growth and (existential) chain quality obtained for Ouroboros-Chronos in Section 6.4 remain valid also when new parties join the protocol later during its execution. Again in this section Δ denotes the upper bound on the sum of the maximum network delay and maximum clock drift. We use the letter t to refer to (nominal) time, i.e., to the sequence of execution steps where nominal time is t.

Definition 5 (Joining party). *We say that an honest party* P *is* joining the protocol execution *at time* $t_{\mathsf{join}} > 0$ *if* t_{join} *is the nominal time at the point of the execution where* P *becomes operational, time-aware and online for the first time.*

Consider an execution of Ouroboros-Chronos over its full lifetime and a joining party $\mathsf{P}_{\mathsf{join}}$ with access to all its resources. Informally speaking, our analysis shows two properties of the joining process of $\mathsf{P}_{\mathsf{join}}$ that hold as long as some parametrizations of CP, CG, ∃CQ as well as the assumptions of Theorem 5 remain satisfied throughout the joining process:

1. After Phase B, $\mathsf{P}_{\mathsf{join}}$ will be holding a chain $\mathcal{C}_{\mathsf{join}}$ that satisfies $\mathcal{C}_{\mathsf{alert}}^{\lceil k} \preceq \mathcal{C}_{\mathsf{join}}$ with respect to any $\mathcal{C}_{\mathsf{alert}}$ held by an alert party at least Δ time steps ago.
2. In Phase D, $\mathsf{P}_{\mathsf{join}}$ correctly identifies an epoch i^* for which it has collected all the beacons that alert parties had used in their execution of SyncProc after epoch i^*, and based on these beacons mimics the synchronization procedure so that starting with epoch $i^* + 1$, $\mathsf{P}_{\mathsf{join}}$ does not violate Skew_{Δ} as it becomes alert.

Again, we state each of these properties formally as Lemmas 6 and 7 and give their proofs in Appendix I.

6.6 The Dynamic-Availability Setting

Using the above analysis of the joining procedure, we can generalize the results from previous sections to the dynamic availability setting of Section 4, where the parties get arbitrarily registered and deregistered from their resources upon the decision of the environment. We give an informal theorem here and defer the formal statement (Theorem 6) and its proof to Appendix J.

Theorem 3 (Dynamic availability, informal). *In the dynamic-availability setting, under the assumptions of Theorem 2 and Lemma 7, Ouroboros-Chronos achieves the same guarantees for* CP, CG, CQ *and* ∃CQ *as given in Theorem 2 except for a negligible additional error probability that corresponds to violating the assumptions of Lemmas 6 and 7.*

Recall that Lemmas 6 and 7 referred in the above theorem are the formalizations of the argument of Section 6.4.2 and hence their assumptions are informally summarized in that section.

6.7 From Logical-Time to Real-Time Guarantees

Recall that eventually, we are interested in a ledger that provides consistency and liveness and they typically follow black-box from the three blockchain properties above. However, since in our protocol, parties emulate a global time themselves, we must make related logical time advancement with the nominal time, which is especially important for liveness. Since parties adjust their timestamps at the boundary of every epoch, an external observer that takes nominal time as the baseline, would conclude that parties are slightly off. To quantify the general relationship, we introduce a concrete discount factor τ_{TG}. The concrete expression is given by Lemma 8 in the technical Section K, and we state the lemma informally here:

Lemma 1 (Nominal vs. logical time, informal). *Consider an execution of the full protocol Ouroboros-Chronos in the dynamic-availability setting, let* P *be a party that is synchronized between (and including) slots* $\mathtt{sl}$ *and* $\mathtt{sl}'$*, let* t *and* t' *be the nominal times when* P *enters slot* $\mathtt{sl}$ *and* $\mathtt{sl}'$ *for the first time, respectively. Denote by* $\delta\mathtt{sl}$ *and* δt *the respective differences* $|\mathtt{sl}' - \mathtt{sl}|$ *and* $|t' - t|$*. Then, under the same assumptions as before, we have* $\delta\mathtt{sl} \geq \tau_{\mathrm{TG}} \cdot \delta t$ *for large enough* δt*.*

It is important to point out that the τ_{TG} is close to 1 for typical parameter choices and that the lower bound on δt does depend on Δ and not on the security parameter. We are ready to state chain-growth with respect to nominal time. Again, the concrete bounds are given in the formal version, Corollary 7 in technical Section K.

Corollary 1 (Nominal time CG, informal). *Consider the event that the execution of Ouroboros Chronos under the assumptions as above does not violate property* CG *with parameters* $\tau \in (0, 1], s \in \mathbb{N}$*. Let* $\tau_{\mathsf{CG},\mathrm{glob}} \triangleq \tau \cdot \tau_{\mathrm{TG}}$*. Consider a chain* $\mathcal{C}$ *possessed by an alert party at a point in the execution where the party is at an onset of a (local) round and where the nominal time is* t*. Let further* t_1, t_2*, and* δt *be such that* $t_1 + \delta t \leq t_2 \leq t$*. Let* $\mathtt{sl}_1$ *and* $\mathtt{sl}_2$ *be the last slot numbers that* P *reported in the execution when nominal time was* t_1 *(resp.* t_2*) Then it must hold that* $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \geq \tau_{\mathsf{CG},\mathrm{glob}} \cdot \delta t$ *whenever* δt *is sufficiently large,*

6.8 Composable Guarantees of Ledger and Clock

The final step of our treatment is to show that we get composable security. To this end, we can leverage all the above proven statements and compile all obtained properties into concrete instantiations of ledger parameters. In Section 3 we provided the short form of the full UC statement. The full version is given in Section L where we give the simulation argument and some additional proofs to obtain the stated clock parameters.

References

1. Handan Kılınç Alper. Ouroboros clepsydra: Ouroboros praos in the universally composable relative time model. Cryptology ePrint Archive, Report 2019/942, 2019. `https://eprint.iacr.org/2019/942`.
2. Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Optimal clock synchronization under different delay assumptions (preliminary version). In Jim Anderson and Sam Toueg, editors, *12th ACM PODC*, pages 109–120. ACM, August 1993.
3. Michael Backes, Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. On fairness in simulatability-based cryptographic systems. In Vijay Atluri, Pierangela Samarati, Ralf Küsters, and John C. Mitchell, editors, *Proceedings of the 2005 ACM workshop on Formal methods in security engineering, FMSE 2005, Fairfax, VA, USA, November 11, 2005*, pages 13–22. ACM, 2005.
4. 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 18*, pages 913–930. ACM Press, October 2018.
5. Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.
6. Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `http://eprint.iacr.org/2016/919`.

7. Gabriel Bracha. An asynchronou [(n-1)/3]-resilient consensus protocol. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *3rd ACM PODC*, pages 154–162. ACM, August 1984.
8. Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
10. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.
11. Ran Canetti, Kyle Hogan, Aanchal Malhotra, and Mayank Varia. A universally composable treatment of network time. Cryptology ePrint Archive, Report 2017/1256, 2017. `https://eprint.iacr.org/2017/1256`.
12. Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.
13. Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2019.
14. Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.
15. Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *16th ACM STOC*, pages 504–511. ACM Press, 1984.
16. Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization (extended abstract). In Jim Anderson and Sam Toueg, editors, *12th ACM PODC*, pages 97–108. ACM, August 1993.
17. Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults (abstract). In James H. Anderson, editor, *14th ACM PODC*, page 256. ACM, August 1995.
18. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.
19. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. Cryptology ePrint Archive, Report 2016/1048, 2016. `http://eprint.iacr.org/2016/1048`.
20. 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 PODC*, pages 89–102. ACM, August 1984.
21. Dennis Hofheinz and Joern Mueller-Quade. A synchronous model for multi-party computation and the incompleteness of oblivious transfer. Cryptology ePrint Archive, Report 2004/016, 2004. `http://eprint.iacr.org/2004/016`.
22. Yael Kalai, Yehuda Lindell, and Manoj Prabhakaran. Concurrent composition of secure protocols in the timing model. Cryptology ePrint Archive, Report 2005/036, 2005. `http://eprint.iacr.org/2005/036`.
23. Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. Concurrent composition of secure protocols in the timing model. *Journal of Cryptology*, 20(4):431–492, October 2007.
24. Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.
25. T. Kerber, A. Kiayias, M. Kohlweiss, and V. Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, pages 984–1001, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.
26. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017.
27. Leslie Lamport and P. M. Melliar-Smith. Byzantine clock synchronization. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *3rd ACM PODC*, pages 68–74. ACM, August 1984.
28. Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Clock synchronization with bounded global and local skew. In *49th FOCS*, pages 509–518. IEEE Computer Society Press, October 2008.
29. Aanchal Malhotra, Matthew Van Gundy, Mayank Varia, Haydn Kennedy, Jonathan Gardner, and Sharon Goldberg. The security of ntp's datagram protocol. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, volume 10322 of *Lecture Notes in Computer Science*, pages 405–423. Springer, 2017.
30. David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition*. CRC Press, 2010.

31. Jesper Buus Nielsen. *On Protocol Security in the Cryptographic Model*. PhD thesis, University of Aarhus, June 2003.
32. Rafail Ostrovsky and Boaz Patt-Shamir. Optimal and efficient clock synchronization under drifting clocks. In Brian A. Coan and Jennifer L. Welch, editors, *18th ACM PODC*, pages 3–12. ACM, May 1999.
33. Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 643–673. Springer, Heidelberg, April / May 2017.
34. Rafael Pass and Elaine Shi. Rethinking large-scale consensus. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 115–129. IEEE Computer Society, 2017.
35. Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 380–409. Springer, Heidelberg, December 2017.
36. Barbara B. Simons, Jennifer Lundelius Welch, and Nancy A. Lynch. An overview of clock synchronization. In Barbara B. Simons and Alfred Z. Spector, editors, *Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*. Springer, Heidelberg, 1990.
37. T. K. Srikanth and Sam Toueg. Optimal clock synchronization. In Michael A. Malcolm and H. Raymond Strong, editors, *4th ACM PODC*, pages 71–86. ACM, August 1985.

Appendix and Technical Sections

A Short Survey of Related Literature

PoS protocols have been proposed as a sustainable and potentially more scalable alternative to the resource-intensive proof-of-work paradigm, but had initially been criticised as a much more limited technology, due to early attacks such as the nothing-at-stake or the stake-bleeding attack. The systematic study of such blockchains, however, has established them as a viable replacement to PoW blockchains, by proving the ability to counter adaptive attacks [14,26], removing the need of checkpointing while assuming dynamic availability [4], adding transaction privacy [25], and, new in our work, removing the dependency on global time for participation-unrestricted environments.

On a different note, as a tool for bootstrapping loose synchronization—local clocks advancing with approximately the same speed—to (approximate) global time, Chronos shares similarities in its goal with *synchronizers*, which occupy an important part of the distributed computing literature. A synchronizer is a sophisticated distributed fault-tolerant protocol which achieves a similar bootstrapping goal, but relies on either knowledge of concrete participation bounds, or at times even knowledge of credentials of the participants. In a nutshell, in typical synchronizer scenarios [15,27,20] it is assumed that the parties have initially (loosely) synchronized clocks and that the clocks advance at about the same speed. It is proved that without setup assumptions, such as a public-key infrastructure that enables digital signatures, n parties can synchronize their clocks and keep them (loosely) synchronized if and only if no more than $t < n/3$ of the parties report far-drifting or inconsistent clocks values [15,27,20]. This bound can be improved to $t < n/2$ by use of existentially unforgeable digital signatures [27]. A number of follow up works have investigated clock synchronization in various settings [17,16,2,37,28,32] and we refer the reader to [36] for a survey (albeit somewhat outdated).

Unfortunately the techniques used in the above traditional synchronizer setting do not translate to the unrestricted participation setting we consider here. Informally, the reason is that these techniques rely on knowledge (and counting) the total number of parties in the system, an assumption which one cannot make in our setting. More concretely, the main tool used by classical synchronizers is counting of messages in combination with signatures to thwart malicious behavior. In the dynamic availability setting, counting messages does not work (or even counting "stake" in the PoS setting), as the parties have no way of knowing how many, or which parties are present at any given time. We note in passing that this is also a major factor that distinguishes the Algorand [12] approach to decentralized consensus from that we follow here. Concretely, Algorand explicitly assumes that parties know (approximately) how many parties are in any committee, and therefore, know how many honest messages they can expect by any such committee. This assumption has been shown to allow Algorand to achieve complete agreement on the whole blockchain after each block; and,

although not explicitly proven we conjecture that techniques from the permissioned synchronizers literature can be employed here to achieve global clock synchronization in the setting of [12]. It is evident, however, that this is poses a much stronger assumption than that of dynamic availability leaving open the question of whether PoS-based clock synchronization in the dynamic availability setting is even possible, which we answer affirmatively in our work.

The protocol most closely related to Chronos, which is also the starting point of the underlying blockchain, is Ouroboros Genesis [4]. Notwithstanding, the two solutions have major differences both in terms of their assumptions (access to global time vs. merely similar speed clocks), and in terms of techniques used in the protocol design and in the proof. We refer to Section C for a discussion of these differences. We note that a recent and concurrent work [1] which appeared after our paper (an earlier version of our paper[17] with only minor differences was made public prior to the publication of [1]) also investigated the possibility of using the Ouroboros family of protocols for synchronization of newly joining parties. However [1] lacks a full treatment of dynamic availability and appears to follow a protocol design that we have already argued in the introduction to be insufficient for providing secure synchronization.

B Model Details

The purpose of this section is to give the details of the underlying (UC) model.

Relaxed Synchrony. We use the following version of imperfect local clocks that allow parties to proceed at roughly the same speed, where "roughly" is captured by the upper bound Δ_{clock} on the drift between any two honest parties:

Functionality $\mathcal{G}_{\text{IMPERFLCLOCK}}^{\Delta_{\text{clock}}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $\mathsf{P} = (\text{pid}, \text{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} := \emptyset$ and $F := \emptyset$.

For each identity $\mathsf{P} := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages bit variables d_{P} and d_P^{Imp}, and a (possibly negative) integer $\texttt{drift}_{\mathsf{P}}$. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})} \in \{0, 1\}$ (all these variables are initially set to 0).

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $\mathsf{P} \in \mathcal{P}$ set $d_{\mathsf{P}}^{Imp} := 1$ and $d_{\mathsf{P}} := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-PUSH}, \text{sid}_C, \mathsf{P})$ from $\mathcal{A}$ where party $\mathsf{P} \in \mathcal{P}$, if $d_{\mathsf{P}}^{Imp} := 1$ and $\texttt{drift}_{\mathsf{P}} < \Delta_{\text{clock}}$ then update $d_{\mathsf{P}}^{Imp} := 0$ and $\texttt{drift}_{\mathsf{P}} := \texttt{drift}_{\mathsf{P}} + 1$ and return $(\text{CLOCK-PUSH-OK}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$. Otherwise ignore the message.
- Upon receiving $(\text{CLOCK-GET}, \text{sid}_C)$ 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 *Round-Update*; return $(\text{CLOCK-GET}, \text{sid}_C, d_{\mathsf{P}}^{Imp})$ (resp. $(\text{CLOCK-GET}, \text{sid}_C, d_{(\mathcal{F},\text{sid})})$) to the requestor (where sid is the sid of the calling instance).

Procedure Round-Update: For each session sid do: If $d_{(\mathcal{F},\text{sid})} := 1$ for all $\mathcal{F} \in F$ and $d_{\mathsf{P}} = 1$ for all honest parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$, then update $d_{(\mathcal{F},\text{sid})} := 0$ and update $d_{\mathsf{P}} := 0$ and $\texttt{drift}_{\mathsf{P}} := \texttt{drift}_{\mathsf{P}} - 1$ for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$; for all $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$ with $\texttt{drift}_{\mathsf{P}} < 0$ reset $d_{\mathsf{P}}^{Imp} := 0$ and $\texttt{drift}_{\mathsf{P}} := 0$.

To improve accessibility, we colored in blue the differences, i.e., the much weaker coordination that the functionality provides compared to the original clock of [24]. In order to appreciate the difference, we sketch the clock from [24] here:

[17] See `https://eprint.iacr.org/2019/838`

Functionality $\mathcal{G}_{\text{PERFLCLOCK}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $\mathsf{P} = (\text{pid}, \text{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} := \emptyset$ and $F := \emptyset$.

For each identity $\mathsf{P} := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable $d_{\mathsf{P}} \in \{0, 1\}$. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})} \in \{0, 1\}$ (both bit variables are initially set to 0).

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $\mathsf{P} \in \mathcal{P}$ set $d_{\mathsf{P}} := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-GET}, \text{sid}_C)$ 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 *Round-Update*, return $(\text{CLOCK-GET}, \text{sid}_C, d_{\mathsf{P}})$ (resp. $(\text{CLOCK-GET}, \text{sid}_C, d_{(\mathcal{F},\text{sid})})$) to the requestor (where sid is the sid of the calling instance).

Procedure Round-Update: For each session sid do: If $d_{(\mathcal{F},\text{sid})} := 1$ for all $\mathcal{F} \in F$ and $d_{\mathsf{P}} = 1$ for all honest parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$, then update $d_{(\mathcal{F},\text{sid})} := 0$ and $d_{\mathsf{P}} := 0$ for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$.

Modeling Peer-to-Peer Communication. We assume a diffusion network in which all messages sent by honest parties are guaranteed to be fetched by protocol participants after a specific delay Δ_{net}. Additionally, the network guarantees that once a message has been fetched by an honest party, this message is fetched by any other honest party within a delay of at most Δ_{net}, even if the sender of the message is corrupted. This is a slightly stronger guarantee than the multicast-functionality from [5,4] which only guaranteed this bounded delivery for messages sent by honest parties, however it appears to capture well the behavior of peer-to-peer gossiping where honest parties always forward messages they have not already seen. To avoid confusion, we refer to using such a network as broadcasting. We detail the corresponding functionality below:

Functionality $\mathcal{F}_{\text{N-MC}}^{\Delta_{\text{net}}}$

The functionality is parameterized with a set possible senders and receivers $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$.

- **Honest sender multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from some $\mathsf{P} \in \mathcal{P}$, where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set, choose n new unique message-IDs $\text{mid}_1, \ldots, \text{mid}_n$ of the form $\text{mid}_i = (\text{mid}, i)$, initialize $2n$ new variables $D_{\text{mid}_1} := D_{\text{mid}_1}^{MAX} \ldots := D_{\text{mid}_n} := D_{\text{mid}_n}^{MAX} := 1$, a per message-delay $\Delta_{\text{mid}_i} = \Delta_{\text{net}}$ for $i = 1, \ldots, n$ and set $\boldsymbol{M} := \boldsymbol{M}||(m, \text{mid}_1, D_{\text{mid}_1}, U_1)|| \ldots ||(m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, and send $(\text{MULTICAST}, \text{sid}, m, \mathsf{P}, (U_1, \text{mid}_1), \ldots, (U_n, \text{mid}_n))$ to the adversary.
- **Adversarial sender multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from some $\mathsf{P} \in \mathcal{P}$ (where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set), do execute it the same way as an honest-sender multicast, with the only difference that $\Delta_{\text{mid}_i} = \infty$.
- **Honest party fetching.** Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P if P is corrupted):
  1. For all tuples $(m, \text{mid}, D_{\text{mid}}, \mathsf{P}) \in \boldsymbol{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\boldsymbol{M}_0^{\mathsf{P}}$ denote the subvector $\boldsymbol{M}$ including all tuples of the form $(m, \text{mid}, D_{\text{mid}}, \mathsf{P})$ with $D_{\text{mid}} = 0$ (in the same order as they appear in $\boldsymbol{M}$). Then, delete all entries in $\boldsymbol{M}_0^{\mathsf{P}}$ from $\boldsymbol{M}$ and in case some $(m, \text{mid}, D_{\text{mid}}, \mathsf{P})$ is in $\boldsymbol{M}_0^{\mathsf{P}}$, where P is honest, set $\Delta_{\text{mid}'} = \Delta_{\text{net}}$ for any $(m, \text{mid}', D_{\text{mid}'}, \mathsf{P}')$ in $\boldsymbol{M}$ and replace this record by $(m, \text{mid}', \min\{D_{\text{mid}'}, \Delta_{\text{net}}\}, \mathsf{P}')$. Finally, send $\boldsymbol{M}_0^{\mathsf{P}}$ to P.
- **Adding adversarial delays.** Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\text{mid}_{i_j}}, \text{mid}_{i_j})$:

If $D^{MAX}_{\mathrm{mid}_{i_j}} + T_{\mathrm{mid}_{i_j}} \leq \Delta_{\mathrm{mid}_{i_j}}$ and mid_{i_j} is a message-ID registered in the current $\boldsymbol{M}$, set $D_{\mathrm{mid}_{i_j}} := D_{\mathrm{mid}_{i_j}} + T_{\mathrm{mid}_{i_j}}$ and set $D^{MAX}_{\mathrm{mid}_{i_j}} := D^{MAX}_{\mathrm{mid}_{i_j}} + T_{\mathrm{mid}_{i_j}}$; otherwise, ignore this pair.

– **Adversarially reordering messages.** Upon receiving $(\textsc{swap}, \mathrm{sid}, \mathrm{mid}, \mathrm{mid}')$ from the adversary, if mid and mid′ are message-IDs registered in the current $\boldsymbol{M}$, then swap the triples $(m, \mathrm{mid}, D_{\mathrm{mid}}, \cdot)$ and $(m, \mathrm{mid}', D_{\mathrm{mid}'}, \cdot)$ in $\boldsymbol{M}$. Return $(\textsc{swap}, \mathrm{sid})$ to the adversary.

The Genesis Block Distribution with Weak Start Agreement. In this work, we not only allow that parties' local time-stamps might shift apart over the course of an execution, we also model that the genesis block is received by initial stakeholders in a delayed fashion (such that one party already starts staking, but another one is still waiting for the genesis block). To this aim, we weaken the functionality $\mathcal{F}_{\mathsf{INIT}}$ described in [4] to allow for bounded offsets in starting times. Namely, $\mathcal{F}^{\Delta_{\mathrm{net}}}_{\mathsf{INIT}}$ does not enforce that all honest stakeholders receive the created genesis block in the same round, but merely guarantees delivery differences of Δ_{net}, just as the message delays above (the offsets are under adversarial control). In fact, we find it reasonable to think of the genesis block delay as being considered similar to a "message delay", i.e., we model that the genesis blocks "arrives with a delay". The details of the $\mathcal{F}^{\Delta_{\mathrm{net}}}_{\mathsf{INIT}}$ functionality appear below:

More concretely, we allow the adversary to define the offsets upon the first activation to the functionality. For the sake of convenience, we consider this initial offset query to the adversary as restricting (and prefix the query with the keyword Respond) as defined by Camenisch et al. [8] which means that the adversary is required to answer this query immediately (and hence the offsets can technically be seen as chosen "when the $\mathcal{F}_{\mathsf{INIT}}$ is created").[18] The functionality makes sure that at the onset of the execution, the genesis block is actually created with the keying material form the initial stakeholders. As long as this creation is not yet complete, the baseline cannot advance (recall that ideal functionalities are allowed to proceed at the baseline speed). Once the creation is complete, which defines the "big bang" for this execution, the clock is released, and at this point, the offsets steer when a initial stakeholder receives genesis block and start working.

Functionality $\mathcal{F}^{\Delta_{\mathrm{net}}}_{\mathsf{INIT}}$

The functionality $\mathcal{F}_{\mathsf{INIT}}$ is parameterized by the delay Δ_{net}, the set $\mathsf{P}_1, \ldots, \mathsf{P}_n$ of initial stakeholders n and their respective stakes $s_1, \ldots, s_n$. It additionally stores n variables offset_i, one for each stakeholder P_i to steer when they receive the genesis block (once ready), a variable ready to steer when the genesis block is fully generated. It maintains the set of registered parties $\mathcal{P}$.

– On the first activation of the functionality, send $(\mathsf{Respond}, (\mathsf{DefineOffset}, sid))$ to the adversary. Upon receiving the response $(\mathsf{DefineOffset}, sid, o_1, \ldots, o_n)$ where $o_1 \in [0, \ldots, \Delta_{\mathrm{net}}]$, set $\mathsf{offset}_i := o_i$. It further sets $\mathsf{ready} \leftarrow 0$ and registers with the clock. Proceed handling the first input as specified in the following.
– On receiving any input by a registered party, do the following case distinction:
  • If $\mathsf{ready} = 0$ and the message is a request from some initial stakeholder $\mathsf{P} = \mathsf{P}_i$, $i \in [n]$, of the form $(\mathsf{ver\_keys}, sid, \mathsf{P}, v^{\mathrm{vrf}}, v^{\mathrm{kes}})$, then $\mathcal{F}_{\mathsf{INIT}}$ stores the verification keys tuple $(\mathsf{P}_i, v_i^{\mathrm{vrf}}, v_i^{\mathrm{kes}})$ and acknowledges its receipt. If some of the registered public keys are equal, it outputs an error and halts. Otherwise, it samples and stores a random value $\eta_1 \xleftarrow{\$} \{0,1\}^\lambda$ and constructs a genesis block $(\mathbb{S}_1, \eta_1)$, where $\mathbb{S}_1 = \big((\mathsf{P}_1, v_1^{\mathrm{vrf}}, v_1^{\mathrm{kes}}, s_1), \ldots, (\mathsf{P}_n, v_n^{\mathrm{vrf}}, v_n^{\mathrm{kes}}, s_n)\big)$. If all initial stakeholders have made their requests, then $\mathcal{F}_{\mathsf{INIT}}$ de-registers from the clock to release the execution.
  • If $\mathsf{ready} = 1$, then do the following
    ∗ If the currently received input is the kth request of the form $(\mathsf{genblock\_req}, sid, \mathsf{P})$ from an initial stakeholder $\mathsf{P} = \mathsf{P}_i$ (for some $i \in [n]$), if $k > \mathsf{offset}_i$, $\mathcal{F}_{\mathsf{INIT}}$ sends $(\mathsf{genblock}, sid, (\mathbb{S}_1, \eta_1))$ to P.

[18] In case the query would be not be restricting, this would incur a slight but rather artificial complication of the initialization procedure of the protocol as we would have to take into account that the very first activated honest protocol participant (and only this one) will actually lose its activation token right away. Defining this query to be restricting is not crucial for our treatment.

∗ If it is a request from a party that is not initial stakeholder then return $((\mathsf{genblock}, sid, (\mathbb{S}_1, \eta_1)), \mathsf{Running})$ to the requestor. // The case when the setup is completed and potentially, the protocol has advanced a lot and the party must bootstrap from this genesis block.

B.1 Additional Functionalities Used in the Proof

The protocol makes use of a VRF (verifiable random function) functionality $\mathcal{F}_{\mathsf{VRF}}$, a KES (key-evolving signature) functionality $\mathcal{F}_{\mathsf{KES}}$, a (global) random oracle functionality $\mathcal{G}_{\mathrm{RO}}$. We use the random oracle as the idealization of a hash function. We use the strongest form of a global random oracle to express that our new consensus algorithm does not need any kind of programmability or query restrictions (and the result using a local random oracle is implied). The idealizations $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$ are shown to be realizable under standard assumptions or an additional random oracle in [14].

Recall that in order to reflect the synchronization and liveness pattern properly in UC, ideal signing or verification (in both, KES or VRF) is a local operation performed by a slot-leader and hence one invokes the proposed formalism of Camenisch et al. [8] and declare signing request as *restricting*, which means that although activated to provide a signature, the adversary has to provide the answer, i.e. the signature string for example, immediately (no other output to another protocol machine is allowed) and return the activation token back to the functionality. Such responsive queries are prefixed with the keyword Respond in outputs to the adversary.

Verifiable Random Functions. The idealized VRF functionality is defined below:

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

$\mathcal{F}_{\mathsf{VRF}}$ interacts with its set of registered parties $\mathcal{P}$ (denoted by $U_1, \ldots, U_{|\mathcal{P}|}$) as follows:

- **Key Generation.** Upon receiving a message (KeyGen, sid) from a stakeholder U_i, hand $(\mathsf{Respond}, \mathsf{KeyGen}, sid, U_i)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, sid, U_i, v)$ from the adversary, if U_i is honest, verify that v is unique, record the pair (U_i, v) and return $(\mathsf{VerificationKey}, sid, v)$ to U_i. Initialize the table $T(v, \cdot)$ to empty.
- **Malicious Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, sid, v)$ from $\mathcal{S}$, verify that v has not being recorded before; in this case initialize table $T(v, \cdot)$ to empty and record the pair $(\mathcal{S}, v)$.
- **VRF Evaluation.** Upon receiving a message (Eval, sid, m) from U_i, verify that some pair (U_i, v) is recorded. If not, then ignore the request. Then, if the value $T(v, m)$ is undefined, pick a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v, m) = (y, \emptyset)$. Then output $(\mathsf{Evaluated}, sid, y)$ to U_i, where y is such that $T(v, m) = (y, S)$ for some S.
- **VRF Evaluation and Proof.** Upon receiving a message $(\mathsf{EvalProve}, sid, m)$ from U_i, verify that some pair (U_i, v) is recorded. If not, then ignore the request. Else, send $(\mathsf{Respond}, \mathsf{EvalProve}, sid, U_i, m)$ to the adversary. Upon receiving $(\mathsf{EvalProve}, sid, m, \pi)$ from the adversary, if value $T(v, m)$ is undefined, verify that π is unique, pick a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v, m) = (y, \{\pi\})$. Else, if $T(v, m) = (y, S)$, set $T(v, m) = (y, S \cup \{\pi\})$. In any case, output $(\mathsf{Evaluated}, sid, y, \pi)$ to U_i.
- **Malicious VRF Evaluation.** Upon receiving a message $(\mathsf{Eval}, sid, v, m, \pi)$ from $\mathcal{S}$ for some v, do the following. First, if $(\mathcal{S}, v)$ is recorded and $T(v, m)$ is undefined, then choose a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v, m) = (y, S)$ and output $(\mathsf{Evaluated}, sid, y)$ to $\mathcal{S}$. The same is performed in case (U_i, v) is recorded and U_i corrupted. Else, if $T(v, m) = (y, S')$ for some $S' \neq \emptyset$, union S to S' and output $(\mathsf{Evaluated}, sid, y)$ to $\mathcal{S}$, else ignore the request.
- **Verification.** Upon receiving a message $(\mathsf{Verify}, sid, m, y, \pi, v')$ from some party P, send $(\mathsf{Respond}, \mathsf{Verify}, sid, m, y, \pi, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, y, \pi, v')$ from the adversary do:
  1. If $v' = v$ for some $(\cdot, v)$ and the entry $T(v, m)$ equals (y, S) with $\pi \in S$, then set $f = 1$.
  2. Else, if $v' = v$ for some recorded pair of the form $(\cdot, v)$, but no entry $T(v, m)$ of the form $(y, \{\ldots, \pi, \ldots\})$ is recorded, then set $f = 0$.

3. Else, initialize the table $T(v', \cdot)$ to empty, and set $f = 0$.
Output $(\mathsf{Verified}, sid, m, y, \pi, f)$ to P.

Key-Evolving Signatures. The idealized KES module is defined below:

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

$\mathcal{F}_{\mathsf{KES}}$ is parameterized by the total number of signature updates T, interacting with a signer U_S and registered parties in $\mathcal{P}$ (denoted by $U_1, \ldots, U_{|\mathcal{P}|}$) as follows:

- **Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from a stakeholder U_S , send $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, sid, U_S, v)$ from the adversary, verify that v is unique and send $(\mathsf{VerificationKey}, sid, v)$ to U_S, record the triple (sid, U_S, v) and set counter $\mathsf{k_{ctr}} = 1$.
- **Sign and Update.** Upon receiving a message $(\mathsf{USign}, sid, U_S, m, j)$ from U_S, verify that (sid, U_S, v) is recorded for some sid and that $\mathsf{k_{ctr}} \leq j \leq T$. If not, then ignore the request. Else, set $\mathsf{k_{ctr}} = j + 1$ and send $(\mathsf{Respond}, (\mathsf{Sign}, sid, U_S, m, j))$ to the adversary. Upon receiving $(\mathsf{Signature}, sid, U_S, m, j, \sigma)$ from the adversary, verify that no entry $(m, j, \sigma, v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send $(\mathsf{Signature}, sid, m, j, \sigma)$ to U_S, and record the entry $(m, j, \sigma, v, 1)$.
- **Signature Verification.** Upon receiving a message $(\mathsf{Verify}, sid, m, j, \sigma, v')$ from some stakeholder U_i do:
  1. If $v' = v$ and the entry $(m, j, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
  2. Else, if $v' = v$, the signer is not corrupted, and no entry $(m, j, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. (This condition guarantees unforgeability: If v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
  3. Else, if there is an entry (m, j, σ, v', f') recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
  4. Else, if $j < \mathsf{k_{ctr}}$, let $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. Otherwise, if $j = \mathsf{k_{ctr}}$, hand $(\mathsf{Verify}, sid, m, j, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, j, \phi)$ from the adversary let $f = \phi$ and record the entry (m, j, σ, v', ϕ). (This condition guarantees that the adversary is only able to forge signatures under keys belonging to corrupted parties for time periods corresponding to the current or future slots.)

  Output $(\mathsf{Verified}, sid, m, j, f)$ to U_i.

C A Complete Description of the Protocol

The purpose of this section is to specify more formally the code of the Chronos protocol for more clarity with respect to the security claims. We present the ledger protocol following the structure of other ledger protocols in UC [5,4].

C.1 The Main Protocol Instance

Ouroboros Chronos is a ledger-protocol and the main protocol dispatches to the relevant sub-processes. It accepts three kinds of input: inputs in order to register the party to the required setup and which models the dynamic availability of parties.

Registration. A party P can only start operation once it is registered to all resources. The registration handling is given in Section C.2. The protocol will initialize a party P's local time P.`localTime` to 0 and the protocol is aware that it is not synchronized (since existing participants might be far off) and sets P.`isSync` to false. Finally, in order to be able to recognize a new round, the party maintains a variable `lastTick` that stores the most recent tick from $\mathcal{G}_{\text{ImperfLClock}}$ (either 0 or 1).

Second, the ledger-specific queries to submit new transactions (SUBMIT), to read the ledger state (READ), to read the exported time value, and finally to make parties work and proceed with their "real round operations" to advance and maintain the ledger.

Submit transactions. As in [4] parties take as inputs transactions that serve as the inputs to the ledger.

Read state. As in [4], the ledger protocol exports a stable ledger state to the environment (implemented as a certain prefix of the longest chain of a party).

Read time. A novelty compared to Ouroboros Genesis, where the global clock showed the global time to all parties, Ouroboros Chronos will export a feature to read the logical protocol time. The exact guarantees on this "new clock", in particular the skew between reported times and the offset to a related "real-time" notion are given first as properties and then by the ideal ledger-functionality that Ouroboros Chronos realizes.

Activations requests. Note that MAINTAIN-LEDGER inputs are the activations that "make the parties work" and perform their round actions in the main procedure LedgerMaintenance later.

Finally, the protocol allows a caller controlled access to the features of the global shared setups through this protocol instance and to directly steer the registration status (to model the state in which a party is). We present the relevant sub-protocols and procedures in handling all these calls in the sequel. A summary of the state variables appears in Section M.1. Additionally, the protocol is designed to allow for a predictable number of activations per round and party to allow for a guaranteed advancement (independent of the adversarial actions) as defined in [4].

Technical remark: handling interrupts in a UC protocol. As a general paradigm to write the ledger protocol as a UC protocol, we follow the approach taken in [4] to simplify the treatments with interrupts in UC. Note that a protocol command might consists of a sequence of operations. In UC, certain operations, such as sending a message to another party or just the inability to conclude a task because a resource is unavailable, result into the protocol machine having to lose its activation. Thus, one needs a mechanism for ensuring that a party that looses the activation in the middle of such a multi-step command is able to resume and complete this command.

The general mechanism is to introduce an anchor a that stores a pointer to the current operation; the protocol associates each anchor with an input I, so that when such an input is received (again) it directly jumps to the stored anchor, executes the next operation(s) and updates (increases) the anchor before releasing the activation. We refer to execution in such a manner as *I-interruptible*.

Protocol Ouroboros-Chronos$_k$(P, sid; $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{G}_{\text{IMPERFLCLOCK}}$, $\mathcal{G}_{\text{RO}}$, $\mathcal{F}^{\Delta}_{\text{N-MC}}$)

Global Variables:

- Read-only (parameters): R, k, f, s, t_{off}, t_{stable}, t_{minSync},t_{pre}
- Read-write: $v_{\mathsf{P}}^{\text{vrf}}$, $v_{\mathsf{P}}^{\text{kes}}$, `localTime`, `ep`, `sl`, $\mathcal{C}_{\text{loc}}$, $T_{\mathsf{P}}^{\text{ep}}$, `isInit`, t_{work}, `buffer`, `futureChains`, `lastTick`, `isSync`, EpochUpdate($\cdot$), `fetchCompleted`, `lastTimeAlert`,Timestamp$_{SB}(\cdot)$. (recall that we use Timestamp($\cdot$) to denote the first (and numerical) element of the pair Timestamp$_{SB}(\cdot)$)

Registration/Deregistration:

- Upon receiving input (REGISTER, $\mathcal{R}$), where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{IMPERFLCLOCK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol Registration-Chronos(P, sid, `Reg`, $\mathcal{R}$).
- Upon receiving input (DE-REGISTER, $\mathcal{R}$), where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{IMPERFLCLOCK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol Deregistration-Chronos(P, sid, `Reg`, $\mathcal{R}$).
- Upon receiving input (IS-REGISTERED, sid) return (REGISTER, sid, 1) if the local registry `Reg` indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}_{\text{LEDGER}}$ (and did not de-register since then). Otherwise, return (REGISTER, 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** $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ **then** set $\texttt{buffer} \leftarrow \texttt{buffer}||\texttt{tx}$, and send $(\text{MULTICAST}, \text{sid}, \texttt{tx})$ to $\mathcal{F}_{\text{N-MC}}^{\Delta}$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \text{sid}, k, s, R, f)$; if $\mathsf{LedgerMaintenance}$ halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\text{READ}, \text{sid})$ **then** invoke protocol $\mathsf{ReadState}(k, \mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \text{sid}, R, f)$.
- **If** $I = (\text{EXPORT-TIME}, \text{sid})$ **then** do the following: if `isSync` or `isInit` is false, then return $(\text{EXPORT-TIME}, \text{sid}, \bot)$ to the caller. Otherwise call $\mathsf{UpdateTime}(\mathsf{P}, R)$ and do:
  1. Define e to be the highest value s.t. $\mathsf{EpochUpdate}(e) = \mathsf{Done}$.
  2. Return $(\text{EXPORT-TIME}, \text{sid}, (e, \texttt{localTime}))$ to the caller.

Handling calls to the shared setup:

- Upon receiving $(\text{CLOCK-GET}, \text{sid}_C)$ forward it to $\mathcal{G}_{\text{ImperfLClock}}$ and output $\mathcal{G}_{\text{ImperfLClock}}$'s response.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$, 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 $\mathsf{UpdateTime}(\mathsf{P}, R)$ to update `localTime` and evolve the KES signing key by sending $(\mathsf{USign}, \text{sid}, \mathsf{P}, 0, \texttt{localTime})$ to $\mathcal{F}_{\mathsf{KES}}$. If the party has passed a synchronization slot, then set $\texttt{isSync} \leftarrow \mathsf{false}$.
  2. If this instance is only stalled but $\texttt{isSync} = \mathsf{true}$, then additionally execute $\mathsf{FetchInformation}(\mathsf{P}, \text{sid})$, extract all new synchronization beacons $\mathcal{B}$ from the fetched chains and record their arrival times and set $\texttt{fetchCompleted} \leftarrow \mathsf{true}$. Also, any unfinished interruptible execution of this round is marked as completed.
  3. Forward $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{ImperfLClock}}$ to finally conclude the round.
- Upon receiving $(\text{EVAL}, \text{sid}_{RO}, x)$ forward the query to $\mathcal{G}_{\text{RO}}$ and output $\mathcal{G}_{\text{RO}}$'s response.

C.2 Registration, De-registration

A party P needs access to all its resources in order to start operation. Once it is registered to all resources it is able to perform basic operations. In contrast to Ouroboros Genesis, the protocol will initialize a party P's local time $\mathsf{P}.\texttt{localTime}$ to 0. Furthermore, the protocol is aware that it is not synchronized (since existing participants might be far off) and sets $\mathsf{P}.\texttt{isSync}$ to false. Finally, in order to be able to recognize a new round, the party maintains a variable `lastTick` that stores the most recent tick from $\mathcal{G}_{\text{ImperfLClock}}$ (either 0 or 1).

Protocol Registration-Chronos$(\mathsf{P}, \text{sid}, \texttt{Reg}, \mathcal{G})$

if $\mathcal{G} \in \{\mathcal{G}_{\text{ImperfLClock}}, \mathcal{G}_{\text{RO}}\}$ **then** send $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
end if
if $\mathcal{G} = \mathcal{G}_{\text{LEDGER}}$ **then**
    if the party is not registered with $\mathcal{G}_{\text{ImperfLClock}}$ or $\mathcal{G}_{\text{RO}}$ **then** or already registered with all setups ignore this input
    else
        for each $\mathcal{F} \in \{\mathcal{F}_{\mathsf{INIT}}^{\Delta}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}}\}$ **do**
            Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}$, set its registration status as registered with $\mathcal{F}$, but do not output the received values.
        end for
        Send $(\text{CLOCK-GET}, \text{sid}_C)$ to $\mathcal{G}_{\text{ImperfLClock}}$ and receive $(\text{CLOCK-GET}, \text{sid}_C, \text{tick})$ and set $\texttt{lastTick} \leftarrow \text{tick}$
        Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}_{\text{N-MC}}^{\Delta}$.
        Set $\texttt{localTime} := 0$ and $\texttt{isSync} \leftarrow \mathsf{false}$.

11: If this is the first registration invocation for this ITI, then set $\texttt{isInit} \leftarrow \mathsf{false}$.
12: Output $(\textsc{register}, \mathsf{sid}, \mathsf{P})$ once completing the registration with all the above resources $\mathcal{F}$.
end if
end if

De-registration is then the analogous action of setting variables to the initial values and record the synchronization status correctly:

Protocol Deregistration-Chronos$(\mathsf{P}, \mathsf{sid}, \texttt{Reg}, \mathcal{G})$

1: If the party is alert, set $\texttt{lastTimeAlert} \leftarrow \texttt{localTime}$
2: **if** $\mathcal{G} \in \{\mathcal{G}_{\text{ImperfLClock}}, \mathcal{G}_{\text{RO}}\}$ **then**
3: **if** $\mathcal{G} = \mathcal{G}_{\text{ImperfLClock}}$ **then** set $\texttt{isSync} \leftarrow \mathsf{false}$
4: Send $(\textsc{de-register}, \mathsf{sid})$ to $\mathcal{G}$ and set registration status as de-registered with $\mathcal{G}$.
5: Output the valued received by $\mathcal{G}$.
end if
6: **if** $\mathcal{G} = \mathcal{G}_{\text{ledger}}$ **then**
7: Set $\texttt{isSync} \leftarrow \mathsf{false}$
Send $(\textsc{de-register}, \mathsf{sid})$ to $\mathcal{F}_{\text{N-MC}}^{\Delta}$, set its registration status as de-registered with $\mathcal{F}_{\text{N-MC}}^{\Delta}$ and output $(\textsc{de-register}, \mathsf{sid}, \mathsf{P})$.
end if

C.3 Ledger Maintenance

What exactly a party might execute in a round depends on its status: newly registered parties first run through initialization and only later start to create blocks. All these steps are grouped in the main ledger operation procedure below and the relevant processes follow in the following paragraph. Compared to Genesis, the main round procedure is substantially more involved and we try to modularly adjust the code to see the difference.

Protocol LedgerMaintenance$(\mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \mathsf{sid}, k, s, R, f)$

The following steps are executed in an $(\textsc{maintain-ledger}, \mathsf{sid}, \text{minerID})$-interruptible manner:

1: **if** $\texttt{isInit}$ is false **then** invoke Initialization-Chronos$(\mathsf{P}, \mathsf{sid}, R)$; if Initialization-Chronos halts then halt (this will abort the execution)
end if
2: // From here the variables $v_{\mathsf{P}}^{\text{vrf}}, v_{\mathsf{P}}^{\text{kes}}, \texttt{localTime}, \texttt{ep}, \texttt{sl}, \mathcal{C}_{\mathsf{loc}}, \texttt{isSync}, T_{\mathsf{P}}^{\text{ep}}, T_{\mathsf{P}}^{\text{ep,bc}}$, $\texttt{fetchCompleted}$, t_{work} can be used to read from as they are guaranteed to be initialized.
3: **if** $\texttt{isSync}$ and stalled before (and now up and running) **then**
4: SimulateClockAdjustments(P, R, k, f, s)
5: **end if**
6: **if not** $\texttt{isSync}$ **then**
7: Call JoinProc$(\mathsf{P}, \mathsf{sid}, R, k, f, s, t_{\text{off}}, t_{\text{stable}}, t_{\text{minSync}})$
8: **end if**
9: // normal operation when alert
10: Call FetchInformation$(\mathsf{P}, \mathsf{sid})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\texttt{tx}_1, \ldots, \texttt{tx}_k)$.
11: Set $\texttt{buffer} \leftarrow \texttt{buffer}||(\texttt{tx}_1, \ldots, \texttt{tx}_k)$ and define $\texttt{futureChains} \leftarrow \texttt{futureChains}||(\mathcal{C}_1, \ldots, \mathcal{C}_M)$
12: Call UpdateTime(P, R)
13: // Ensures the processing of new beacons arrived in chains only.
14: Extract beacons $\mathcal{B} \leftarrow \{\texttt{SB}_1, \ldots, \texttt{SB}_n\}$ contained in $\mathcal{C}_1, \ldots, \mathcal{C}_M$ and not yet contained in $\texttt{syncBuffer}$.
15: Call ProcessBeacons$(\mathsf{P}, \mathsf{sid}, \mathcal{B})$
16: Let $\mathcal{N}_0$ be the subsequence of $\texttt{futureChains}$ s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \texttt{localTime}$

17: Remove each $\mathcal{C} \in \mathcal{N}_0$ from `futureChains`.
18: `fetchCompleted` $\leftarrow$ true
19: Call SelectChain$(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
20: **if** $t_{\mathrm{work}} <$ `localTime` **then**
21: Call UpdateStakeDist$(\mathsf{P}, \mathrm{sid}, k, R, f)$ to update the values $\mathbb{S}_{\mathsf{ep}}, \alpha_{\mathsf{P}}^{\mathsf{ep}}, T_{\mathsf{P}}^{\mathsf{ep}}$, $T_{\mathsf{P}}^{\mathsf{ep,bc}}$ and η_{ep}.
22: Call StakingProcedure$(k, \mathsf{P}, \mathtt{ep}, \mathtt{sl}, \mathtt{buffer}, \mathcal{C}_{\mathsf{loc}})$
23: Set $t_{\mathrm{work}} \leftarrow$ `localTime`
24: **if** `localTime` $\bmod R = 0$ **then**
25: Call SyncProc$(\mathsf{P}, \mathrm{sid}, R)$
26: **end if**
end if
27: Call FinishRound(P) // Mark normal round actions as finished.

Standard Round Operations. Before discussing the involved special case of joining, let us first define all relevant procedures for an alert party to complete its task per round as described in Section 5 of the main body, that is:

- Fetch information from the network (by a call to FetchInformation).
- Update the time (by a call to UpdateTime): the party locally advances its time-stamp whenever it realizes that a new round has started by a call $\mathcal{G}_{\text{IMPERFLCLOCK}}$ and comparing it to `lastTick`.
- Record the arrival times of the synchronization beacons the protocol sends out (call to ProcessBeacons).
- Process the received chains: as some chains might be created by parties whose time-stamps might be ahead, the future chains are stored in the buffer `futureChains` for later usage. Among the remaining chains, the protocol will according to the Genesis chain-selection rule decide whether any chain is more preferable than the local chain (procedure SelectChain).
- Run the main staking procedure (StakingProcedure) to evaluate slot leadership, and potentially create and emit a new block or synchronization beacon. Before the main staking procedure is executed, the local state is updated including the current stake distribution (call to UpdateStakeDist).
- If the end of the round coincides with the end of an epoch, the synchronization procedure is executed. This core procedure of our proposal is detailed below.

The above round operations are based on several core procedures of blockchain protocols, most notably three procedures we introduce next:

- The initialization algorithm to properly define the initial values.
- The verification algorithms to evaluate whether a chain is valid, and for the specific case of Ouroboros Chronos also whether a beacon is valid.

C.4 Initialization

The first special procedure a party runs through is initialization. It is invoked upon the first MAINTAIN-LEDGER input given to this instance. Since every party starts at local time 0 and has no knowledge whether the session is already running, it will first try to claim stake from $\mathcal{F}_{\mathsf{INIT}}$ in its first round and then in subsequent rounds, tries to receive the genesis block until it is successful. As a difference to Ouroboros Genesis, a party might be delayed in receiving the genesis block. In any case, once a party obtains the genesis block it will initialize the variables of this instance which are described in Table M.1. In particular, as specified by our setup $\mathcal{F}_{\mathsf{INIT}}$ if the genesis block is delivered to an inaugural party of the system, then the party considers itself as synchronized.[19] Otherwise it has to invoke the joining procedure.

[19] Note that this knowledge is needed to bootstrap the system with a set of alert parties.

Protocol Initialization-Chronos($\mathsf{P}, \mathrm{sid}, R$)

The following steps are executed in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:

1: Send $(\mathsf{KeyGen}, sid, \mathsf{P})$ to $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$; receiving $(\mathsf{VerificationKey}, sid, v_{\mathsf{P}}^{\mathrm{vrf}})$ and $(\mathsf{VerificationKey}, sid, v_{\mathsf{P}}^{\mathrm{kes}})$, respectively.// Note that this can be seen as defining the party's address $\langle v_p^{\mathrm{vrf}}, v_p^{\mathrm{kes}} \rangle$ (where $\langle\rangle$ denotes an encoding).
2: **if** `localTime` $= 0$ **then** // Right after registration of this instance
3: Send $(\mathsf{ver\_keys}, sid, \mathsf{P}, v_{\mathsf{P}}^{\mathrm{vrf}}, v_{\mathsf{P}}^{\mathrm{kes}})$ to $\mathcal{F}_{\mathsf{INIT}}$ to claim stake from the genesis block.
4: FinishRound(P) // Mark round actions as finished. Resume below upon next activation
5: Call UpdateTime(P, R, f) to update `localTime`, `ep`, and `sl`
6: **while** `localTime` $= 0$ **do**
7: Call UpdateTime(P, R, f) to update `localTime`, `ep`, and `sl` and give up the activation (set anchor here)
end while
8: **end if**
// The following is executed in future init-activations of this instance
9: **if** `localTime` > 0 **then**
10: **if** $\mathcal{F}_{\mathsf{INIT}}$ signals an error **then**
11: Halt the execution.
12: **end if**
13: Send $(\mathsf{genblock\_req}, sid, \mathsf{P})$ to $\mathcal{F}_{\mathsf{INIT}}$.
14: **while** $\mathcal{F}_{\mathsf{INIT}}$ ignores the input **do**
15: FinishRound(P) // Round actions as finished. Resume below upon next activation
16: Send $(\mathsf{genblock\_req}, sid, \mathsf{P})$ to $\mathcal{F}_{\mathsf{INIT}}$.
17: **end while**
18: Receive the genesis block $(\mathsf{genblock}, sid, \mathbf{G} = (\mathbb{S}_1, \eta_1))$, where

$$\mathbb{S}_1 = \Big((U_1, v_1^{\mathrm{vrf}}, v_1^{\mathrm{kes}}, s_1), \ldots, (U_n, v_n^{\mathrm{vrf}}, v_n^{\mathrm{kes}}, s_n) \Big) \ .$$

19: Set $\mathcal{C}_{\mathsf{loc}} \leftarrow (\mathbf{G})$.
20: If $\mathcal{F}_{\mathsf{INIT}}$ did not mark the returned value as Running (the execution just started) and this instance was time-aware and online from registration onward, then additionally set `isSync` $\leftarrow$ true // This party becomes a synchronized inaugural protocol participant.
end if
21: Set `isInit` $\leftarrow$ true, `fetchCompleted` $\leftarrow$ false, $t_{\mathrm{work}} \leftarrow 0$, `lastTimeAlert` $\leftarrow 0$, `localTime`, `sl`, `ep` $\leftarrow 1$ and call UpdateStakeDist($\mathsf{P}, \mathrm{sid}, k, R, f$) to initialize the values $\mathbb{S}_{\mathsf{ep}}, \alpha_{\mathsf{P}}^{\mathsf{ep}}, T_{\mathsf{P}}^{\mathsf{ep}}, T_{\mathsf{P}}^{\mathsf{ep,bc}}$ and η_{ep}. // Now, ready to start, either with the first slot in first epoch, or with bootstrapping (joining procedure).
22: `buffer` $\leftarrow \emptyset$, `futureChains` $\leftarrow \emptyset$
23: EpochUpdate($\cdot$) $\leftarrow$ empty table (initial symbol $\perp$), EpochUpdate(0) $\leftarrow$ Done

C.5 Validity Checks of Chains and Beacons in Chronos

Chain verification. A core procedure is to distinguish valid from invalid blockchains. The procedure is depicted below. It is a slightly extended version of Ouroboros Genesis to cover beacons.

Protocol IsValidChain($\mathsf{P}, \mathrm{sid}, \mathcal{C}, f, R$)

if $\mathcal{C}$ contains empty epochs or starts with a block other than $\mathbf{G}$, or isvalidstate(`st`) $= 0$ **then**
return false
end if
if `isSync` and $(\exists B \in \mathcal{C} : \mathsf{slotnum}(B) >$ `localTime`$)$ **then**
return false
end if
for each epoch `ep` **do**

// Derive stake distribution and randomness for this epoch from $\mathcal{C}$
// In the following, $H(\cdot)$ stands for an RO evaluation for simplicity.
Set $\mathbb{S}^{\mathcal{C}}_{\mathtt{ep}}$ to be the stakeholder distribution at the end of epoch $\mathtt{ep}-2$ in $\mathcal{C}$.
Set $\alpha^{\mathtt{ep},\mathcal{C}}_{\mathsf{P}'}$ to be the relative stake of any party P' in $\mathbb{S}^{\mathcal{C}}_{\mathtt{ep}}$ and $T^{\mathtt{ep},\mathcal{C}}_{\mathsf{P}'} \leftarrow 2^{\ell_{\mathsf{VRF}}}\phi_f(\alpha^{\mathtt{ep},\mathcal{C}}_{\mathsf{P}'})$. Also define $T^{\mathtt{ep},\mathtt{bc},\mathcal{C}}_{\mathsf{P}'} \leftarrow 2^{\ell_{\mathsf{VRF}}} \cdot \alpha^{\mathtt{ep},\mathcal{C}}_{\mathsf{P}'}$.
Set $\eta^{\mathcal{C}}_{\mathtt{ep}} \leftarrow H(\eta^{\mathcal{C}}_{\mathtt{ep}-1} \,\|\, \mathtt{ep} \,\|\, v)$ where v is the concatenation of the VRF outputs y_ρ from all blocks in $\mathcal{C}$ from the first two-thirds of slots of epoch $\mathtt{ep}-1$, and $\eta^{\mathcal{C}}_1 \triangleq \eta_1$ from $\mathbf{G}$.
for each block B in $\mathcal{C}$ from epoch $\mathtt{ep}$ **do**
Parse B as $(h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$.
// Check hash
Set $\mathsf{badhash} \leftarrow (h \neq H(B^{-1}))$, where B^{-1} is the last block in $\mathcal{C}$ before B.
// Check VRF values
Parse crt as (P', y, π) for some P'.
Send $(\mathsf{Verify}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST}, y, \pi, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$,
denote its response by $(\mathsf{Verified}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST}, y, \pi, b_1)$.
Send $(\mathsf{Verify}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$,
denote its response by $(\mathsf{Verified}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, b_2)$,
Set $\mathsf{badvrf} \leftarrow \left(b_1 = 0 \vee b_2 = 0 \vee y \geq T^{\mathtt{ep},\mathcal{C}}_{\mathsf{P}'}\right)$.
// Check signature
Send $(\mathsf{Verify}, sid, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, \sigma, v^{\mathrm{kes}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{KES}}$,
denote its response by $(\mathsf{Verified}, sid, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, b_3)$.
Set $\mathsf{badsig} \leftarrow (b_3 = 0)$.
// Check Beacons
if $\exists \mathtt{SB} \in B \wedge \mathsf{slotnum}(B) > (\mathtt{ep}-1)R + 2R/3$ **then**
Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
else if $\exists \mathtt{SB} \in B : \mathsf{slotnum}(\mathtt{SB}) > \mathsf{slotnum}(B) \vee \mathsf{slotnum}(\mathtt{SB}) \notin [(\mathtt{ep}-1)R+1, \mathtt{ep} \cdot R]$ **then**
Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
else
for each $\mathtt{SB} \in B$ **do**
Parse $\mathtt{SB}$ as $(\mathtt{sl}', \mathsf{P}', y, \pi)$
If $\mathcal{C}$ contains more than one beacon with $(\mathtt{sl}', \mathsf{P}', \cdot, \cdot)$ then set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
Send $(\mathsf{Verify}, sid, \eta_{\mathtt{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{SYNC}, y, \pi, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$.
Denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Verified}, sid, \eta_{\mathtt{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{SYNC}, y, \pi, b_4)$,
if $(b_4 = 0)$ or $(y \geq T^{\mathtt{ep},\mathtt{bc},\mathcal{C}}_{\mathsf{P}'})$ **then**
Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
end if
end for
end if
if $(\mathsf{badhash} \vee \mathsf{badvrf} \vee \mathsf{badsig} \vee \mathsf{badBeacon})$ **then**
return false
end if
end for
end for
return truek

The beacon validity predicate. Beacons validity is related to chain validity as one has to verify validity of leadership. The details are found below:

Protocol $\mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}, \mathcal{C}, f, R)$

// Precondition: Chain $\mathcal{C}$ is valid. Returns true if the beacon is a valid beacon w.r.t. $\mathcal{C}$, $\mathsf{undecided}$ if no judgement is possible, and false if the beacon is invalid w.r.t. $\mathcal{C}$.

Parse $\mathtt{SB}$ as $(\mathtt{sl}', \mathsf{P}', y, \pi)$
Let ep' be the epoch number slot $\mathtt{sl}'$ falls into. Let $ep := ep' - 2$.
if $\mathcal{C}$ contains no block in epoch ep' **then**
  return undecided // no judgement possible for this beacon
end if
// Derive stake distribution and randomness for epoch ep' as indicated by $\mathcal{C}$
Set $\mathbb{S}^{\mathcal{C}}_{ep'}$ to be the stakeholder distribution at the end of epoch $ep' - 2$ in $\mathcal{C}$.
Set $\alpha^{ep',\mathcal{C}}_{\mathsf{P}'}$ to be the relative stake of party P' in $\mathbb{S}^{\mathcal{C}}_{ep'}$ and $T^{ep',\mathsf{bc},\mathcal{C}}_{\mathsf{P}'} \leftarrow 2^{\ell_{\mathsf{VRF}}} \cdot \alpha^{ep',\mathcal{C}}_{\mathsf{P}'}$.
Set $\eta^{\mathcal{C}}_{ep'} \leftarrow H(\eta^{\mathcal{C}}_{ep'-1} \,\|\, ep' \,\|\, v)$ where v is the concatenation of the VRF outputs y_ρ from the existing blocks in $\mathcal{C}$ with slot numbers of the first two-thirds slots of epoch $ep' - 1$ (and $\eta^{\mathcal{C}}_1 \triangleq \eta_1$ from $\mathbf{G}$).
// Check VRF value
Send $(\mathsf{Verify}, sid, \eta_{ep'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{SYNC}, y, \pi, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$.
Denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Verified}, sid, \eta_{ep'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{SYNC}, y, \pi, b_1)$,
if $b_1 = 0$ or $y \geq T^{ep',\mathsf{bc},\mathcal{C}}_{\mathsf{P}'}$ **then**
  return false
end if
return true

C.6 Fetching information, stake distribution and time update

The two algorithms FetchInformation and UpdateTime are basically identical to Ouroboros Genesis except that FetchInformation was simplified since newly joining parties do not make an active request.

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

1: **if** `fetchCompleted` **then**
2: Set fetchcount $\leftarrow 0$
3: **else**
4: Set fetchcount $:= 1$ // Compared to Genesis, time-aware and online parties in Chronos do always fetch once per round and never have to catch up missed round messages.
5: **end if**
// Fetching on $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$.
6: Send fetchcount fetch-queries $(\text{FETCH}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$; denote the ith response from $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ by $(\text{FETCH}, \mathrm{sid}, b_i)$.
7: Extract chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from $b_1 \ldots b_{\mathsf{fetchcount}}$.
// Fetching on $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
8: Send fetchcount fetch-queries $(\text{FETCH}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$; denote the ith response from $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ by $(\text{FETCH}, \mathrm{sid}, b_i)$.
9: Extract received transactions $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ from $b_1 \ldots b_{\mathsf{fetchcount}}$.
10: **if not** `isSync` or P is stalled **then**
11: `buffer` $\leftarrow$ `buffer`$||(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
12: `futureChains` $\leftarrow$ `futureChains` $\cup \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$
13: **end if**

OUTPUT: The protocol outputs $(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ and $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ to its caller (but not to $\mathcal{Z}$).

The stake distributions for epochs defined in the local chain (and all associated state-variables) are computed as follows:

Protocol UpdateStakeDist(P, k, R, f)

1: Set $\mathbb{S}_{\mathsf{ep}}$ to be the stakeholder distribution at the end of epoch $\mathsf{ep} - 2$ in $\mathcal{C}_{\mathsf{loc}}$ in case $\mathsf{ep} \geq 2$ (and keep the initial stake distribution in case $\mathsf{ep} < 2$).
2: Set $\alpha^{\mathsf{ep}}_{\mathsf{P}}$ to be the relative stake of P in $\mathbb{S}_{\mathsf{ep}}$ and $T^{\mathsf{ep}}_{\mathsf{P}} \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha^{\mathsf{ep}}_{\mathsf{P}})$ as well as $T^{\mathsf{ep},\mathsf{bc}}_{\mathsf{P}} \leftarrow 2^{\ell_{\mathsf{VRF}}} \cdot \alpha^{\mathsf{ep}}_{\mathsf{P}}$

3: Set $\eta_{\mathsf{ep}} \leftarrow H(\eta_{\mathsf{ep}-1} \,\|\, \mathsf{ep} \,\|\, v)$ where v is the concatenation of the VRF outputs y_ρ from all blocks in $\mathcal{C}_{\mathsf{loc}}$ from the first $2R/3$ slots of epoch $\mathsf{ep}-1$ (and if $\mathsf{ep}=1$, η_1 is the nonce of the genesis block).

OUTPUT: The protocol outputs $\mathbb{S}_{\mathsf{ep}}, \alpha_{\mathsf{P}}^{\mathsf{ep}}, T_{\mathsf{P}}^{\mathsf{ep}}, \eta_{\mathsf{ep}}$ and $T_{\mathsf{P}}^{\mathsf{ep,bc}}$ to its caller (but not to $\mathcal{Z}$).

And, finally, the time update procedure:

Protocol UpdateTime(P, R)

// Precondition: Only executed if time-aware
1: Send (CLOCK-GET, sid_C) to $\mathcal{G}_{\text{IMPERFLCLOCK}}$ and receive (CLOCK-GET, sid_C, tick)
2: **if** `lastTick` $\neq$ tick **then**
3: `lastTick` $\leftarrow$ tick
4: `localTime` $\leftarrow$ `localTime` $+ 1$
5: `fetchCompleted` $\leftarrow$ false
6: **end if**
7: Set `ep` $\leftarrow \lceil$`localTime`$/R\rceil$, and `sl` $\leftarrow$ `localTime`.

OUTPUT: The protocol outputs `localTime`, `ep`, `sl` to its caller (but not to $\mathcal{Z}$).

C.7 Process Beacons and Arrival Times

The following procedure records and processes beacons, their arrival times, and filters out invalid beacons. Special care needs to be made to properly filter out bogus beacons as soon as possible. The validity predicate for beacons follows in the next section, where we discuss all validity predicates in Ouroboros Chronos.

Protocol ProcessBeacons(P, sid, $\mathcal{B}$)

1: **if not** `fetchCompleted` **then**
2: Send (FETCH, sid) to $\mathcal{F}_{\text{N-MC}}^{\text{sync}}$. denote the ith response from $\mathcal{F}_{\text{N-MC}}^{\text{sync}}$ by (FETCH, sid, b).
3: Extract all received beacons $(\mathtt{SB}_1, \ldots, \mathtt{SB}_k)$ contained in $b \cup \mathcal{B}$.
4: **for each** $\mathtt{SB}_i$ with $\mathsf{Timestamp}_{SB}(\mathtt{SB}) = \perp$ **do**
5: `syncBuffer` $\leftarrow$ `syncBuffer` $\cup \{\mathtt{SB}\}$
6: Let ep be the epoch number $\mathsf{slotnum}(\mathtt{SB})$ belongs to
7: **if** `isSync` $\wedge$ ($\mathsf{EpochUpdate}(ep-1) = \mathsf{Done}$) **then**
8: Set $\mathsf{Timestamp}_{SB}(\mathtt{SB}_i) \leftarrow$ (`localTime`, final). // The measurement is final.
9: **else**
10: $\mathsf{Timestamp}_{SB}(\mathtt{SB}_i) \leftarrow$ (`localTime`, temp) // Will be adjusted upon next time shift.
11: **end if**
12: **end for**
13: // Buffer cleaning. Keep one representative arrival time.
14: **if** `isSync` **then**
15: Remove from `syncBuffer` all beacons such that $\mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}, \mathcal{C}_{\mathsf{loc}}, f, R)$ returns false.
16: $\mathtt{syncBuffer}_{\text{valid}} \leftarrow \{\mathtt{SB}' \in \mathtt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}', \mathcal{C}_{\mathsf{loc}}, f, R) = \mathsf{true}\}$
17: Let $L = (\mathtt{SB}_1, \ldots, \mathtt{SB}_n)$ be a canonical ordering of $\mathtt{syncBuffer}_{\text{valid}}$
18: **for** each $\mathtt{SB} = (\mathtt{sl}, \mathsf{P}, y, \pi) \in L$ **do**
19: $Q_{\mathtt{SB}} \leftarrow \{\mathtt{SB}' = (\mathtt{sl}', \mathsf{P}', \cdot, \cdot) \in L \mid \mathsf{P}' = \mathsf{P} \wedge \mathtt{sl}' = \mathtt{sl}\}$
20: $\min_{\mathtt{SB}} \leftarrow \min\{\mathsf{Timestamp}(\mathtt{SB}') \mid \mathtt{SB}' \in Q_{\mathtt{SB}}\}$
21: $\mathtt{SB}' \leftarrow \min\{\mathtt{SB}'' \in Q_{\mathtt{SB}} \mid \mathsf{Timestamp}(\mathtt{SB}'') = \min_{\mathtt{SB}}\}$ // Min w.r.t. ordering in L
22: Remove from `syncBuffer` all beacons $(\mathtt{sl}, \mathsf{P}, \cdot, \cdot)$ except $\mathtt{SB}'$.
23: **end for**
24: **end if**
25: **end if**

OUTPUT: ok to its caller (but not to $\mathcal{Z}$).

C.8 Select Chain

Chain selection consists of two steps: filtering out valid chains, and second compare them using the Genesis rule.

Protocol $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N} = \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}, k, s, R, f)$

1: Initialize $\mathcal{N}_{\mathrm{valid}} \leftarrow \emptyset$
2: **for** $i = 1 \ldots M$ **do**
3: Invoke $\mathsf{IsValidChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_i, f, R)$; if it returns true then update $\mathcal{N}_{\mathrm{valid}} \leftarrow \mathcal{N}_{\mathrm{valid}} \cup \mathcal{C}_i$
end for
4: Execute Algorithm $\mathsf{maxvalid\text{-}bg}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_{\mathrm{valid}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}, k, s, f)$ and receive its output $\mathcal{C}_{\mathsf{max}}$.
5: Replace $\mathcal{C}_{\mathsf{loc}}$ by $\mathcal{C}_{\mathsf{max}}$

OUTPUT: The protocol outputs $\mathcal{C}_{\mathsf{max}}$ to its caller (but not to $\mathcal{Z}$).

C.9 The Genesis Chain Selection Rule

The genesis rule is given below:

Algorithm $\mathsf{maxvalid\text{-}bg}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N} = \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}, k, s, f)$

// Compare $\mathcal{C}_{\mathsf{max}}$ to each $\mathcal{C}_i \in \mathcal{N}$
1: Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_{\mathsf{loc}}$.
2: **for** $i = 1$ to M **do**
3: **if** ($\mathcal{C}_i$ forks from $\mathcal{C}_{\mathsf{max}}$ at most k blocks) **then**
4: **if** $|\mathcal{C}_i| > |\mathcal{C}_{\mathsf{max}}|$ **then** // Condition A
Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$.
end if
5: **else**
6: Let $j \leftarrow \max\{j' \geq 0 \mid \mathcal{C}_{\mathsf{max}}$ and $\mathcal{C}_i$ have the same block in $\mathtt{sl}_{j'}\}$
7: **if** $\left|\mathcal{C}_i[0 : j + s]\right| > \left|\mathcal{C}_{\mathsf{max}}[0 : j + s]\right|$ **then** // Condition B
Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$.
end if
end if
end for
8: **return** $\mathcal{C}_{\mathsf{max}}$.

C.10 The Core Staking Procedure of Alert Parties

Once a party has properly filtered all information and updated its state, it can run the core staking procedure and beacon emitting process formally given below:

Protocol $\mathsf{StakingProcedure}(\mathsf{P}, \mathrm{sid}, k, \mathtt{ep}, \mathtt{sl}, \mathtt{buffer}, \mathcal{C}_{\mathsf{loc}})$

The following steps are executed in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:

// Determine leader status
1: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y_\rho, \pi_\rho)$.
2: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{SYNC})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y_{\mathrm{sync}}, \pi_{\mathrm{sync}})$.
3: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$.
4: **if** $y < T_{\mathsf{P}}^{\mathsf{ep}}$ and this party is *sign-capable* **then**
// Generate a new block

Set $\texttt{buffer}' \leftarrow \texttt{buffer}$, $\boldsymbol{N} \leftarrow \texttt{tx}_{\mathsf{P}}^{\text{base-tx}}$, and $\texttt{st} \leftarrow \mathsf{blockify}_{\mathsf{OC}}(\boldsymbol{N})$
repeat
  Parse $\texttt{buffer}'$ as sequence $(\texttt{tx}_1, \ldots, \texttt{tx}_n)$
  for $i = 1$ to n **do**
    if $\mathsf{ValidTx}_{\mathsf{OC}}(\texttt{tx}_i, \texttt{st}||\texttt{st}) = 1$ **then**
      $\boldsymbol{N} \leftarrow \boldsymbol{N}||\texttt{tx}_i$
      Remove $\texttt{tx}$ from $\texttt{buffer}'$
      Set $\texttt{st} \leftarrow \mathsf{blockify}_{\mathsf{OC}}(\boldsymbol{N})$
    end if
  end for
until $\boldsymbol{N}$ does not increase anymore
Set $crt = (\mathsf{P}, y, \pi)$, $\rho = (y_\rho, \pi_\rho)$ and $h \leftarrow H(\texttt{head}(\mathcal{C}_{\mathsf{loc}}))$.
if Slot $\texttt{sl}$ is within the first $2R/3$ slots of this epoch **then**
  $\texttt{sb} \leftarrow \{\texttt{SB}' \in \texttt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \text{sid}, \texttt{SB}', \mathcal{C}_{\mathsf{loc}}, f, R) = \mathsf{true}\}$
  Remove from $\texttt{sb}$ all beacons $\texttt{SB} = (\texttt{sl}, \mathsf{P}, \cdot, \cdot)$ that satisfy:
    $(\mathsf{slotnum}(\texttt{SB}) > \texttt{sl}) \vee (\mathsf{slotnum}(\texttt{SB}) \leq (\texttt{ep} - 1) \cdot R) \vee \mathcal{C}_{\mathsf{loc}}$ contains a beacon $(\texttt{sl}, \mathsf{P}, \cdot, \cdot)$
end if
Send $(\mathsf{USign}, \text{sid}, \mathsf{P}, (h, \texttt{st}, \texttt{sb}, \texttt{sl}, crt, \rho), \texttt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$; denote the response from $\mathcal{F}_{\mathsf{KES}}$ by $(\mathsf{Signature}, \text{sid}, (h, \texttt{st}, \texttt{sb}, \texttt{sl}, crt, \rho), \texttt{sl}, \sigma)$.
Set $B \leftarrow (h, \texttt{st}, \texttt{sb}, \texttt{sl}, crt, \rho, \sigma)$ and update $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,||\, B$.
// Multicast the extended chain and wait.
Send $(\text{MULTICAST}, \text{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and proceed from here upon next activation of this procedure.
else
  Evolve the KES signing key at least to $\texttt{localTime}$ by sending $(\mathsf{USign}, \text{sid}, \mathsf{P}, 0, \texttt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$ (and ignore the returned value). Give up activation and set anchor here to resume on next maintenance activation
end if
if $y_{\text{sync}} < T_{\mathsf{P}}^{\mathsf{ep,bc}}$ and $\texttt{sl}$ lies within the first $R/6$ slots of this epoch **then**
  $\texttt{SB} \leftarrow (\texttt{sl}, \mathsf{P}, y_{\text{sync}}, \pi_{\text{sync}})$.
  Send $(\text{MULTICAST}, \text{sid}, \texttt{SB})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$ and set anchor at end of procedure to resume on next maintenance activation
else
  Give up activation and set anchor at end of procedure to resume on next maintenance activation
end if

C.11 Code of the Synchronization Procedure

The synchronization procedure is run at an epoch boundary, and the code is given below:

Protocol $\mathsf{SyncProc}(\mathsf{P}, \text{sid}, R)$

// Only called when: P is alert, $\texttt{localTime} \mod R = 0$ and $\texttt{localTime} > 0$
Set $i \leftarrow \texttt{localTime} \operatorname{div} R$
if (**not** $\mathsf{EpochUpdate}(i) = \mathsf{Done}$) **then**
  $\mathsf{EpochUpdate}(i) \leftarrow \mathsf{Done}$ // Remember that clock adjustment has happened
  $\mathcal{B}_i \leftarrow \mathcal{C}_{\mathsf{loc}}[(i-1)R : (i-1)R + 2R/3]$
  $\mathcal{S}_i \leftarrow \{\texttt{SB} \mid \exists B \in \mathcal{B}_i : \texttt{SB} \in B \wedge \mathsf{slotnum}(\texttt{SB}) \in \{(i-1)R, \ldots, (i-1)R + R/6\}\}$
  for each $\texttt{SB} = (\texttt{sl}, \mathsf{P}, y, \pi) \in \mathcal{S}_i$ **do**
    // Find representative beacon and compute recommendation.
    Find unique $\texttt{SB}' = (\texttt{sl}, \mathsf{P}, \cdot, \cdot) \in \texttt{syncBuffer}$. If inexistent, set $\texttt{SB}' \leftarrow \perp$.
    if $\texttt{SB}' \neq \perp$ **then**
      Set $\mathsf{Timestamp}_{SB}(\texttt{SB}) \leftarrow \mathsf{Timestamp}_{SB}(\texttt{SB}')$ // Assign correct value
      $\mathsf{recom}(\texttt{SB}) \leftarrow \mathsf{slotnum}(\texttt{SB}) - \mathsf{Timestamp}(\texttt{SB})$
    else

$\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathtt{SB}\}$ // Negligible probability event in execution.
end if
end for
$\mathsf{shift}_i \leftarrow \mathsf{med}\,\{\mathsf{recom}(\mathtt{SB}) \mid \mathtt{SB} \in \mathcal{S}_i\}$
for each $\mathtt{SB}$ with $\mathsf{Timestamp}_{SB}(\mathtt{SB}) = (a, \mathsf{temp})$ **do**
$\mathsf{Timestamp}_{SB}(\mathtt{SB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{final})$
end for
if $\mathsf{shift}_i > 0$ **then** // Move fast forward
$\mathsf{newTime} \leftarrow \mathtt{localTime} + \mathsf{shift}_i$
$\mathcal{M}_{\text{chains}} \leftarrow \mathcal{M}_{\text{sync}} \leftarrow \emptyset$
while $\mathtt{localTime} < \mathsf{newTime}$ **do**
$\mathtt{localTime} \leftarrow \mathtt{localTime} + 1$
Let $\mathcal{N}_0$ be the subsequence of $\mathtt{futureChains}$ s.t.
$\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \mathtt{localTime}$
Remove each $\mathcal{C} \in \mathcal{N}_0$ from $\mathtt{futureChains}$.
Call $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
Call $\mathsf{UpdateStakeDist}(\mathsf{P}, k, R, f)$
Emulate $\mathsf{StakingProcedure}(k, \mathsf{P}, \mathtt{ep}, \mathtt{sl}, \mathtt{buffer}, \mathcal{C}_{\mathsf{loc}})$ but instead of multicasting new chains or beacons, add them to the sets $\mathcal{M}_{\text{chains}}$ and $\mathcal{M}_{\text{sync}}$, respectively (activation is not lost).
end while
Send $(\textsc{multicast}, \mathsf{sid}, \mathcal{M}_{\text{chains}})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and $(\textsc{multicast}, \mathsf{sid}, \mathcal{M}_{\text{sync}})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$ and proceed from here upon next activation of this procedure.
end if
if $\mathsf{shift}_i < 0$ **then** // Need to wait
Set $t_{\text{work}} \leftarrow \mathtt{localTime}$
Set $\mathtt{localTime} \leftarrow \mathtt{localTime} + \mathsf{shift}_i$ // Next slot in which staking will be performed is slot $\mathtt{localTime} + 1$ according to the "new time".
end if
end if

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

C.12 Reading the Ledger State

Reading the ledger state is rather straightforward: first process all relevant information for this round and then extract the state.

Protocol $\mathsf{ReadState}(k, \mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \mathrm{sid}, R, f)$

If any of $\mathtt{isInit}$ or $\mathtt{isSync}$ is false output the empty state $(\textsc{read}, \mathrm{sid}, \varepsilon)$ (to $\mathcal{Z}$). Otherwise, do the following:
Call $\mathsf{FetchInformation}(k, \mathsf{P})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$.
Set $\mathtt{buffer} \leftarrow \mathtt{buffer} || (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and define $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$
Call $\mathsf{UpdateTime}(\mathsf{P}, R)$
Call $\mathsf{ProcessBeacons}(\mathsf{P}, \mathrm{sid})$
Let $\mathcal{N}_0 := \{\mathcal{C} \in \mathcal{N} \cup \mathtt{futureChains} \mid \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \mathtt{localTime}\}$
Let $\mathcal{N}_1 := \{\mathcal{C} \in \mathcal{N} \mid \exists B \in \mathcal{C} : \mathsf{slotnum}(B) > \mathtt{localTime}\}$
$\mathtt{futureChains} \leftarrow (\mathtt{futureChains} \setminus \mathcal{N}_0) \cup \mathcal{N}_1$
$\mathtt{fetchCompleted} \leftarrow \mathsf{true}$
Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
Extract the state $\mathtt{st}$ from the current local chain $\mathcal{C}_{\mathsf{loc}}$.
Output $(\textsc{read}, \mathrm{sid}, \mathtt{st}^{\lceil k})$ (to $\mathcal{Z}$). // $\mathtt{st}^{\lceil k}$ denotes the prefix of $\mathtt{st}$ with the last k state blocks chopped off

C.13 Simulate Clock Adjustments

Parties that have been absent only for a limited time (i.e., stalled for a limited time) can bootstrap very easily to the actual reliable state and time. Note that after doing this, they might be not yet alert for a small amount of time until they can start to work again, since they might have evolved their keys too far.

Protocol SimulateClockAdjustments(P, R, k, f, s)

1: $\mathsf{simulatedTime} \leftarrow \texttt{lastTimeAlert}$
2: **for** $\texttt{localTime} - \texttt{lastTimeAlert}$ iterations **do**
3: Let $\mathcal{N}_0$ be the subsequence of $\texttt{futureChains}$ s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \mathsf{simulatedTime}$
4: Remove each $\mathcal{C} \in \mathcal{N}_0$ from $\texttt{futureChains}$.
5: Emulate $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ with simulated time $\mathsf{simulatedTime}$ (instead of $\texttt{localTime}$)
6: - Update $\mathcal{C}_{\mathsf{loc}}$
7: **if** $\mathsf{simulatedTime} \mod R = 0$ **then**
8: Emulate $\mathsf{SyncProc}(\mathsf{P}, R)$ on simulated time $\mathsf{simulatedTime}$ (instead of $\texttt{localTime}$)
9: - Execute Lines 1 to 13 to compute the shift $\mathsf{shift}_{\mathsf{ep}}$ and to adjust already recorded arrival times.
10: - Set $\mathsf{simulatedTime} \leftarrow \mathsf{simulatedTime} + \mathsf{shift}_{\mathsf{ep}}$
11: **end if**
12: $\mathsf{simulatedTime} \leftarrow \mathsf{simulatedTime} + 1$
13: **end for**
14: Evolve the KES signing key by sending $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, 0, \texttt{localTime})$ to $\mathcal{F}_{\mathsf{KES}}$
15: Set $t_{\mathrm{work}} \leftarrow \mathsf{localTime}$
16: Set $\texttt{localTime} \leftarrow \mathsf{simulatedTime}$

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

C.14 The round finish procedure

Once a party is done its actions in a round it has to advance the synchronous computation by sending the indication to $\mathcal{G}_{\text{IMPERFLCLOCK}}$. Since the functionality is shared, an update request in the ideal world will be relayed which implies that the protocol cannot simply ignore this input in the real world either. However, the update-request by the environment might not be well aligned with the round actions, so the protocol merely remembers that such an update has been received. At the end of its functions it then executes FinishRound which enforces that the protocol only sends the clock-update once (1) the round operations are concluded and (2) the environment has given the command to advance the round. (Note that in the ideal world, it is the ledger functionality which is registered with $\mathcal{G}_{\text{IMPERFLCLOCK}}$ and enforces the same principal time-evolving behavior as in the real world.)

Protocol FinishRound(P)

while A $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ has not been received during the current round **do**
Give up activation (set the anchor here)
end while
2: Send $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{IMPERFLCLOCK}}$. // Party will lose its activation here.

C.15 The Joining Procedure

One of the main procedures of Ouroboros Chronos concludes this description. The joining procedure will make any party that joins the system getting synchronized with the blockchain and to derive a local time-stamp that is in a small interval around the current alert parties time-stamps. The parameters of the procedure, along with their default values, are summarized in Table 1.

Protocol $\mathsf{JoinProc}(\mathsf{P}, \mathsf{sid}, R, k, f, s, t_{\text{off}}, t_{\text{stable}}, t_{\text{minSync}})$

Call $\mathsf{UpdateTime}(\mathsf{P}, R, f)$ // Align with newest round
if $\texttt{localTime} > 1$ **then** // Set back to local round 1
  Set $\texttt{localTime} \leftarrow 1$
  Set $\texttt{ep} \leftarrow \lceil \texttt{localTime}/R \rceil$, and $\texttt{sl} \leftarrow \texttt{localTime}$.
  $\texttt{fetchCompleted} \leftarrow \mathsf{false}$, $\texttt{futureChains}, \texttt{buffer} \leftarrow \emptyset$, $\mathsf{Timestamp}_{SB} \leftarrow$ empty array.
end if
// Phase B
while $\texttt{localTime} \leq t_{\text{off}}$ **do**
  if $\texttt{fetchCompleted} = \mathsf{false}$ **then**
    Call $\mathsf{FetchInformation}(k, \mathsf{P})$ and denote fetched chains by $\mathcal{N} := (\mathcal{C}_1, \ldots, \mathcal{C}_M)$
    Call $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$ // Since $\texttt{isSync} = \mathsf{false}$, all chains are considered
    $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
    $\mathsf{FinishRound}(\mathsf{P})$ // Mark round actions as finished. Resume below upon next activation
  end if
  Call $\mathsf{UpdateTime}(\mathsf{P}, R, f)$ to update $\texttt{localTime}, \texttt{ep}$, and $\texttt{sl}$ // $\texttt{fetchCompleted}$ will reset.
end while
// Phases C
while $\texttt{localTime} \leq t_{\text{off}} + t_{\text{minSync}} + t_{\text{stable}}$ **do**
  if $\texttt{fetchCompleted} = \mathsf{false}$ **then**
    Call $\mathsf{FetchInformation}(k, \mathsf{P})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\texttt{tx}_1, \ldots, \texttt{tx}_k)$.
    Set $\texttt{buffer} \leftarrow \texttt{buffer}||(\texttt{tx}_1, \ldots, \texttt{tx}_k)$ and define $\texttt{futureChains} \leftarrow \texttt{futureChains}||(\mathcal{C}_1, \ldots, \mathcal{C}_M)$
    Call $\mathsf{ProcessBeacons}$ to collect new beacons in this round.// All arrival times are temporary
    Call $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \texttt{futureChains}, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
    $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
    $\mathsf{FinishRound}(\mathsf{P})$ // Mark round actions as finished. Resume below upon next activation
  end if
  Call $\mathsf{UpdateTime}(\mathsf{P}, R, f)$ to update $\texttt{localTime}, \texttt{ep}$, and $\texttt{sl}$ // $\texttt{fetchCompleted}$ will reset.
end while
// Phase D
Define the function $I_{\text{sync}}(j) : j \mapsto I_j := [(j-1)R + 1, \ldots, (j-1)R + 2R/3]$.
$\texttt{syncBuffer}_{\text{valid}} \leftarrow \{\texttt{SB}' \in \texttt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathsf{sid}, \texttt{SB}', \mathcal{C}_{\mathsf{loc}}, f, R) = \mathsf{true}\}$
Initialize $i := 0$. Now set i to be the minimum positive integer such that $\forall \texttt{SB} \in \mathcal{C}_{\mathsf{loc}}[I_{\text{sync}}(i)] : \texttt{SB} \in \texttt{syncBuffer}_{\text{valid}} \wedge \mathsf{Timestamp}(\texttt{SB}) > t_{\text{off}} + t_{\text{pre}}$ (if no interval exists, i is unchanged).
if $i \geq 1$ **then**
  for at most $((t_{\text{stable}} + t_{\text{minSync}}) \operatorname{div} R))$ iterations **do**
    $\mathcal{S}_i \leftarrow \{\texttt{SB} \mid \exists B \in \mathcal{C}_{\mathsf{loc}}[I_{\text{sync}}(i)] : \texttt{SB} \in B \wedge \mathsf{slotnum}(\texttt{SB}) \in \{(i-1)R + 1, \ldots, (i-1)R + R/6\}\}$
    for each $\texttt{SB} = (\texttt{sl}, \mathsf{P}, y, \pi) \in \mathcal{S}_i$ **do**
      $Q_{\texttt{SB}} \leftarrow \{\texttt{SB}' = (\texttt{sl}', \mathsf{P}', \cdot, \cdot) \in \texttt{syncBuffer}_{\text{valid}} \mid \mathsf{P}' = \mathsf{P} \wedge \texttt{sl}' = \texttt{sl}\}$
      if $Q_{\texttt{SB}} \neq \emptyset$ **then**
        $\min_{\texttt{SB}} \leftarrow \min\{\mathsf{Timestamp}(\texttt{SB}') \mid \texttt{SB}' \in Q_{\texttt{SB}}\}$
        $\mathsf{Timestamp}_{SB}(\texttt{SB}) \leftarrow (\min_{\texttt{SB}}, \mathsf{final})$
        $\mathsf{recom}(\texttt{SB}) \leftarrow \mathsf{slotnum}(\texttt{SB}) - \mathsf{Timestamp}(\texttt{SB})$
      else
        $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\texttt{SB}\}$ // Negligible probability event in execution.
      end if
    end for
    $\mathsf{shift}_i \leftarrow \mathsf{med}\{\mathsf{recom}(\texttt{SB}) \mid \texttt{SB} \in \mathcal{S}_i\}$
    for each $\texttt{SB}$ with $\mathsf{Timestamp}_{SB}(\texttt{SB}) = (a, \mathsf{temp})$ **do**
      $\mathsf{Timestamp}_{SB}(\texttt{SB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{temp})$
    end for
    Set $\texttt{localTime} \leftarrow \texttt{localTime} + \mathsf{shift}_i$; $\mathsf{EpochUpdate}(i) \leftarrow \mathsf{Done}$
    Break if $\texttt{localTime} \leq (i+1)R$. Otherwise, set $i \leftarrow i + 1$ and continue iteration.
  end for
  $\texttt{isSync} \leftarrow \mathsf{true}$; Run $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \texttt{futureChains}, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$; $t_{\text{work}} \leftarrow \texttt{localTime} - 1$
  If $\texttt{localTime} \leq i \cdot R$ then set $t_{\text{work}} \leftarrow i \cdot R$. // Wait if shifted back before sync slot of i.
  Evolve the KES signing key by sending $(\mathsf{USign}, \mathsf{sid}, \mathsf{P}, 0, t_{\text{work}})$ to $\mathcal{F}_{\mathsf{KES}}$
  for each beacon $\texttt{SB} \in \texttt{syncBuffer}_{\text{valid}}$ with $\mathsf{slotnum}(\texttt{SB}) \leq (i+1)R$ **do**
    Parse $\mathsf{Timestamp}_{SB}(\texttt{SB})$ as (a, temp). Define $\mathsf{Timestamp}_{SB}(\texttt{SB}) \leftarrow (a, \mathsf{final})$
  end for
end if

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

Fig. 2. The joining procedure.

Parameter	Default	Phase
t_{off}	$R/3$	B
t_{minSync}	$2R$	C
t_{stable}	R	C
t_{pre}	$3R/4$	D

Table 1. Parameters of the joining procedure and phases in which they play a role.

D The Extended Ledger Functionality

As in [4], we prove composable security of this proof-of-stake protocol by showing that it realizes a ledger functionality that additionally exports additional time-stamps. This is important to show on what guarantees a higher-level protocol can rely as it requires to abstract the time advancement of the protocol in a way that is less complex than the real world. Compared to [4], we have thus two major differences:

1. The ledger uses $\mathcal{G}_{\text{IMPERFLCLOCK}}$ to maintain its baseline reference time (time since creation of the functionality).
2. We need to incorporate the detailed achieved guarantees for the exported time-stamps and provide a useful behavior for alert parties to be used by external protocols.

Overview. The ledger functionality introduced in [4] builds upon the general functionality [5]. In a nutshell, it maintains a central and unique ledger state denoted by `state`. How fast this state grows and which transactions it includes are part of the ledger policy—and therefore fully adjustable—as this depends heavily on the protocol achieving it. In any case, each registered party can request to see the state, and is guaranteed to receive a sufficiently long prefix of it; the size of each party's view of the state is captured by (monotonically) increasing pointers that define which part of the state each party can read; the adversary has a limited control on these pointers. The dynamics of this can be reflected with a sliding window over the sequence of state blocks, with width `windowSize` and starting at the head of the state; each party's pointer points to a location withing this window. The adversary can choose the position of the pointers within this sliding window. As is common in UC, parties advance the ledger when they are instructed to (activated with specific maintain-ledger input by their environment $\mathcal{Z}$). The ledger uses these queries along with the function predict-time$(\cdot)$ to ensure that the ideal world execution advances with the same pace (relatively to the pacemaker)—in this work the *baseline speed captured by the nominal time derived from* $\mathcal{G}_{\text{IMPERFLCLOCK}}$—as the protocol.

Any party can input a transaction to the ledger; upon reception transactions are validated using a predicate Validate and, if found valid, are added to a buffer. Each new block of the state consists of transactions which were previously accepted to the buffer. To give protocols syntactic freedom to structure their state blocks, a vector of transactions, say $\boldsymbol{N}_i$, is mapped to the ith state block via function Blockify($\boldsymbol{N}_i$). Validate and Blockify are two of the ledger's parametrization algorithms. The third algorithm is the predicate predict-time that is instantiated by the predictable time-behavior predicate of the protocol under consideration (in this case Ouroboros Chronos). We note that that Ouroboros Chronos has a predictable time-advancement pattern due to the way the UC protocol is designed and it is always clear when honest parties call FinishRound in the protocol.

Party sets maintained by the ledger. In accordance with the classification of parties in Section 4, the ledger divides the registered honest into two different basic categories called *synchronized* and *desynchronized*. Synchronized parties are the parties that enjoy full security guarantees. Formally, a party that is considered synchronized by the ledger and which is connected to all shared setups is what we usually called *alert* in the dynamic availability setting. A party is considered synchronized if it has been continuously connected to all its resources for a sufficiently long interval and has maintained connectivity to these resources until the current time. Formally, here, "sufficiently long" refers to `Delay`-many rounds, where `Delay` is a parameter of the ledger and directly relates to the real-world time duration between joining and becoming synchronized. In the general formulation, de-synchronized parties receive significantly weaker guarantees.

State-extend policy. A defining part of the behavior of the ledger is the (parameterizable) procedure which defines when/how to extend `state` and what the constraints are for an adversary. The state-extend policy of the ledger in this work is almost identical to the ledger in [4] except that we establish a slightly better bound for transaction liveness guarantees. In nutshell, the basic mode of operation of ExtendPolicy is that it takes as an input a proposal from the adversary for extending the state, and can decide to follow this proposal if it satisfies its policy; if it does not, ExtendPolicy can ignore the proposal (and enforce a default extension). It will enforce minimal chain growth, a certain fraction of "good blocks," and transaction liveness guarantees for old and (still) valid transactions.

Given this similarity, we give a concise summary of the ledger parameters used in this work in Table M.2. The ledger functionality and its ExtendPolicy are further provided in the appendix for the sake of self-containment.

The Export-Time Extension. We introduce the export time extension to $\mathcal{G}_{\text{LEDGER}}$. first, we represent a time-stamps $\texttt{time}_{\mathsf{P}}$ associated to party P as a pair (e, t), where t is the actual time stamp, and e refers to what we call a epoch.[20] An alert party's time t in (e, t) is guaranteed to increase during an epoch with every tick of the reference speed. Once t hits an epoch boundary, defined as multiples of an epoch length parameter R_L[21], the epoch value increases as well. Clearly, this would a perfect, monotonically increasing, two-dimensional time-stamp. We have to weaken this guarantee by allowing to the adversary to apply a limited shift whenever a party is at an epoch boundary (parameters shiftLB, shiftUB). Furthermore the ledger enforces that any two alert parties with respective time-stamps (e, t) and (e', t'), satisfy the constraints $|t - t'| \leq \texttt{timeSlack}_{\text{total}}$ and $|t - t'| \leq \texttt{timeSlack}_{\text{ep}}$ if $e = e'$, and $|e - e'| \leq 1$ for the respective ledger parameters $\texttt{timeSlack}_{\text{ep}}$, $\texttt{timeSlack}_{\text{total}}$ that define the maximally allowed skewness of parties. Note that we give the possibility than Within an epoch the slack could be potentially different (i.e., much better) than across epochs.

We give an overview of the parameters in Table M.2 and provide the functionality in Section D.1.

D.1 The Functionality

For completeness, we describe here the ledger functionality in the $\mathcal{G}_{\text{IMPERFLCLOCK}}$ world together with the export-time extension. A couple of technicalities might be mentioned here regarding the novel aspects for time-management: as explained in the main body, the timestamps are two-dimensional of the form (e, t), where e is an epoch number and t is the actual timestamp. Initially, the epoch is $e = -1$ which stands for "prior" to be active. Timestamps advance in a monotone fashion controlled by the functionality. The only irregularities are introduced, by the adversary, once the epoch increases from e to $e + 1$ for a party. Then the functionality allows the adversary shifting the reported timestamp of that party by a limited amount. The influence is captured by the above explained clock parameters. The first switch the adversary can do (when $e = -1$ and $t = 0$) corresponds to the initial offset. We highlight the differences to the PoS ledger from [4] in blue to make the quite complex definition more modular.

Functionality $\mathcal{G}_{\text{LEDGER}}$ with Export-Clock Extension

General: The functionality is parameterized by four algorithms, Validate, ExtendPolicy, Blockify, and predict-time, along with three parameters: $\texttt{windowSize}, \texttt{Delay} \in \mathbb{N}$, and $\mathcal{S}_{\text{initStake}} := \{(U_1, s_1), \ldots, (U_n, s_n)\}$. The functionality manages variables `state`, `NxtBC`, `buffer`, τ_L, and $\boldsymbol{\tau}_{\texttt{state}}$, as described above. The variables are initialized as follows: $\texttt{state} := \boldsymbol{\tau}_{\texttt{state}} := \texttt{NxtBC} := \varepsilon$, $\texttt{buffer} := \emptyset$, $\tau_L = 0$. For each party $\mathsf{P} \in \mathcal{P}$ the functionality maintains a pointer $\texttt{pt}_i$ (initially set to 1) and a current-state view $\texttt{state}_p := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\boldsymbol{I}_H^T$ (initially $\boldsymbol{I}_H^T := \varepsilon$).

Party Management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (as discussed below). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are

[20] In the real world, e would be the number of adjustments a party has made to its local clock.
[21] In this work, this is always exactly the length of normal epoch, i.e., $R_L = R$.

all initially set to $\emptyset$. When a (currently unregistered) honest party is registered at the ledger, *if it is registered with the clock-tick functionality and the global RO already,* then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to $\mathcal{G}_{\text{IMPERFLCLOCK}}$ in round $\tau_L = 0$ or whenever $\mathcal{H} \neq \emptyset$ in which case it updates the clock at the speed of the slowest party in $\mathcal{H}$.

Time management: When activated with the first registration command, the ledger prepends to its actions above the following steps: it asks for the clock tick to $\mathcal{G}_{\text{IMPERFLCLOCK}}$ and stores the variable internally as `lastTick` and sets $\tau_L := 0$.
It further activates the adversary with restricting query $(\mathsf{Respond}, (\mathsf{start}, sid))$ and resumes with normal registration after receiving the immediate acknowledgment from the adversary.

Handling initial stakeholders: The ledger blocks the advancement of the baseline time of the clock until each initial stakeholder, i.e., $\mathsf{P} \in \mathcal{S}_{\text{initStake}}$ has registered. Additionally, the adversary can assign a unique reference string ref_p once to each registered party P.

Extension Parameters: The extension is parameterized by $\mathsf{shiftLB}$, $\mathsf{shiftUB}$, $\texttt{timeSlack}_{\text{ep}}$, $\texttt{timeSlack}_{\text{total}}$, and the epoch length R_L.

Extension Variables: The extension introduces the new variables $\texttt{time}_{\mathsf{P}}$ for each registered party $\mathsf{P} \in \mathcal{P}$ (initial value $(-1, 0)$).

Upon receiving any input I from any party or from the adversary, send $(\text{CLOCK-GET}, \text{sid}_C)$ to $\mathcal{G}_{\text{IMPERFLCLOCK}}$ and upon receiving response $(\text{CLOCK-GET}, \text{sid}_C, \text{tick})$ set $d = 1$ if $\texttt{lastTick} \neq \text{tick}$ and 0 otherwise. Set $\tau_L := \tau_L + d$. Do the following if $\tau_L > 0$ (otherwise, ignore input):

1. Updating synchronized/desynchronized party set:
   (a) Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously) to the ledger,$\mathcal{G}_{\text{IMPERFLCLOCK}}$, and the GRO since time $\tau' < \tau_L - \texttt{Delay}$.
   (b) For each party $\mathsf{P} \in \widehat{\mathcal{P}}$ with $\texttt{time}_{\mathsf{P}} = (e, t)$, ensure valid range of timestamps: verify that $t \leq \tau_L + \Delta_{\text{clock}}$ and that for any party $\mathsf{P}' \in \mathcal{H} \setminus \mathcal{P}_{DS}$ with $\texttt{time}_{\mathsf{P}'} = (e', t')$, it holds that $|t - t'| \leq \texttt{timeSlack}_{\text{total}}$, if $e = e'$ it also holds that $|t - t'| \leq \texttt{timeSlack}_{\text{ep}}$ and that $|e - e'| \leq 1$. If a comparison fails, set $\texttt{time}_{\mathsf{P}}$ to that value $\texttt{time}_{\mathsf{P}'}$.
   (c) Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.
   (d) For any synchronized party $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if P is not registered to the clock, then consider it desynchronized, i.e., set $\mathcal{P}_{DS} \cup \{\mathsf{P}\}$.
2. If I was received from an honest party $\mathsf{P} \in \mathcal{P}$:
   (a) Set $\boldsymbol{\mathcal{I}}_H^T := \boldsymbol{\mathcal{I}}_H^T || (I, \mathsf{P}, \tau_L)$;
   (b) Compute $\boldsymbol{N} = (\boldsymbol{N}_1, \ldots, \boldsymbol{N}_\ell) := \mathsf{ExtendPolicy}(\boldsymbol{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \boldsymbol{\tau}_{\texttt{state}})$ and if $\boldsymbol{N} \neq \varepsilon$ set $\texttt{state} := \texttt{state} || \mathsf{Blockify}(\boldsymbol{N}_1) || \ldots || \mathsf{Blockify}(\boldsymbol{N}_\ell)$ and $\boldsymbol{\tau}_{\texttt{state}} := \boldsymbol{\tau}_{\texttt{state}} || \tau_L^\ell$, where $\tau_L^\ell = \tau_L || \ldots, || \tau_L$.
   (c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$ then delete $\texttt{BTX}$ from $\texttt{buffer}$. Also, reset $\texttt{NxtBC} := \varepsilon$.
   (d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ or $\texttt{pt}_j < |\texttt{state}_j|$, then set $\texttt{pt}_k := |\texttt{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.
3. Increase the party time stamps in a new round:
   (a) For all $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$ do: parse $\texttt{time}_{\mathsf{P}}$ as (e, t) and set $\texttt{time}_{\mathsf{P}} \leftarrow (e, t + 1)$ for all the parties that have advanced locally according to $\mathcal{G}_{\text{IMPERFLCLOCK}}$ since the last activation. If P is stalled and $t + 1 \operatorname{div} R_L > e$ then $\mathcal{P}_{DS} \cup \{\mathsf{P}\}$.
4. If the calling party P is *stalled or time-unaware* (according to the defined party classification), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\text{LEDGER}}$ executes the corresponding code from the following list:

- *Submitting a transaction:*
  If $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ and is received from a party $\mathsf{P} \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party P) do the following
  (a) Choose a unique transaction ID txid and set $\texttt{BTX} := (\texttt{tx}, \text{txid}, \tau_L, \mathsf{P})$
  (b) If $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 1$, then $\texttt{buffer} := \texttt{buffer} \cup \{\texttt{BTX}\}$.
  (c) Send $(\text{SUBMIT}, \texttt{BTX})$ to $\mathcal{A}$.

- *Reading the state:*
  If $I = (\text{READ}, \text{sid})$ is received from a party $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$ then immediately return $(\text{READ}, \text{sid}, \varepsilon)$ if $\texttt{time}_{\mathsf{P}} = (e, t)$ with $e \leq 0 \wedge t < 0$. Else, set $\texttt{state}_p := \texttt{state}|_{\min\{\texttt{pt}_p, |\texttt{state}|\}}$ and return $(\text{READ}, \text{sid}, \texttt{state}_p)$ to the requester. If the requester is $\mathcal{A}$ then send $(\texttt{state}, \texttt{buffer}, \boldsymbol{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

- *Maintaining the ledger state:*
  If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $\mathsf{P} \in \mathcal{P}$ and (after updating $\boldsymbol{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\boldsymbol{\mathcal{I}}_H^T, \text{aux}) = \widehat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{IMPERFLCLOCK}}$. Here we provide $\text{aux} = (\texttt{state}, (\mathsf{P}_1, \text{ref}_{\mathsf{P}_1}, \texttt{time}_{\mathsf{P}_1}), \ldots, (\mathsf{P}_{|\mathcal{P}|}, \text{ref}_{\mathsf{P}_{|\mathcal{P}|}}, \texttt{time}_{\mathsf{P}_{|\mathcal{P}|}}))$. Else, send I to $\mathcal{A}$.
  Before losing the activation, if party $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$ has only locally completed its round actions, update the clock in the name of this party. If $\texttt{time}_{\mathsf{P}} = (e, t)$ for $t > 0$ and $t \operatorname{div} R_L > e$ then set $\texttt{time}_{\mathsf{P}} \leftarrow (e + 1, t)$.

- *The adversary proposing the next block:*
  If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \ldots, \text{txid}_\ell))$ is sent from the adversary, update $\texttt{NxtBC}$ as follows:
  (a) Set listOfTxid $\leftarrow \epsilon$
  (b) For $i = 1, \ldots, \ell$ do: if there exists $\texttt{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, U_j) \in \texttt{buffer}$ with ID txid $= \text{txid}_i$ then set listOfTxid := listOfTxid$||\text{txid}_i$.
  (c) Finally, set $\texttt{NxtBC} := \texttt{NxtBC}||(\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.

- *The adversary setting state-slackness:*
  If $I = (\text{SET-SLACK}, (U_{i_1}, \widehat{\texttt{pt}}_{i_1}), \ldots, (U_{i_\ell}, \widehat{\texttt{pt}}_{i_\ell}))$, with $\{\mathsf{P}_{i_1}, \ldots, \mathsf{P}_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:
  (a) If for all $j \in [\ell] : |\texttt{state}| - \widehat{\texttt{pt}}_{i_j} \leq \texttt{windowSize}$ and $\widehat{\texttt{pt}}_{i_j} \geq |\texttt{state}_{i_j}|$, set $\texttt{pt}_{i_1} := \widehat{\texttt{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$.
  (b) Otherwise set $\texttt{pt}_{i_j} := |\texttt{state}|$ for all $j \in [\ell]$.

- *The adversary setting the state for desychronized parties:*
  If $I = (\text{DESYNC-STATE}, (U_{i_1}, \texttt{state}'_{i_1}), \ldots, (U_{i_\ell}, \texttt{state}'_{i_\ell}))$, with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\texttt{state}_{i_j} := \texttt{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, ok)$ to $\mathcal{A}$.

- *Reading the time:*
  If $I = (\text{EXPORT-TIME}, \text{sid})$ is received from a party $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$ then parse $\texttt{time}_{\mathsf{P}}$ as (e, t). If $e \geq 0$ and $t \geq 0$ then return $(\text{EXPORT-TIME}, \text{sid}, \texttt{time}_{\mathsf{P}})$ to the requester. Otherwise, return $(\text{EXPORT-TIME}, \text{sid}, \perp)$.

- *The adversary setting the shift on epoch boundaries :*
  If $I = (\text{APPLY-SHIFT}, \text{sid}, (\mathsf{P}, s))$ is received from the adversary $\mathcal{A}$ and $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$ then do the following:
  (a) Parse $\texttt{time}_{\mathsf{P}}$ as (e, t). If $t \bmod R_L \neq 0$ return to the adversary.
  (b) Verify that $\mathsf{shiftLB} \leq s \leq \mathsf{shiftUB}$. If the check fails, return to the adversary.
  (c) Set $t' \leftarrow t + s$ and verify correct range of timestamps:
  Check that $t' \leq \tau_L$ and that for each party $\mathsf{P}' \in \mathcal{H} \setminus \mathcal{P}_{DS}$ with $\texttt{time}_{\texttt{time}'} = (e'', t'')$, it holds that $|t' - t''| \leq \texttt{timeSlack}_{\text{total}}$, $|e' - e''| \leq 1$, and if $e' = e'$ verify that $|t' - t''| \leq \texttt{timeSlack}_{\text{ep}}$.
  (d) If all checks succeeds, set $\texttt{time}_{\mathsf{P}} \leftarrow (e, t')$. Otherwise, set $\texttt{time}_{\mathsf{P}} \leftarrow \texttt{time}_{\mathsf{P}'}$ where P' is the lexicographically smallest identity of $\mathcal{H} \setminus \mathcal{P}_{DS}$. Return activation to the adversary.

- *The adversary setting the timestamp for desynchronized parties:*
  If $I = (\text{SET-TIME}, \text{sid}, \mathsf{P}, \texttt{time})$ is received from the adversary $\mathcal{A}$ do the following: if $\mathsf{P} \in \mathcal{P}_{DS}$ then set $\texttt{time}_{\mathsf{P}} \leftarrow \texttt{time}$

D.2 Extend Policy

For completeness, we state here the extend policy of [4] and mark in blue the minor modification. Note that the default block mechanism DefaultExtension is identical to [4] and thus omitted.

Algorithm ExtendPolicy for $\mathcal{G}_{\text{LEDGER}}$

function ExtendPolicy($\mathcal{I}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \boldsymbol{\tau}_{\mathtt{state}}$)
  // First, create a default honest client block as alternative:
  $\boldsymbol{N}_{\mathtt{df}} \leftarrow$ DefaultExtension($\mathcal{I}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \boldsymbol{\tau}_{\mathtt{state}}$) // Extension if adversary violates policy.
  Let τ_L be current ledger time (computed from $\mathcal{I}_H^T$)
  // The function must not have side-effects: Only modify copies of relevant values.
  Create local copies of the values $\mathtt{buffer}$, $\mathtt{state}$, and $\boldsymbol{\tau}_{\mathtt{state}}$.
  // Now, parse the proposed block by the adversary
  Parse $\mathtt{NxtBC}$ as a vector $((\text{hFlag}_1, \mathtt{NxtBC}_1), \cdots, (\text{hFlag}_n, \mathtt{NxtBC}_n))$
  $\boldsymbol{N} \leftarrow \varepsilon$ // Initialize Result
  // Determine the time of the state block which is $\mathtt{windowSize}$ blocks behind the head of the state
  if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then**
    Set $\tau_{\mathtt{low}} \leftarrow \boldsymbol{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
  else
    Set $\tau_{\mathtt{low}} \leftarrow 1$
  end if
  oldValidTxMissing $\leftarrow$ false // Flag to keep track whether old enough, valid transactions are inserted.
  for each list $\mathtt{NxtBC}_i$ of transaction IDs **do**
    // Compute the next state block
    // Verify validity of $\mathtt{NxtBC}_i$ and compute content
    Use the txid contained in $\mathtt{NxtBC}_i$ to determine the list of transactions
    Let $\mathtt{tx} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_{|\mathtt{NxtBC}_i|})$ denote the transactions of $\mathtt{NxtBC}_i$
    if $\mathtt{tx}_1$ is not a coin-base transaction **then**
      return $\boldsymbol{N}_{\mathtt{df}}$
    else
      $\boldsymbol{N}_i \leftarrow \mathtt{tx}_1$
      for $j = 2$ to $|\mathtt{NxtBC}_i|$ **do**
        Set $\mathtt{st}_i \leftarrow \mathsf{blockify}_{\mathbb{B}}(\boldsymbol{N}_i)$
        if $\mathsf{ValidTx}_{\mathbb{B}}(\mathtt{tx}_j, \mathtt{state}||\mathtt{st}_i) = 0$ **then**
          return $\boldsymbol{N}_{\mathtt{df}}$
        end if
        $\boldsymbol{N}_i \leftarrow \boldsymbol{N}_i||\mathtt{tx}_j$
      end for
      Set $\mathtt{st}_i \leftarrow \mathsf{blockify}_{\mathbb{B}}(\boldsymbol{N}_i)$
    end if
    // Test that all old valid transaction are included
    if the proposal is declared to be an honest block, i.e., $\text{hFlag}_i = 1$ **then**
      for each $\mathtt{BTX} = (\mathtt{tx}, \text{txid}, \tau', \mathsf{P}) \in \mathtt{buffer}$ of an honest party P with time $\tau' < \tau_{\mathtt{low}} - \mathtt{Delay}_{\mathtt{tx}}$ **do**
        if $\mathsf{ValidTx}_{\mathbb{B}}(\mathtt{tx}, \mathtt{state}||\mathtt{st}_i) = 1$ but $\mathtt{tx} \notin \boldsymbol{N}_i$ **then**
          oldValidTxMissing $\leftarrow$ true
        end if
      end for
    end if
    $\boldsymbol{N} \leftarrow \boldsymbol{N}||\boldsymbol{N}_i$
    $\mathtt{state} \leftarrow \mathtt{state}||\mathtt{st}_i$
    $\boldsymbol{\tau}_{\mathtt{state}} \leftarrow \boldsymbol{\tau}_{\mathtt{state}}||\tau_L$
    // Must not proceed with too many adversarial blocks
    $i \leftarrow \max\{\{\mathtt{windowSize}\} \cup \{k \mid \mathtt{st}_k \in \mathtt{state} \wedge \text{proposal of } \mathtt{st}_k \text{ had hFlag} = 1\}\}$ // Determine most
    // recent honestly-generated block in the interval behind the head.

```
        if |state| − i ≥ advBlcks_window then
            return N_df
        end if
        // Update τ_low: the time of the state block which is windowSize blocks behind the head of the
        // current, potentially already extended state
        if |state| ≥ windowSize then
            Set τ_low ← τ_state[|state| − windowSize + 1]
        else
            Set τ_low ← 1
        end if
    end for
    // Final checks (if policy is violated, it is enforced by the ledger):
    // Must not proceed too slow or with missing transaction.
    if τ_low > 0 and τ_L − τ_low > maxTime_window then // A sequence of blocks cannot take too much time.
        return N_df
    else if τ_low = 0 and τ_L − τ_low > 2 · maxTime_window then // Bootstrapping cannot take too much time.
        return N_df
    else if oldValidTxMissing then // If not all old enough, valid transactions have been included.
        return N_df
    end if
    return N
end function
```

E On Using the Exported Clock

The goal of this section is to give an overview on how external protocols could make use of the timing service provided by Ouroboros Chronos. For simplicity, this section assume that all parties run at equal speed, i.e., $\Delta_{\text{clock}} = 0$. In this model, the nominal time has a simple meaning: it is the objective number of rounds passed in the execution.

E.1 General Considerations

Cryptographic protocols can use the exported clock of Chronos and make use of the provided timestamps. It is instructive to see different cases depending on the parameters of the clock. For example, if $\texttt{timeSlack}_{\text{ep}} = \texttt{timeSlack}_{\text{total}} = 0$, and $\mathsf{shiftLB} = \mathsf{shiftUB} = 0$, then we have an equivalent formulation of the global clock of previous works. Each weakening of the parameters will result in a higher-level protocol to require specific reactions. This is, however, possible: for example, if shifts are to be expected but still $\texttt{timeSlack}_{\text{ep}} = \texttt{timeSlack}_{\text{total}} = 0$, then the parties will know that some strange behavior could happen around the epoch boundaries. However, the behavior is limited and predictable based on the clock parameters. For example, parties could stall their operations just before the epoch boundary switches and depending on the shift, resume their operations later at a specific time. Furthermore, by the limited shift and the guaranteed advancement the parties will proceed and, if the protocol uses explicit knowledge of $\mathsf{shiftLB}$ and $\mathsf{shiftUB}$, liveness can be explicitly quantified. If parties can further be skewed, in addition to the above, the higher level protocol has to be resilient against small variations in the time-stamps. Again, the level of resilience required is clearly defined by parameters $\texttt{timeSlack}_{\text{total}}$ and $\texttt{timeSlack}_{\text{ep}}$.

E.2 Clock Properties in Optimistic Network Models

In the above cryptographic treatment, we made worst-case assumptions regarding the delivery times of the synchronization beacons and chains. One can ask the question how well Chronos adjusts to more optimistic network models, i.e., how the clock parameters get more precise if Chronos is executed in a less adversarial environment. We can infer some rough estimates based on our analysis by formally imposing restrictions on the delays occurring in the network and then reworking the derivation of the main clock properties along the

lines of Lemma 9. We exemplify this with two examples. Let us assume that we have initial coordination, i.e., that all parties obtain the genesis block in the same objective round. Let us further assume that all messages sent in round ℓ are guaranteed to be delivered in round $\ell + 1$—or more generally in round $\ell + m$. Along the lines of Lemma 9, we observe that the timestamps reported by alert parties are very coordinated. In fact, if τ denotes objective time (as recorded inside $\mathcal{G}_{\text{IMPERFLCLOCK}}$), then each alert party reports time $t = \tau - m \cdot e$ (where e is the timestamp's epoch number) and thus the clock parameters of the ledger would in this case be $\mathsf{shiftLB} = \mathsf{shiftUB} = -m$ and $\mathtt{timeSlack}_{\text{ep}} = 0$, $\mathtt{timeSlack}_{\text{total}} = m$. Even more, in this case, any external protocol can recompute the objective time (as recorded by $\mathcal{G}_{\text{IMPERFLCLOCK}}$) from a timestamp (e, t) of an alert party by the adjustment $t + e \cdot m$ and thereby obtain a perfect "global clock" whenever this particular "network parameter" m is known.

More realistically, if we consider delays in an interval of the form $[m-d, m+d] \subseteq [0, \Delta]$ then the arguments of Lemma 9 suggest the clock parameters $\mathsf{shiftLB} = -m - 3d$, $\mathsf{shiftUB} = -m + 3d$ and $\mathtt{timeSlack}_{\text{ep}} = 2d$, $\mathtt{timeSlack}_{\text{total}} = m+3d$. In this case, the above proposed time-adjustment $t+e\cdot m$, based on the characteristic value m of the network, would yield an approximation of the objective τ in the order of the width $2d$ of the interval (note that this naturally matches our security analysis when choosing $m = d$ and $2d = \Delta$, i.e., when m is the center of $[0, \Delta]$).

In the above considerations, the term $2d$ refers to a logical width measured in number of rounds. Depending on the message sizes and the physical time duration of one round, if messages that a party needs to send during round ℓ are always sent at the end of round ℓ, then a logical width of $2d \approx 0$ might not be unrealistic to achieve. Also, an accurate approximation of the "typical delay" parameter m seems in principle feasible. Hence, obtaining quite accurate approximations of objective time seems achievable by Ouroboros Chronos in networks that have a somewhat predictable behavior. This, however, depends heavily on the implementations and the network stack and a more accurate study is part of future work.

F Single-Epoch Security with Static Registration and Δ-Bounded Skew

In this section we prove Theorem 1. First, in Section F.1, we introduce a simplified chain-selection rule that will make our protocol easier to analyze. In Section F.2 we draw the connection between a single-epoch execution of this simplified protocol and the formalism of characteristic strings and forks that we later employ. We then analyze the distribution of the characteristic strings induced by an execution of the simplified Ouroboros Chronos in Section F.3, and establish the implications of that for the properties CP, CG and CQ in Section F.4. Finally, in Section F.5, we replace the simplified chain-selection rule with the actual one, concluding the proof of Theorem 1.

F.1 The Simplified Chain-Selection Rule maxvalid-mc

To make our analysis more modular, and take advantage of the combinatorial framework for analyzing common-prefix violations of longest-chain rule protocols developed gradually in [26,14,4], we first consider the protocol Ouroboros-Chronos with a simplified chain-selection rule maxvalid-mc (given in Fig. 3) instead of the actual rule maxvalid-bg given in Fig. C.8; we will denote this variant $\text{Ouroboros-Chronos}_{\mathsf{mc}}$ for conciseness.

The rule maxvalid-mc differs in that it applies the longest-chain preference and refuses to revert more than k blocks under any circumstances (hence the "mc" identifier standing for "moving checkpoint"). This is in contrast to the more nuanced behavior of maxvalid-bg that compares the two chains forking more than k blocks ago for density close to the point where they fork (cf. Condition B in Fig. C.8). The latter rule allows for so-called *bootstrapping from genesis* [4] (hence "bg") and so we adopt it for Ouroboros Chronos as well, the consequences for our analysis are discussed in Section F.5.

F.2 From Executions to Forks

We recall the notion of a *characteristic string*, which we use to record, for each slot in a sequence of slots, whether any leader is elected for the slot and, if that is the case, whether this leader is unique and alert.

Protocol maxvalid-mc($\mathcal{C}_{\mathsf{loc}}, \mathcal{C}_1, \ldots, \mathcal{C}_\ell$)

Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_{\mathsf{loc}}$.
for $i = 1$ to ℓ **do**
  if IsValidChain($\mathcal{C}_i$) **then**
  // Compare $\mathcal{C}_{\mathsf{max}}$ to $\mathcal{C}_i$
    if ($\mathcal{C}_i$ forks from $\mathcal{C}_{\mathsf{max}}$ at most k blocks) **then**
      if $|\mathcal{C}_i| > |\mathcal{C}_{\mathsf{max}}|$ **then** // Condition A
        Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$.
      end if
    end if
  end if
end for
return $\mathcal{C}_{\mathsf{max}}$.

Fig. 3. The simplified chain selection rule maxvalid-mc.

Definition 6 (Characteristic string [26,14,4]). *Let $S = \{\mathtt{sl}_1, \ldots, \mathtt{sl}_R\}$ be a sequence of slots of length R; consider an execution (with adversary $\mathcal{A}$ and environment $\mathcal{Z}$) of the protocol. For a slot $\mathtt{sl}_j$, let $\mathrm{P}(j)$ denote the set of active parties assigned to be slot leaders for slot j by the protocol. We define the* characteristic string $w \in \{0,1,\perp\}^R$ *of S to be the random variable so that*

$$w_j = \begin{cases} \perp & \text{if } \mathrm{P}(j) = \emptyset, \\ 0 & \text{if } |\mathrm{P}(j)| = 1 \text{ and the assigned party is alert,} \\ 1 & \text{otherwise.} \end{cases} \tag{7}$$

For such a characteristic string $w \in \{0,1,\perp\}^*$ we say that the index j is uniquely alert *if $w_j = 0$,* empty *if $w_j = \perp$, and* potentially active *if $w_j \in \{0,1\}$.*

If the execution is fixed (i.e., the randomness of the execution is fixed), we use the notation $w_{\mathcal{E}}$ to denote the fixed characteristic string resulting from that particular execution, where the subscript $\mathcal{E}$ is used to indicate its difference to the random variable above.

We also recall the notion of a Δ-fork, a tool developed to reason about the various blockchains that can be induced by an adversary in the Δ-synchronous setting with a particular characteristic string.

Definition 7 (Δ-fork [14]). *Let $w \in \{0,1,\perp\}^k$ and Δ be a non-negative integer. Let $A = \{i \,|\, w_i \neq \perp\}$ denote the set of potentially active indices, and let $H = \{i \,|\, w_i = 0\}$ denote the set of uniquely alert indices. A* Δ-fork *for the string w is a rooted tree $F = (V, E)$ with a labeling $\ell : V \to \{0\} \cup A$ so that*

(i) the root $r \in V$ is given the label $\ell(r) = 0$;
(ii) the labels along any (simple) path beginning at the root are strictly increasing;
(iii) each uniquely alert index $i \in H$ is the label of exactly one vertex of F;
(iv) the function $\mathbf{d} : H \to \{1, \ldots, k\}$, defined so that $\mathbf{d}(i)$ is the depth in F of the unique vertex v for which $\ell(v) = i$, satisfies the following Δ-monotonicity *property: if $i, j \in H$ and $i + \Delta < j$, then $\mathbf{d}(i) < \mathbf{d}(j)$.*

For convenience, we direct the edges of forks so that depth increases along each edge; then there is a unique directed path from the root to each vertex and, in light of (ii), labels along such a path are strictly increasing. As a matter of notation, we write $F \vdash_\Delta w$ to indicate that F is a Δ-fork for the string w. We typically refer to a Δ-fork as simply a "fork".

Note that both notions of a characteristic string and a fork can be directly ported to our setting without a global clock, interpreting the slot indices as *logical time*, in accordance with the rest of this paper (cf. Section 2). However, this change of the setting requires us to re-establish the connection between executions and forks from [14]. The relevant part of the outcome of an execution is captured in the notion of an *execution tree* which we first define, the transition from executions to forks is then stated in Lemma 2.

Definition 8 (Execution tree [14]). *Consider an execution $\mathcal{E}$ of the real-world experiment. The* execution tree *for this execution is a directed, rooted tree $T_{\mathcal{E}} = (V, E)$ with a labeling $\ell : V \to \mathbb{N}_0$ that is constructed during the execution as follows:*

(i) At the beginning, $V = \{r\}$, $E = \emptyset$ and $\ell(r) = 0$.
(ii) Every chain C' that is input to maxvalid-bg *as a part of $\mathcal{N}$ or created as a new local chain in Step 20 of* StakingProcedure *of* Ouroboros-Chronos *run by any alert party is immediately processed block-by-block from the genesis block to $\mathtt{head}(C')$. For every block $B = (h, \mathtt{st}, \mathsf{sb}, \mathtt{sl}, crt, \rho, \sigma)$ processed for the first time:*
  – a new vertex v_B is added to V;
  – a new edge (v_{B^-}, v_B) is added to E where B^- is the unique block such that $H(B^-) = h$;
  – the labeling ℓ is extended by setting $\ell(v_B) = \mathtt{sl}$.

Lemma 2. *Consider a single-epoch execution $\mathcal{E}$ of* Ouroboros-Chronos$_{\mathsf{mc}}$ *with static registration and Δ-bounded skew, where Δ is the sum of the maximum (local) clock drift and maximum network delay; let R be the epoch length.*

1. *Every message sent by an alert party P in slot $\mathtt{sl}$ (according to the local time of P) will be received by any other alert party P' by slot $\mathtt{sl}' \triangleq \mathtt{sl} + \widetilde{\Delta}$ for $\widetilde{\Delta} \triangleq 2\Delta$ (according to the local time of P').*
2. *In particular, we have $T_{\mathcal{E}} \vdash_{\widetilde{\Delta}} w_{\mathcal{E}}$ unless a collision in the responses of the random oracle occurs.*

Proof (sketch). For the first statement, note that by the assumption $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$, we know that P will be executing its (logical) slot $\mathtt{sl}$ at most Δ rounds (by the increase of nominal time) later than P' executed $\mathtt{sl}$; and if P sends a message in $\mathtt{sl}$, the party P will receive it at most Δ_{net} activations later after the send event. Note that in the worst case, the send-event happens when the receiving party already has an offset of Δ_{clock} rounds to the sender. Therefore, it will receive it Δ rounds later (again, with respect to the nominal time advancement) be the assumption on network delay. Combining these two bounds, P' will receive the message at latest by slot $\mathtt{sl}'$ according to its own local clock.

As for the second statement, observe that the properties (i)–(iii) in Definition 7, as well as the requirement that $\mathsf{range}(\ell) = \{0\} \cup A$, are satisfied for the same reasons as given in [14, Lemma 6]. The remaining property (iv) is satisfied for $\widetilde{\Delta} \triangleq 2\Delta$ thanks to the first statement of this lemma: Given that an alert party P' is aware of any block produced by P for slot $\mathtt{sl}$, it will act based upon it and if it creates any block for slot $\mathtt{sl}'$, its depth will be strictly larger than the depth of any block created by P for the slot $\mathtt{sl}$ by the description of the protocol. □

To maintain readability, in the following treatment we will omit the (negligible) failure probability caused by random-oracle collisions that are mentioned in the second statement of Lemma 2.

F.3 Protocol-Induced Distribution of the Characteristic String

Badertscher *et al.* [4] identified the following property of a characteristic string distribution to be of particular interest.

Definition 9 (The characteristic conditions [4]). *Consider a family of random variables $W_1, \ldots, W_n$ taking values in $\{0, 1, \perp\}$. We say that they satisfy the* $(f; \gamma)$-characteristic conditions *if, for each $k \geq 1$,*

$$\Pr[W_k = \perp \,|\, W_1, \ldots, W_{k-1}] \geq (1 - f)\,,$$
$$\Pr[W_k = 0 \,|\, W_1, \ldots, W_{k-1}, W_k \neq \perp] \geq \gamma\,, \text{ and hence}$$
$$\Pr[W_k = 1 \,|\, W_1, \ldots, W_{k-1}, W_k \neq \perp] \leq 1 - \gamma\,.$$

In the expressions above, conditioning on a collection of random variables indicates that the statement is true for any conditioning on the values taken by variables.

The distribution of the characteristic string induced by Ouroboros-Chronos$_{\mathsf{mc}}$ satisfies $(f; \gamma)$-characteristic conditions for parameters recorded in the next lemma.

Lemma 3 (Protocol-Induced Distribution). *Consider an execution of the protocol* Ouroboros-Chronos$_{\mathsf{mc}}$ *in the single-epoch setting, with static registration and Δ-bounded skew. Let R denote the epoch length and f be the active-slot coefficient. Let α (resp., β) be a lower bound on the alert stake ratio (resp., participating stake ratio) over the execution. This execution then induces a characteristic string $W_1, \ldots, W_R$ (with each $W_t \in \{0, 1, \perp\}$) satisfying the $(f; (1-f)^2\alpha)$-characteristic conditions, and moreover $\Pr[W_t = \perp \mid W_1, \ldots, W_{t-1}] \leq 1 - f\beta$.*

Proof (sketch). The lemma can be established by following the same reasoning as in [4, Corollary 2] with respect to logical slots rather than nominal-time rounds. □

F.4 Single-Epoch Security Properties

Any characteristic string that satisfies particular $(f; \gamma)$-characteristic conditions does not allow for too large common prefix violations, as proven in [4, Theorem 6] and formally captured by the notion of divergence. We record this result below as Theorem 4, after presenting the necessary formalism in Definitions 10 and 11.

Definition 10 (Tines, length, and viability [4]). *A path in a fork F originating at the root is called a* tine. *For a tine t we let* $\mathsf{length}(t)$ *denote its* length, *equal to the number of edges on the path. For a vertex v, we call the length of the tine terminating at v the* depth of v. *For convenience, we overload the notation $\ell(\cdot)$ so that it applies to tines by defining $\ell(t) \triangleq \ell(v)$, where v is the terminal vertex on the tine t. We say that a tine t is* Δ-viable *if $\mathsf{length}(t) \geq \max_{h+\Delta \leq \ell(t)} \mathbf{d}(h)$, this maximum extended over all uniquely alert indices h (appearing Δ or more slots before $\ell(t)$). Note that any tine terminating in a uniquely alert vertex is necessarily viable by the Δ-monotonicity property.*

Definition 11 (Divergence [26,14]). *Let F be a Δ-fork for a string $w \in \{0, 1, \perp\}^*$. For two Δ-viable tines t and t' of F, we define the notation t/t' by the rule*

$$t/t' = \mathsf{length}(t) - \mathsf{length}(t \cap t')\,,$$

where $t \cap t'$ denotes the common prefix of t and t'. Then define the divergence *of two viable tines t_1 and t_2 to be the quantity*

$$\operatorname{div}(t_1, t_2) = \begin{cases} t_1/t_2 & \text{if } \ell(t_1) < \ell(t_2), \\ t_2/t_1 & \text{if } \ell(t_2) < \ell(t_1), \\ \max(t_1/t_2, t_2/t_1) & \text{if } \ell(t_1) = \ell(t_2). \end{cases}$$

We extend this notation to the fork F by maximizing over viable tines: $\operatorname{div}_\Delta(F) \triangleq \max_{t_1, t_2} \operatorname{div}(t_1, t_2)$, taken over all pairs of Δ-viable tines of F. Finally, we define the Δ-divergence of a characteristic string w to be the maximum over all Δ-forks: $\operatorname{div}_\Delta(w) \triangleq \max_{F \vdash_\Delta w} \operatorname{div}_\Delta(F)$.

Theorem 4 ([4, Theorem 6]). *Let $W = W_1, \ldots, W_R$ be a family of random variables, taking values in $\{0, 1, \perp\}$ and satisfying the (f, γ)-characteristic conditions. If $\Delta > 0$ and $\epsilon > 0$ satisfy $\gamma(1-f)^{\Delta-1} \geq (1+\epsilon)/2$ then*

$$\Pr[\operatorname{div}_\Delta(W) \geq k + \Delta] \leq \frac{19R}{\epsilon^4} \exp(-\epsilon^4 k/18)\,.$$

This general statement allows us to translate the properties of the characteristic string distribution induced by an execution of Ouroboros-Chronos$_{\mathsf{mc}}$, as recorded in Lemma 3, into a common-prefix guarantee for the protocol, given below.

Corollary 2 (Common prefix). *Let $W = W_1, \ldots, W_R$ denote the characteristic string induced by the* Ouroboros-Chronos$_{\mathsf{mc}}$ *protocol in the single-epoch setting with static registration and Δ-bounded skew. Let R be the epoch length and f the active-slot coefficient. Assume that $\epsilon > 0$ satisfies*

$$\alpha(1-f)^{\widetilde{\Delta}+1} \geq (1+\epsilon)/2\,,$$

where α is a lower-bound on the alert stake ratio over the execution and $\widetilde{\Delta} = 2\Delta$ is twice the network delay. Then

$$\Pr[\mathrm{div}_{\widetilde{\Delta}}(W) \geq k + \widetilde{\Delta}] \leq \frac{19R}{\epsilon^4} \exp(-\epsilon^4 k/18)\,,$$

and hence a k-common-prefix violation occurs with probability at most

$$\bar{\epsilon}_{\mathsf{CP}}(k; R, \Delta, \epsilon) \triangleq \frac{19R}{\epsilon^4} \exp(\widetilde{\Delta} - \epsilon^4 k/18)\,.$$

Proof. Follows directly from Theorem 4 and Lemma 3, using the first statement of Lemma 2 as a bound on the observed message delivery delay. □

The remaining Corollaries 3 (resp., 4, 5) below can be established by following the same reasoning as used in the proof of [4, Corollary 4] (resp., [4, Corollary 5], [4, Lemma 11]) with respect to logical slots instead of the nominal time, and using the first statement of Lemma 2 to bound the observed message delivery delay. □

Corollary 3 (Chain Growth). *Let $W = W_1, \ldots, W_R$ denote the characteristic string induced by the Ouroboros-Chronos$_{\mathsf{mc}}$ protocol in the single-epoch setting with static registration and Δ-bounded skew. Let R be the epoch length and f the active-slot coefficient. Let $\alpha, \beta \in [0, 1]$ denote lower bounds on the alert stake ratio and the participating stake ratio over the execution as per Definition 2, and assume that for some some $\varepsilon \in (0, 1)$ the parameter α satisfies*

$$\alpha(1 - f)^{\widetilde{\Delta}+1} \geq (1 + \epsilon)/2\,.$$

where $\widetilde{\Delta} = 2\Delta$ is twice the network delay. Then for

$$s = 48\widetilde{\Delta}/(\epsilon\beta f) \qquad \text{and} \qquad \tau = \beta f/16 \tag{8}$$

we have

$$\Pr[W \text{ admits a } (s, \tau)\text{-CG violation}] \leq \bar{\epsilon}_{\mathsf{CG}}(\tau, s; R, \epsilon) \triangleq \frac{1}{2} s R^2 \exp\left(-(\epsilon\beta f)^2 s/256\right)\,.$$

Corollary 4 (Chain Quality). *Let $W = W_1, \ldots, W_R$ denote the characteristic string induced by the Ouroboros-Chronos$_{\mathsf{mc}}$ protocol in the single-epoch setting with static registration and Δ-bounded skew. Let R be the epoch length and f the active-slot coefficient. Let $\alpha, \beta \in [0, 1]$ denote lower bounds on the alert stake ratio and the participating stake ratio as per Definition 2, and assume that for some some $\varepsilon \in (0, 1)$ the parameter α satisfies*

$$\alpha(1 - f)^{\widetilde{\Delta}+1} \geq (1 + \epsilon)/2\,.$$

where $\widetilde{\Delta} = 2\Delta$ is twice the network delay. Then for

$$k = 48\widetilde{\Delta}/(\epsilon\beta f) \qquad \text{and} \qquad \mu = \epsilon\beta f/16$$

we have

$$\Pr[W \text{ admits a } (\mu, k)\text{-CQ violation}] \leq \bar{\epsilon}_{\mathsf{CQ}}(\mu, k; R, \epsilon) \triangleq \frac{1}{2} k R^2 \exp\left(-(\epsilon\beta f)^2 k/256\right)\,.$$

Corollary 5 (Existential Chain Quality). *Let $W = W_1, \ldots, W_r$ denote the characteristic string induced by the protocol Ouroboros-Chronos in the single-epoch setting with static registration and Δ-bounded skew. Let R be the epoch length and f the active-slot coefficient. Let $\alpha, \beta \in [0, 1]$ denote lower bounds on the alert stake ratio and the participating stake ratio over the execution as per Definition 2, and assume that for some $\varepsilon \in (0, 1)$ the parameter α satisfies*

$$\alpha(1 - f)^{\widetilde{\Delta}+1} \geq (1 + \epsilon)/2\,,$$

where $\widetilde{\Delta} = 2\Delta$ is twice the network delay. Then for $s \geq 12\widetilde{\Delta}/(\epsilon\beta f)$,

$$\Pr[W \text{ admits a } s\text{-}\exists\mathsf{CQ} \text{ violation}] \leq \bar{\epsilon}_{\exists\mathsf{CQ}}(s; r, \epsilon) = r^2(s + 1) \exp\left(-(\epsilon\beta f)^2 s/64\right)\,.$$

F.5 Switching to maxvalid-bg

To capture the security of the full protocol Ouroboros Chronos with the chain-selection rule maxvalid-bg given in Section C.8, the bounds above need to be adjusted by an additional term that reflects the probability that the new chain selection rule deviates from the easier-to-analyze rule maxvalid-mc. This error term was quantified by Theorem 2 in [4] and this quantification translates directly into our setting.

Corollary 6 (Security of maxvalid-bg). *Consider the protocol Ouroboros-Chronos (with maxvalid-bg), executed in the same setting and under the same assumptions as in Corollaries 2–5 above. If the maxvalid-bg parameters, k and s, satisfy*

$$k > 192\widetilde{\Delta}/(\epsilon\beta) \qquad \text{and} \qquad R/6 \geq s = k/(4f) \geq 48\widetilde{\Delta}/(\epsilon\beta f)$$

then the guarantees given in Corollaries 2–5 for common prefix, chain growth, chain quality and existential chain quality are also valid for Ouroboros-Chronos except for an additional error probability

$$\bar{\epsilon}_{\mathsf{mv}} \triangleq \exp\left(\ln R - \Omega(k)\right) + \bar{\epsilon}_{\mathsf{CG}}(\beta f/16, k/(4f)) + \bar{\epsilon}_{\exists\mathsf{CQ}}(k/(4f)) + \bar{\epsilon}_{\mathsf{CP}}(k\beta/64)\ , \tag{9}$$

where the subscript "mv" stands for "maxvalid".

Proof (sketch). The corollary follows by the same reasoning as Theorem 2 in [4], again by using the first statement of Lemma 2 to bound the observed message delivery delay. □

Putting things together. The proof of Theorem 1 now follows by combining Corollaries 2, 3, 4, 5 and 6.

G Analysis of the Synchronization Procedure

G.1 The Proof of Fact 1

Proof (of Fact 1). Without loss of generality order the pairs so that $a_i \leq a_j$ for $i < j$ (and note that this does not necessarily imply $b_i \leq b_j$ for $i < j$). Now we have

$$\begin{aligned} a_{\lceil n/2\rceil} - \Delta &\overset{(a)}{\leq} \min\left\{b_i \,:\, \lceil n/2\rceil \leq i \leq n\right\} \overset{(b)}{\leq} \mathsf{med}\left((b_i)_{i=1}^n\right) \\ &\overset{(c)}{\leq} \max\left\{b_i \,:\, 1 \leq i \leq \lceil n/2\rceil\right\} \overset{(d)}{\leq} a_{\lceil n/2\rceil} + \Delta\ , \end{aligned}$$

where inequalities (a) and (d) follow from the assumption of the lemma and the assumed ordering of the values a_i, inequalities (b) and (c) follow from the definition of med. The proof is concluded by observing that $\mathsf{med}((a_i)_{i=1}^n) = a_{\lceil n/2\rceil}$ by definition. □

G.2 SyncProc maintains Skew_{Δ}

Lemma 4 (SyncProc maintains Skew_{Δ}). *Consider an execution of the full protocol Ouroboros-Chronos over a lifetime of $L = ER$ slots, where R is the epoch length. Let Δ_{net} be the upper bound on message delay enforced by $\mathcal{F}_{\text{N-MC}}$ and let Δ_{clock} be the maximal drift enforced by $\mathcal{G}_{\text{ImperfLClock}}$ and let $\Delta = \Delta_{\text{clock}} + \Delta_{\text{net}}$; and let $\mathtt{sl} \geq 1$ be the last slot of some epoch $\mathtt{ep} \geq 1$, i.e., such that $\mathtt{sl} \mod R = 0$. If the properties $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$ and $\mathsf{CP}(\tau_{\mathsf{CG}}R/3)$ for τ_{CG} as in Theorem 1 are not violated during the execution up to slot $\mathtt{sl}$, then the predicate $\mathsf{Skew}_{\Delta}[\mathtt{sl}']$ is satisfied for $\mathtt{sl}' = \mathtt{sl} + 1$ onward to the next synchronization slot.*

Proof. We first establish two intermediate claims under the lemma assumptions:

(i) All alert parties use the same set of synchronization beacons in their execution of the procedure SyncProc between epochs ep and ep + 1, formally $\mathcal{S}_j^{\mathsf{P}_1} = \mathcal{S}_j^{\mathsf{P}_2}$ for any two parties P_1,P_2 that are alert in the j-th synchronization slot.

(ii) For any fixed beacon $\mathtt{SB} \in \mathcal{S}_j^{\mathsf{P}_1} = \mathcal{S}_j^{\mathsf{P}_2}$, the quantity

$$\mu(\mathsf{P}_i, \mathtt{SB}) \triangleq \mathsf{Skew}^{\mathsf{P}_i}[\mathtt{sl}] + \mathsf{slotnum}(\mathtt{SB}) - \mathsf{P}_i.\mathsf{Timestamp}(\mathtt{SB})$$

will differ by at most Δ between any two alert parties P_1 and P_2.

To see (i), note that the set $\mathcal{S}_j^{\mathsf{P}}$ is constructed by the party P by collecting all beacons $\mathtt{SB}$ that report a slot number $\mathsf{slotnum}(\mathtt{SB})$ within $[(i-1)\cdot R+1,\ldots,(i-1)\cdot R+R/6]$ (the preceding synchronization interval) and which are included in a block of the adopted chain $\mathsf{P}.\mathcal{C}_{\mathsf{loc}}$ of P up to slot $(i-1)\cdot R+2R/3$. Based on the chain growth property, the chain $\mathcal{C}_{\mathsf{loc}}$ contain as least $\tau_{\mathsf{CG}}R/3$ blocks in the last $R/3$ slots, and by the common prefix property, the chains are identical up to slot $(i-1)\cdot R+2R/3$.

Now observe that

$$\mu(\mathsf{P}_i,\mathtt{SB}) = (\mathsf{slotnum}(\mathtt{SB})-t) - \theta_{\mathsf{P}_i,\mathtt{SB}} - \delta_{\mathsf{P}_i,\mathtt{SB}}\,, \tag{10}$$

where t is the nominal time at the point of the execution where $\mathtt{SB}$ was sent, and $\delta_{\mathsf{P}_i,\mathtt{SB}} \in [\Delta_{\text{net}}]$ is the number of rounds that $\mathtt{SB}$ was delayed in its transit from $\tilde{\mathsf{P}}$ to P_i , i.e., where one round is counted per local clock update of P_i, and where $\theta_{\mathsf{P}_i,\mathtt{SB}} \in [\Delta_{\text{clock}}]$ is the drift of P_i from the nominal time t at the point of the execution when $\mathtt{SB}$ was sent. Clearly, equation (10) establishes (ii). Note that (10) holds even if the sender $\tilde{\mathsf{P}}$ of $\mathtt{SB}$ was corrupted when sending it (owing to the network model); for alert parties $\tilde{\mathsf{P}}$ the first bracket in (10) is equal to $\mathsf{Skew}^{\tilde{\mathsf{P}}}[\mathsf{slotnum}(\mathtt{SB})]$ which includes any drift from the sender to the nominal time by definition.

Given the above properties, we invoke Fact 1 for the tuples

$$(\mu(\mathsf{P}_1,\mathtt{SB}))_{\mathtt{SB}\in\mathcal{S}_j^{\mathsf{P}_1}} \text{ and } (\mu(\mathsf{P}_2,\mathtt{SB}))_{\mathtt{SB}\in\mathcal{S}_j^{\mathsf{P}_2}}$$

for two arbitrary alert parties P_1 and P_2. By property (i) both tuples are of the same size, and by property (ii) one can see that the update mechanism of Chronos will yield close time stamps because the preconditions of Fact 1 are fulfilled by some $\Delta' \le \Delta$. Therefore we obtain

$$\left|\mathsf{med}\left((\mu(\mathsf{P}_1,\mathtt{SB}))_{\mathtt{SB}\in\mathcal{S}_j^{\mathsf{P}_1}}\right) - \mathsf{med}\left((\mu(\mathsf{P}_2,\mathtt{SB}))_{\mathtt{SB}\in\mathcal{S}_j^{\mathsf{P}_2}}\right)\right| \le \Delta'$$

and further, for $i\in\{1,2\}$, $\mathsf{med}\left((\mu(\mathsf{P}_i,\mathtt{SB}))_{\mathtt{SB}\in\mathcal{S}_j^{\mathsf{P}_i}}\right) = \mathsf{Skew}^{\mathsf{P}_i}[\mathtt{sl}+1]$ by the update mechanism of Chronos. Lemma 4 finally follows by observing that the maximal drift of a party in the execution of the system is an absolute term, and thus the quantity $\Delta_{\text{clock}} - \max\{\theta_{\mathsf{P}_i,\mathtt{SB}} \,|\, \mathtt{SB}\in\mathcal{S}_j^{\mathsf{P}_i}\}$ is the remaining additional term (measured in number of rounds) that the party P_i can be pushed forward (and hence increasing the actual skew Δ') in the execution without the baseline (and therefore everyone) moving forward. We see from the computation in equation (10) that the the skew will never exceed Δ and can thus establish $\mathsf{Skew}_{\Delta}[\mathtt{sl}']$ for $\mathtt{sl}' = \mathtt{sl}+1$ and onward up to the next synchronization slot $j+1$. □

G.3 Bounded shift

Lemma 5 (Bounded shift). *Consider an execution of the full protocol Ouroboros-Chronos over a lifetime of $L = ER$ slots, where R is the epoch length. Let Δ_{net} be the upper bound on message delay enforced by $\mathcal{F}_{\text{N-MC}}$ and let Δ_{clock} be the maximal drift enforced by $\mathcal{G}_{\text{ImperfLClock}}$ and let $\Delta = \Delta_{\text{clock}} + \Delta_{\text{net}}$. Further assume $\widetilde{\Delta} \triangleq 2\Delta \le R/6$. Let $\mathtt{sl} \ge 1$ be the last slot of some epoch $\mathtt{ep} \ge 1$, i.e., such that $\mathtt{sl} \bmod R = 0$, and assume that $\mathsf{Skew}_{\Delta}[\mathtt{sl}']$ is satisfied for all slots in epoch $\mathtt{ep}$. Let $\alpha\in[0,1]$ denote a lower bound on the alert ratio and participating ratio throughout the execution. If for some $\epsilon\in(0,1)$ we have $\alpha\cdot(1-f) \ge (1+\epsilon)/2$, and if the property $\exists\mathsf{CQ}(R/3)$ is not violated during the execution up to slot $\mathtt{sl}$, then in any invocation of $\mathsf{SyncProc}$ by an alert party during $\mathtt{sl}$, the local variable shift computed on line 17 will satisfy $|\mathsf{shift}| \le 2\Delta$, except with error probability $\exp(\ln L - \Omega(R))$ over the whole execution.*

Proof. We first show that the set $\mathcal{S}_i^{\mathsf{P}}$ used by any alert party P contains all beacon messages produced by alert parties in the last synchronization interval. This follows by observing that all synchronization beacons produced by alert parties in (their) slots $[(i-1)\cdot R+1,\ldots,(i-1)\cdot R+R/6]$ will be delivered to other alert parties by the slot $(i-1)\cdot R+R/3$ by the assumption $\widetilde{\Delta} \le R/6$ and by the first statement of Lemma 2. Moreover, by the $\exists\mathsf{CQ}$ property, the chain $\mathcal{C}_{\mathsf{loc}}$ of any alert party P will contain at least one block created by an alert party over the slot interval $[(i-1)\cdot R+R/3+1,\ldots,(i-1)\cdot R+2R/3]$. When such an alertly-created block is included, it will contain all the synchronization beacons produced by alert parties for slots $[(i-1)\cdot R+1,\ldots,(i-1)\cdot R+R/6]$ that were not included yet.

Next, in light of the lower bound on alert stake ratio, a majority of synchronizing beacons in $\mathcal{S}_i^{\mathsf{P}}$ will be alertly-generated, except with error probability $\exp(\ln L - \Omega(R))$. Therefore, if this error does not occur, due

to the use of median in the definition of shift there exist alert parties $\mathsf{P}_1, \mathsf{P}_2$, which produced synchronization beacons $\mathtt{SB}_1, \mathtt{SB}_2 \in \mathcal{S}_i^{\mathsf{P}}$ respectively, such that

$$\mathsf{Skew}^{\mathsf{P}_1}[\mathsf{slotnum}(\mathtt{SB}_1)] - \Delta \overset{(a)}{\leq} \mathsf{Skew}^{\mathsf{P}}[\mathtt{sl}] + \mathsf{shift} \overset{(b)}{\leq} \mathsf{Skew}^{\mathsf{P}_2}[\mathsf{slotnum}(\mathtt{SB}_2)] \ .$$

The inequalities (a) and (b) follow again from the core computation in equation (10), this time considering $\mu(\mathsf{P}, \mathtt{SB}_1)$ and $\mu(\mathsf{P}, \mathtt{SB}_2)$ and the bounds $\delta_{\mathsf{P},\mathtt{SB}_1} + \theta_{\mathsf{P}_i,\mathtt{SB}_1} \leq \Delta$ and $\delta_{\mathsf{P},\mathtt{SB}_2}, \theta_{\mathsf{P}_i,\mathtt{SB}_2} \geq 0$, respectively. The proof is now concluded by observing that for both $i \in \{1, 2\}$, we have

$$\left|\mathsf{Skew}^{\mathsf{P}_i}[\mathsf{slotnum}(\mathtt{SB}_i)] - \mathsf{Skew}^{\mathsf{P}}[\mathtt{sl}]\right| \leq \Delta$$

thanks to the assumption $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$. □

H Security for the Full Execution with Static Registration

The following theorem is a formal statement corresponding to the informal Theorem 2.

Theorem 5 (Full-execution security with static registration). *Consider the execution of Ouroboros-Chronos with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ in the setting with static registration. Let f be the active-slot coefficient, let Δ be the upper bound on the sum of the maximum network delay and maximum local clock drifts, and let $\widetilde{\Delta} \triangleq 2\Delta$. Let $\alpha, \beta \in [0, 1]$ denote a lower bound on the alert and participating stake ratios throughout the whole execution, respectively. Let R and L denote the epoch length and the total lifetime of the system (in slots), and let Q be the total number of queries issued to $\mathcal{G}_{\mathrm{RO}}$. If the assumptions (5) and (6) are satisfied, then Ouroboros-Chronos achieves the same guarantees for common prefix (resp. chain growth, chain quality, existential chain quality) as given in Theorem 1 (with L replacing R as execution length) except with an additional error probability of*

$$QL \cdot \left(\bar{\epsilon}_{\mathsf{CG}}\left(\tau_{\mathsf{CG}}, R/3; R, \epsilon\right) + \bar{\epsilon}_{\mathsf{CP}}\left(\tau_{\mathsf{CG}} R/3; R, \Delta, \epsilon\right) + \bar{\epsilon}_{\exists\mathsf{CQ}}\left(R/3; R, \epsilon\right)\right) \ , \tag{11}$$

where $\tau_{\mathsf{CG}} = \beta f/16$. If $R \geq 144\widetilde{\Delta}/\epsilon\beta f$ then this term can be upper-bounded by

$$\epsilon_{\mathsf{lift}} \triangleq QL \cdot \left[R^3 \cdot \exp\left(-\frac{(\epsilon\beta f)^2 R}{768}\right) + \frac{19R}{\epsilon^4} \cdot \exp\left(\widetilde{\Delta} - \frac{\epsilon^4 \tau_{\mathsf{CG}} R}{54}\right) + 3\bar{\epsilon}_{\mathsf{mv}}\right] \ . \tag{12}$$

For all $\mathsf{p} \in \{\mathsf{CP}, \mathsf{CG}, \exists\mathsf{CQ}, \mathsf{CQ}\}$, we denote the obtained counterparts of the single-epoch error terms $\bar{\epsilon}_{\mathsf{p}}$ for the full execution with static registration by ϵ_{p}.

Proof (of Theorem 5, sketch). When moving from the single-epoch setting to a setting with several epochs, the following aspects need to be considered:

- **Stake distribution updates.** The stake distribution used for sampling slot leaders changes in every epoch. In Ouroboros-Chronos the distribution used for sampling in epoch $\mathtt{ep}$ is set to be the stake distribution recorded on the blockchain up to the last block of the epoch $\mathtt{ep} - 2$.
- **Randomness updates.** Every epoch needs new public randomness to be used for the private leader election process based on the above distribution. For epoch $\mathtt{ep}$, this randomness is obtained by hashing together VRF-outputs put into blocks in epoch $\mathtt{ep} - 1$ by their creators. More precisely, the protocol hashes together these values from the blocks in the first $2R/3$ slots of epoch $\mathtt{ep} - 1$ (out of its R slots).
- **Resynchronization.** All alert parties execute the resynchronization procedure $\mathsf{SyncProc}$ in the last slot of every epoch.

This proof partly follows the treatment in Section 5 of [14] to argue about stake distribution and randomness updates, and hence we only sketch the reasoning for these parts, adding a discussion of the resynchronization. The proof has an inductive structure over the epochs of the execution.

First, note that in the first epoch, we have both perfect epoch randomness, and the property $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ is satisfied for all slots $\mathtt{sl}$ in this epoch, by definition of the functionality $\mathcal{F}_{\mathsf{INIT}}$ and $\mathcal{G}_{\text{IMPERFLCLOCK}}$. More precisely, they ensure that $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ for all slots in the first epoch (where Δ is the bound on the sum of all incurred delays and clock inaccuracies) and all alert parties advance their local clock at least at the pace of

the baseline. Given that, we can apply Theorem 1 to the first epoch and obtain the bounds for CP, CG and CQ it provides.

For the induction step, Lemma 4 shows that the properties $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$ and $\mathsf{CP}(\tau_{\mathsf{CG}}R/3)$ satisfied in epoch $\mathtt{ep}$ imply $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ for all slots in epoch $\mathtt{ep}+1$. Furthermore, $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$ and $\mathsf{CP}(\tau_{\mathsf{CG}}R/3)$ during the first $R/3$ slots of epoch $\mathtt{ep}$ also imply that each alert player's chain grows by at least $\tau R/3$ blocks and after these slots, all alert players agree on the stake distribution at the end of epoch $\mathtt{ep}-1$. Moreover, $\exists\mathsf{CQ}(R/3)$ implies that during the second $R/3$ slots of epoch $\mathtt{ep}$, each alert player's chain contains at least one honest block. Hence the randomness that will be derived for epoch $\mathtt{ep}+1$ will be influenced by at least one honest VRF-output chosen *after* the stake distribution for $\mathtt{ep}+1$ is fixed. Finally, $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$ and $\mathsf{CP}(\tau_{\mathsf{CG}}R/3)$ during the last $R/3$ slots of epoch $\mathtt{ep}$ imply that each alert player's chain grows by at least $\tau R/3$ blocks and therefore after these slots, all alert players agree on the randomness for the epoch $\mathtt{ep}+1$.

Exactly as in [14,4], we need to additionally account for the limited amount of grinding that the adversary can achieve by deciding whether to include blocks (with VRF outputs) in slots where he is a slot leader. This can be crudely upper-bounded by limiting the number of queries to the random oracle that the adversary makes, adding an additional quantity Q into our bound.

Now, having established $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ for the slots in epoch $\mathtt{ep}+1$ as well as accounted for the quality of the randomness used in epoch $\mathtt{ep}+1$, we can again invoke Theorem 1 to include $\mathtt{ep}+1$ to obtain guarantees on CP, CG, and CQ and complete the induction step.

Finally, the bound (12) is obtained by instantiating (11) with the concrete bounds of Theorem 1. □

I Analysis of Joining

Lemma 6. *Consider an execution of the full protocol Ouroboros-Chronos and let* $\mathsf{P}_{\mathsf{join}}$ *be a party joining the protocol execution at time* $t_{\mathsf{join}} > 0$ *that retains its access to all resources during its joining procedure JoinProc (cf. Fig. 2). Let* $t \in (t_{\mathsf{join}} + t_{\mathrm{off}}, t_{\mathsf{join}} + t_{\mathrm{off}} + t_{\mathrm{minSync}} + t_{\mathrm{stable}} + 1]$ *be the (nominal) time at some point in the execution in which* $\mathsf{P}_{\mathsf{join}}$ *is in Phase C or D of its joining procedure and let* $\mathcal{C}_{\mathsf{join}}$ *denote the chain held by* $\mathsf{P}_{\mathsf{join}}$ *at that point. Let* $\mathcal{C}_{\mathsf{alert}}$ *denote a chain held by any alert party* $\mathsf{P}_{\mathsf{alert}}$ *at some point in the execution where time* $t' \triangleq t - \Delta$*. Then under the assumptions of Theorem 5 and assuming no violations of* $\mathsf{CP}(k\beta/64)$, $\exists\mathsf{CQ}(s)$, *and* $\mathsf{CG}(\tau_{\mathsf{CG}}, s)$ *until the end of the joining procedure (for the parameters* k *and* s *of maxvalid-bg), we have* $\mathcal{C}_{\mathsf{alert}}^{\lceil k} \preceq \mathcal{C}_{\mathsf{join}}$ *except with error probability* $\exp(\ln L - \Omega(R))$ *over the whole execution.*

Proof (sketch). Notice that the chain-selection procedure maxvalid-bg given in Fig. C.8 does not involve the local time `localTime` of the party executing it. Therefore, $\mathsf{P}_{\mathsf{join}}$ and $\mathsf{P}_{\mathsf{alert}}$ would do the same chain-selection decisions in their maxvalid-bg procedures, if presented with the same inputs. The only (but crucial) difference in their chain-adoption behavior comes from the fact that $\mathsf{P}_{\mathsf{alert}}$ has local clock that satisfies the Skew_{Δ} predicate, and based on this local clock the party removes from consideration all received chains that contain blocks from its logical future (with timestamp larger than its local time), this is done on line 4 of procedure IsValidChain in Fig. C.5. Of course, $\mathsf{P}_{\mathsf{join}}$ does no such filtering as it does not possess reliable local time information yet.

To see the implications of this difference, we consider the concept of a *virtual execution* for $\mathsf{P}_{\mathsf{join}}$ introduced in [4]. This is an artificial random experiment that consists of the execution of the protocol with an additional ("*virtual*") party $\mathsf{P}_{\mathsf{virt}}$ that participates from the beginning, is always alert, but commands no stake and hence is passive. Moreover, starting from the point of execution where $\mathsf{P}_{\mathsf{join}}$ joins the system, the virtual party advances exactly like $\mathsf{P}_{\mathsf{join}}$ and receives the same messages in the same slots and order as $\mathsf{P}_{\mathsf{join}}$.

The only case when $\mathsf{P}_{\mathsf{join}}$ may adopt as its local chain a chain $\mathcal{C}_{\mathsf{join}}$ that $\mathsf{P}_{\mathsf{virt}}$ does not adopt over the chain it is currently holding (call it $\mathcal{C}_{\mathsf{virt}}$) is if $\mathcal{C}_{\mathsf{join}}$ contains an adversarially-created suffix of future blocks such that it dominates $\mathcal{C}_{\mathsf{virt}}$ based on Condition A in maxvalid-bg. (As proved in [4, Theorem 2], the adversary is not capable of creating a chain that would dominate an alert chain according to Condition B under the assumptions of the lemma, except for a global bad event with probability $\exp(\ln L - \Omega(R))$.) However, Condition A is only applied if $\mathcal{C}_{\mathsf{virt}}^{\lceil k} \preceq \mathcal{C}_{\mathsf{join}}$ so this must have been the case when $\mathsf{P}_{\mathsf{join}}$ adopted $\mathcal{C}_{\mathsf{join}}$ prior to the point of the execution at time t we are considering in the lemma statement. Moreover, since $\mathsf{P}_{\mathsf{join}}$ is still holding $\mathcal{C}_{\mathsf{join}}$ at the point under consideration at time t, it means that since it adopted it, $\mathsf{P}_{\mathsf{virt}}$ has not received

any chain $\mathcal{C}'_{\mathsf{virt}}$ that would violate $\mathcal{C}'^{\lceil k}_{\mathsf{virt}} \preceq \mathcal{C}_{\mathsf{join}}$, as in that case also $\mathsf{P}_{\mathsf{join}}$ would receive and adopt it (as follows by inspection of maxvalid-bg). Therefore, the chain $\mathcal{C}'_{\mathsf{virt}}$ that $\mathsf{P}_{\mathsf{virt}}$ holds at the point under consideration at time t satisfies $\mathcal{C}'^{\lceil k}_{\mathsf{virt}} \preceq \mathcal{C}_{\mathsf{join}}$. Finally, if any alert party $\mathsf{P}_{\mathsf{alert}}$ held at nominal time t' a chain $\mathcal{C}_{\mathsf{alert}}$ that would violate $\mathcal{C}^{\lceil k}_{\mathsf{alert}} \preceq \mathcal{C}_{\mathsf{join}}$, by our assumptions on the network delay and clock drift, it would be delivered and considered by $\mathsf{P}_{\mathsf{virt}}$ at any point in the execution where nominal time is t and hence adopted, concluding the proof. □

Lemma 7. *Consider an execution of the full protocol Ouroboros-Chronos and let $\mathsf{P}_{\mathsf{join}}$ be a party joining the protocol execution at time $t_{\mathsf{join}} > 0$ that retains its access to all resources during its joining procedure JoinProc (cf. Fig. 2). Under the assumptions of Theorem 5 and Lemma 6, and assuming no violations of $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$, $\mathsf{CP}(\tau_{\mathsf{CG}} R/3)$, and $\exists\mathsf{CQ}(R/3)$ until the end of the joining procedure, we have the following except with error probability $\exp(\ln L - \Omega(R))$ over the whole execution:*

(a) The index value i^* determined on line 32 of its joining procedure JoinProc satisfies $i^* \geq 1$.

(b) For all values of $i \geq i^*$ processed in the iteration on lines 34–52 we have $\mathcal{S}_i^{\mathsf{P}_{\mathsf{join}}} = \mathcal{S}_i^{\mathsf{P}_{\mathsf{alert}}}$, where $\mathcal{S}_i^{\mathsf{P}_{\mathsf{join}}}$ is the set of synchronization beacons determined by $\mathsf{P}_{\mathsf{join}}$ on line 35 and $\mathcal{S}_i^{\mathsf{P}_{\mathsf{alert}}}$ is the set of synchronization beacons determined by any alert party $\mathsf{P}_{\mathsf{alert}}$ on line 6 of its procedure SyncProc for the same i.

(c) For all values of $i \geq i^*$ processed in the iteration on lines 34–52 and for any fixed beacon $\mathcal{SB} \in \mathcal{S}_i^{\mathsf{P}_{\mathsf{join}}} = \mathcal{S}_i^{\mathsf{P}_{\mathsf{alert}}}$, the quantity $\mu(\mathsf{P}, \mathcal{SB})$, which is defined to be $\mathcal{Skew}^{\mathsf{P}}[\mathtt{sl}] + \mathsf{slotnum}(\mathcal{SB}) - \mathsf{P}.\mathsf{Timestamp}(\mathcal{SB})$, will differ by at most Δ between the two parties $\mathsf{P} \in \{\mathsf{P}_{\mathsf{join}}, \mathsf{P}_{\mathsf{alert}}\}$, and furthermore, when the joining party becomes alert $\mathcal{Skew}_{\Delta}[\mathtt{sl}]$ is satisfied for any slot in which the joining party is considered alert.

Proof (sketch). We first show the claim (a) that condition $i^* \geq 1$ on line 32 of JoinProc will be satisfied for $\mathsf{P}_{\mathsf{join}}$. Informally speaking, this means that while $\mathsf{P}_{\mathsf{join}}$ executed Phase C of its joining procedure JoinProc (recall that line 32 is only executed after that), it has observed at least one full synchronization interval $I_{\mathrm{sync}}(i^*)$ that started at least t_{pre} rounds after the beginning of Phase C; and has recorded timestamps (in its data structure Timestamp) for all synchronization beacons SB recorded in its local chain and coming from $I_{\mathrm{sync}}(i)$ according to their included logical slot numbers. Also recall that when we talk about an interval d of rounds that a party locally executes, then this relates to the number of rounds all other parties have executed by an additive offset $\delta \leq \Delta_{\mathrm{clock}}$ and we choose the all intervals in this proof in such a way that it encompasses this small offset.

To proceed, let us split Phase C into two consecutive, non-overlapping Phases $\mathrm{C}_{\mathsf{sync}}$ and $\mathrm{C}_{\mathsf{stable}}$ consisting of t_{minSync} and t_{stable} rounds, respectively. Let $t^{(j)}_{\mathsf{start}}$ denote the nominal time in which it happens for the first time that an alert party enters the synchronization interval $I_{\mathrm{sync}}(j)$ according to its local clock (i.e., enters the logical slot $(j-1)R+1$). Then for all $j \geq 1$ we have

$$t^{(j+1)}_{\mathsf{start}} - t^{(j)}_{\mathsf{start}} \leq R + 3\Delta \leq 13R/12 \tag{13}$$

thanks to the fact that there is a synchronization interval starting at the first slot of every R-slot epoch, the bound of Lemma 5 and the assumptions Skew_{Δ} and (6).

Let now i^* denote the minimal i such that $t^{(i)}_{\mathsf{start}} \geq t_{\mathsf{join}} + t_{\mathrm{off}} + t_{\mathrm{pre}}$, i.e., $t^{(i^*)}_{\mathsf{start}}$ occurs at least t_{pre} rounds *after* the beginning of $\mathsf{P}_{\mathsf{join}}$'s Phase C. According to (13) we have

$$t^{(i^*)}_{\mathsf{start}} \leq t_{\mathsf{join}} + t_{\mathrm{off}} + t_{\mathrm{pre}} + 13R/12 \leq t_{\mathsf{join}} + t_{\mathrm{off}} + t_{\mathrm{minSync}} - R/6$$

by the values of t_{pre} and t_{minSync} (cf. Table 1). Therefore $t^{(i^*)}_{\mathsf{start}}$ is guaranteed to occur at least $R/6$ rounds before the end of Phase $\mathrm{C}_{\mathsf{sync}}$ for $\mathsf{P}_{\mathsf{join}}$.

We now argue that i^* will satisfy the condition on line 32 of JoinProc. This follows as by round $t^{(i^*)}_{\mathsf{start}}$, $\mathsf{P}_{\mathsf{join}}$ has been already recording the timestamps of all received synchronization beacons for t_{pre} rounds, and hence, has recorded into its Timestamp data structure the timestamps of all beacons that (according to the logical slot number they contain) belong to the synchronization interval $I_{\mathrm{sync}}[i^*]$ — either by receiving the beacon directly or observing it as included in a blockchain block. This is exactly what is needed for i^* to pass the test on line 32 of JoinProc. Note that the adversary cannot create valid beacons logically belonging to $I_{\mathrm{sync}}[i]$

before the start of $\mathsf{P}_{\mathsf{join}}$'s Phase C, as before that point the epoch randomness necessary for creating valid synchronization beacons for this synchronization interval is still completely unpredictable (thanks to the choice of t_{pre}).

Moving to claim (b), we first establish it for i^*. This can be argued in a similar way as the validity of item (i) in the proof of Lemma 4: the set $\mathcal{S}_{i^*}^{\mathsf{P}}$ is for both $\mathsf{P} \in \{\mathsf{P}_{\mathsf{join}}, \mathsf{P}_{\mathsf{alert}}\}$ constructed by collecting all beacons $\mathtt{SB}$ (satisfying certain conditions on the reported slot number) from the adopted chain $\mathsf{P}.\mathcal{C}_{\mathsf{loc}}$ of P up to slot $(i^*-1)\cdot R + 2R/3$. As observed above, the synchronization interval $I_{\mathrm{sync}}(i^*)$ will start (from the perspective of the first alert party) at least $R/6$ rounds before the end of $\mathsf{P}_{\mathsf{join}}$'s Phase $\mathrm{C}_{\mathsf{sync}}$, and hence will also end (for this alert party) before the end of Phase $\mathrm{C}_{\mathsf{sync}}$. Therefore, it will be followed by $t_{\mathrm{stable}} = R$ rounds of Phase $\mathrm{C}_{\mathsf{stable}}$, and the relevant beacons will be collected from up to round $2R/3$ of Phase $\mathrm{C}_{\mathsf{stable}}$ (to account for the potential skew of other alert parties and delays plus drifts, as in (13)). Assuming $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$ and $R \geq 3k\tau_{\mathsf{CG}}$ (which follows from (6)), we get that the chain held by $\mathsf{P}_{\mathsf{alert}}$ grows by at least k blocks during the last $R/3$ slots, and is hence identical to the chain held by $\mathsf{P}_{\mathsf{join}}$ up to slot $(i^*-1)\cdot R + 2R/3$ by Lemma 6, resulting in $\mathcal{S}_{i^*}^{\mathsf{P}_{\mathsf{join}}} = \mathcal{S}_{i^*}^{\mathsf{P}_{\mathsf{alert}}}$. The above reasoning applies identically also to all following values of $i \geq i^*$ that the iteration on lines 34–52 considers.

Finally, claim (c) follows by similar arguments as in the proof of Lemma 4 (relying on the network and clock model guarantees). □

J Full Security in the Dynamic Availability Setting

The following theorem is a formal statement corresponding to the informal Theorem 3.

Theorem 6 (Dynamic availability). *Consider an execution of the full protocol Ouroboros-Chronos in the dynamic-availability setting. Under the assumptions of Theorem 5 and Lemma 7, Ouroboros-Chronos achieves the same guarantees for common prefix (resp. chain growth, chain quality, existential chain quality) as given in Theorem 5 except for the negligible additional error probability*

$$\epsilon_{\mathsf{DA}} \triangleq \epsilon_{\mathsf{CP}} \max\left\{\frac{k\beta}{64}, \frac{\tau_{\mathsf{CG}} R}{3}\right\} + \epsilon_{\mathsf{CG}}(\tau_{\mathsf{CG}}, s) + \epsilon_{\exists\mathsf{CQ}}(\tau_{\mathsf{CG}} R/3) + e^{\ln L - \Omega(R)} \ .$$

Proof (of Theorem 6, sketch). The theorem follows by considering each of the new situations that occur when honest parties loose and regain some of their resources. We sketch these considerations below.

If an alert party briefly loses access to its random oracle, it will keep the synchronized status, and start caching all network messages and advancing its local clock "blindly" by 1 slot per tick of $\mathcal{G}_{\text{ImperfLClock}}$. Hence, it will not violate the Skew_{Δ} invariant until it first reaches a synchronization slot, and is able to become alert again immediately upon regaining access to $\mathcal{G}_{\mathrm{RO}}$. However, once it reaches a synchronization slot, it declares itself desynchronized, hence not affecting Skew_{Δ} anymore. Similarly, if an honest party loses access to its clock or its network, it immediately becomes desynchronized.

Desynchronized parties maintain this status until they regain all resources, at which point they run the joining procedure $\mathsf{JoinProc}$ analyzed in Section 6.5, just like newly joining parties. The claims (b) and (c) of Lemma 7 guarantee that upon completion of this procedure, when the party declares itself synchronized again, it will indeed satisfy the invariant Skew_{Δ} except with a negligible error probability: this follows again by invoking Fact 1 in the same way as done for synchronized parties in Lemma 4 based on the claims (i) and (ii) from its proof.

Finally, note that the proof of Theorem 1 relies on the martingale analysis from Appendix F, which was designed in [4] exactly for the purpose of carrying over to the dynamic availability setting that allows the environment to adjust the stake ratios of alert and active parties adaptively during the execution of the protocol. □

K Connection between Logical Time and Nominal Time

We first state and prove the general lemma:

Lemma 8 (Nominal vs. logical time growth). *Consider an execution of the full protocol Ouroboros-Chronos in the dynamic-availability setting, let* P *be a party that is synchronized between (and including) slots* $\mathtt{sl}$ *and* $\mathtt{sl}'$*, let* t *and* t' *be the nominal times when* P *enters slot* $\mathtt{sl}$ *and* $\mathtt{sl}'$ *for the first time, respectively. Denote by* $\delta\mathtt{sl}$ *and* δt *the respective differences* $|\mathtt{sl}' - \mathtt{sl}|$ *and* $|t' - t|$*. Define the quantity* $\tau_{\mathrm{TG}} \triangleq 1-(96\Delta + R\epsilon\beta f)/(48R)$*. Then, under the assumptions of Theorem 6, we have* $\delta\mathtt{sl} \geq \tau_{\mathrm{TG}} \cdot \delta t$ *whenever* $\delta t \geq 48\widetilde{\Delta}/(\epsilon\beta f)$ *(where* $\widetilde{\Delta} = 2\Delta$*).*

Proof (of Lemma 8). The lemma follows directly from Lemma 5 carried over to the dynamic-availability setting. In particular, the skew that the adversary can apply in every sequence when nominal time increases by R is at most 2Δ since no more synchronization slots can occur where the synchronized parties adjust their local time-stamps. In between, all alert parties proceed at least at the baseline speed, which is used to define the nominal time. Given that the interval under consideration could start right at a synchronization slot of alert party P, we need to incorporate an additional offset of 2Δ giving a total shift of at most $2\Delta \cdot \delta t/R + 2\Delta$. Relative to δt, this shift can be expressed as $(2\Delta/R + x)\delta t$ for some $x > 0$ as long as $\delta t \geq 2\Delta/x$. For the sake of concreteness, we pick $x = (\epsilon\beta f)/48$ to obtain the lower bound on $\delta t \geq 48\widetilde{\Delta}/(\epsilon\beta f)$ that aligns with the bound in Corollary 6 (where $\widetilde{\Delta} = 2\Delta$), finally yielding the τ_{TG} of the statement. Note that the coefficient tends to the value $(1 - x)$ for increasing epoch lengths $R \geq 144\widetilde{\Delta}/(\epsilon\beta f)$. □

Its main application is to the following corollary:

Corollary 7. *Consider the event that the execution of Ouroboros Chronos under the assumptions of Theorem 6 does not violate property* CG *with parameters* $\tau \in (0, 1], s \in \mathbb{N}$*. Let* $\tau_{\mathsf{CG,glob}} \triangleq \tau \cdot \tau_{\mathrm{TG}}$*. Consider a chain* $\mathcal{C}$ *possessed by an alert party at a point in the execution where the party is at an onset of a (local) round and where the nominal time is* t*. Let further* t_1, t_2*, and* δt *be such that* $t_1 + \delta t \leq t_2 \leq t$*. Let* $\mathtt{sl}_1$ *and* $\mathtt{sl}_2$ *be the last slot numbers that* P *reported in the execution when nominal time was* t_1 *(resp.* t_2*) Then it must hold that* $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \geq \tau_{\mathsf{CG,glob}} \cdot \delta t$ *whenever* $\delta t \geq \max\{s/\tau, 48\widetilde{\Delta}\}$*.*

Proof (of Corollary 7). By the previous Lemma, if the nominal time increases by δt, then in the view of alert party P, at least $\tau_{\mathrm{TG}} \cdot \delta t \geq s$ slots were reported (immediate by the property that all parties run at least at the baseline speed). Thus, by chain growth as of Definition 6.2, the increase in blocks between the reported logical slots $\mathtt{sl}_1$ and $\mathtt{sl}_2$ must be $\tau_{\mathrm{TG}} \cdot \delta t \cdot \tau = (\tau_{\mathrm{TG}} \cdot \tau) \cdot \delta t$. □

L UC Realization

We are now ready to state the UC theorem in full detail, including all the parameters:

Theorem 7. *Let* k *be the common-prefix parameter,* R *the epoch-length parameter (constrained as required by Theorem 5) and* $\Delta = \Delta_{\mathrm{net}} + \Delta_{\mathrm{clock}}$*. Let* τ_{CG} *be the chain growth coefficient as of Theorem 1, let* $\tau_{\mathsf{CG,glob}}$ *be the derived (nominal time) chain-growth coefficient as of Corollary 7, and let* μ *be the chain quality coefficient as of Theorem 1. Under the constraints*[22] *of Theorem 6, the protocol Ouroboros Chronos realizes the ledger functionality, i.e., there exists a simulator that simulates the protocol execution in the ideal world perfectly except with negligible probability in the parameter* k *for* $R \geq \omega(\log k)$*, for the ledger parameters*

$$\mathtt{windowSize} = k; \qquad \mathtt{Delay} = t_{\mathrm{join}}; \qquad \mathtt{Delay}_{\mathtt{tx}} = 2\Delta;$$

$$\mathtt{maxTime}_{\mathtt{window}} \geq \frac{\mathtt{windowSize}}{\tau_{\mathsf{CG}} \cdot \tau_{\mathsf{CG,glob}}}; \qquad \mathtt{advBlcks}_{\mathtt{window}} \geq (1-\mu)\mathtt{windowSize},$$

and the clock-parameters

$$\mathsf{shiftLB} = -2\Delta; \qquad \mathsf{shiftUB} = \Delta; \qquad R_L = R$$

$$\mathtt{timeSlack}_{\mathrm{total}} = 2\Delta; \qquad \mathtt{timeSlack}_{\mathrm{ep}} = \Delta,$$

and where the algorithms Blockify, Validate, and predict-time are instantiated as stated in Section M.2.

[22] Note that while we express the theorem as a constrained statement, it is possible to express the constraint as a (hybrid) functionality wrapper in the real world that enforces the constraints and leaves the environment unconstrained. For more details we refer to [5] and [4].

Proof. For the simulation part, we observe that compared to the simulator of the protocol Ouroboros Genesis in [4], the protocol Ouroboros Chronos does only introduce code such that the entire process of honest parties can still be simulated perfectly inside the simulator. All added procedures can be emulated by the simulator who is emulating the honest parties code, extracts their states and times, and sets the ledger parameter appropriately. We give the simulator in Section L.1.

We thus define the bad events that any constraints imposed by the above choice of parameters would prohibit the simulator in correctly setting the ledger state or the time (the events are called BAD-CP, BAD-CQ, BAD-CG, and BAD-TIME-RANGE in the simulation).

We first observe that violating the state parameters implies either violation of common-prefix, chain quality or chain-growth in the execution (i.e., we have a identical-until-bad simulation). Furthermore, the protocol is designed such that the activation pattern is still predictable by an efficiently computable predicate predict-time, since for the MAINTAIN-LEDGER it is by design fixed how many inputs are need to reach the FinishRound statement in the code.

Finally, the weak liveness of transactions holds since whenever a transaction is in the network at least $\Delta < t_{\text{join}}$ rounds, it will eventually be included in the next high-quality block (i.e., a block with hFlag = 1) (and in the real-world by any alert party proposing a block) as long as the transaction is still valid. Considering that the analysis conditions no collisions among random oracle outputs, we obtain an upper bound of $\exp(-\Omega(\kappa)) + \exp(\ln \mathsf{poly}(\kappa) - \Omega(k)) + \exp(\ln \mathsf{poly}(\kappa) - \Omega(R))$, where $\mathsf{poly}(\kappa)$ denotes the polynomial upper bound on the runtime of $\mathcal{Z}$ measured with respect to the security parameter κ. (Note that in particular, the parameters L and Q of the security bound can simply be upper bounded by this polynomial.) Combining this with the connection established in 6.7 settles that the chosen parameters do not impose a restriction on the ideal-world adversary except with negligible probability.

We next turn to the export-clock extension parameters. First, setting $R_L = R$ is identical to the real world. Second, the simulator is given enough activation in every round s.t. whenever a synchronized party reaches a synchronization slot $i \cdot R = i \cdot R_L$, it can input a shift value. Next, by Lemma 5 it is clear that $\mathsf{shiftLB} = -2\Delta$ and $\mathtt{timeSlack}_{\text{ep}} = \Delta$ by Lemma 4. What remains to show is that we can essentially bound the (1) overall skew between two adjacent epochs by 2Δ and (2) that no party ever shifts its clock by more than Δ. Both claims follow from directly from strengthening Lemma 5 as done in Lemma 9. By the preceeding analysis, the probability that any constraint is violated, and thus BAD-TIME-RANGE is triggered, is also in this case bounded by a negligible function of the above form for our choice of parameters. □

Lemma 9. *Consider the same setting as in Lemma 5. Let* sl *be the synchronization slot of epoch* ep *and let* $\mathcal{S}_i^{\mathsf{P}}$ *be the set of beacons of an alert party* P *that is used for synchronization. Furthermore, assume that* $\mathit{Skew}_\Delta[\mathtt{sl}]$ *is not violated in this execution. Then it holds that*

1. *The shift* shift *party* P *computes is upper bounded by the maximal recommendation* $\mathsf{recom}(\mathit{SB})$, $\mathit{SB} \in \mathcal{S}_i^{\mathsf{P}}$ *for which* $\mathsf{slotnum}(\mathit{SB}) = s$ *for some* $s \in [(\mathsf{ep}-1)R, \ldots, (\mathsf{ep}-1)R + R/6]$ *and for which that it was created by an alert party* P' *in slot* s*. By the Lemma assumption, this is upper bounded by* Δ*.*
2. *After the shift, party* P *reports a time-stamp that is at most* 2Δ *off of any alert party's reported time stamp.*

Proof (Sketch). The first part of the claim follows from the observation that $\mathsf{Timestamp}(\mathtt{SB})$ recorded by P is received after P' created the beacon. Consider the term $\mathsf{slotnum}(\mathtt{SB}) - \mathsf{Timestamp}(\mathtt{SB})$ that this beacon contributes to the overall recommendation. If at the point the beacon was created the creator reported slot s, and party P' local timestamp (used to measure arrival times in epoch ep) differs by d to the creator's timestamp, then the recommendation is upper bounded by d, as delays can make the term only smaller and subsequent clock drifts do not influence the reported arrival times (as a party fetches once per observed round). The limited skew $\mathsf{Skew}_\Delta[s]$ at that slot s further says that $d \leq \Delta$. Since the alertly generated beacons are in the majority as argued in Lemma 5 the median is bounded by Δ as well.

To prove the second item, note that it is sufficient to show that the difference in reported time stamp of party P in slot sl after the synchronization procedure to any alert party P' that has not yet made the clock adjustment for synchronization slot sl is at most 2Δ and cannot be further beyond that e.g., by doing some additional drift. First by 1., the party's shift is upper bounded by Δ. Since by $\mathsf{Skew}_\Delta[\mathtt{sl}]$, any other alert party

P' that has not yet passed synchronization slot $\mathtt{sl}$, will report a time stamp $\mathtt{sl}' \geq \mathtt{sl} - \Delta$. Finally, similarly to the maximal shift, the lower bound on the shift can be obtained by examining the recommendation computed by alertly generated beacons $\mathsf{slotnum}(\mathtt{SB}) - \mathsf{Timestamp}(\mathtt{SB})$. Analogous to the above case, if at the point of the execution the beacon was created the creator reported slot s, party P' local timestamp (used to measure arrival times in epoch $\mathtt{ep}$) differs by at most d to the creator's timestamp, then the recommendation is lower bounded by $d - \Delta$. Similar to the arguments in Lemma 4, this worst case cannot be further amplified, because the drift allowed by $\mathcal{G}_{\text{IMPERFLCLOCK}}$ is an absolute additive constant over the entire execution (and therefore, either the true shift is in fact lower than the worst case and later maximized, or is maximal and cannot be further increased). Again, assuming $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ is not violated, this is lower bounded by -2Δ. Since any alert party that has not yet made its adjustment reports a local time stamp larger equal to $\mathtt{sl} - \Delta$, the adjustment of P in this round will not increase the distance to any alert party that has not yet passed the synchronization slot to more than 2Δ. As above, this invariant remains true until everyone has made the adjustments (and then the stronger guarantees proven in previous sections apply). □

L.1 Simulator

Below we present the simulator used in the proof that the UC implementation of Ouroboros Chronos securely realizes the ledger functionality $\mathcal{G}_{\text{LEDGER}}$ with the clock extension. The simulator shares a lot of similarities with the simulator provided in [4] and is given below for the sake of concreteness.

Simulator $\mathcal{S}_{\text{ledg}}$ (Part 1 - Main Structure)

Overview:

- The simulator internally emulates all local UC functionalities by running the code (and keeping the state) of $\mathcal{F}_{\mathsf{VRF}}$, $\mathcal{F}_{\mathsf{KES}}$, $\mathcal{F}_{\mathsf{INIT}}^{\Delta_{\text{net}}}$, $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$, $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$, and $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$, where $\text{ref}_{\mathsf{P}} := \langle v_{\mathsf{P}}^{\text{vrf}}, v_{\mathsf{P}}^{\text{kes}} \rangle$ identifies the address of party (i.e., ITI) P (and is given to the ledger when an ITI is registered).
- The simulator mimics the execution of Ouroboros Chronos for each honest party P (including their state and the interaction with the hybrids).
- The simulator emulates a view towards the adversary $\mathcal{A}$ in a black-box way, i.e., by internally running adversary $\mathcal{A}$ and simulating his interaction with the protocol (and hybrids) as detailed below for each hybrid. To simplify the description, we assume $\mathcal{A}$ does not violate the theorem assumptions (as they are enforced by a wrapper $\mathcal{W}_{\text{OG}}^{PoS}(\cdot)$as in [4]).
- For global functionalities, the simulator simply relays the messages sent from $\mathcal{A}$ to the global functionalities (and returns the generated replies). Recall that the ideal world consists of the dummy parties, the ledger functionality, the clock, and the random oracle.

Party sets:

- As defined in the main body of this paper, honest parties are categorized. We denote $\mathcal{S}_{\text{alert}}$ the alert parties (synchronized and executing the protocol) and use $\mathcal{S}_{\text{syncStalled}}$ shorthand for parties that are synchronized (and hence time aware and online) but stalled. Finally, we denote by $\mathcal{P}_{DS}$ all honest but de-synchronized parties (both operational or stalled).
- For each registered honest party, the simulator maintains the local state containing in particular the local chain $\mathcal{C}_{\mathsf{loc}}^{(\mathsf{P})}$, the time t_{on} it remembers when last being online. For each party P, the simulator maintains the objective (or nominal) round number $\tau_L' := \tau_L + \delta_{\mathsf{P}}$, i.e., decomposed into the baseline time and the party-specific offset δ_{P} bounded by Δ_{clock}. For each party P, the simulator stores the reported time $\mathtt{time}_{\mathsf{P}} = (e, \mathtt{localTime})$, and the flags $\mathsf{updateState}_{\mathsf{P},\tau_L'}$, $\mathsf{updateTime}_{\mathsf{P},\tau_L'}$, and $\mathsf{updateInitTime}_{\mathsf{P},\tau_L'}$ (initially false) to remember whether this party has completed its core maintenance tasks in its nominal round τ_L' to update the state and its time (where the initial time for each party is a separate case), respectively. Note that an registered party is registered with all its local hybrids.
- Upon any activation, the simulator will query the current party set from the ledger, the clock, and the random oracle to evaluate in which category an honest party belongs to. If a new honest party is registered to the

ledger, it runs the initialization procedure for this party in each new round until the party is initialized ($\mathsf{P}.\texttt{isInit}$ becomes true).
- We assume that the simulator queries upon any activation for the sequence $\boldsymbol{\mathcal{I}}_H^T$, and the current (baseline) time τ_L from the ledger. We note that the simulator is capable of determining $\mathsf{predict\text{-}time}(\cdot)$ of $\mathcal{G}_{\text{LEDGER}}$ and hence the nominal time of the execution at any point. If the baseline advances, the simulator adjusts all offsets δ_{P} accordingly (which it can do exactly as in $\mathcal{G}_{\text{IMPERFLCLOCK}}$).

Messages involving the Clock:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ from $\mathcal{G}_{\text{IMPERFLCLOCK}}$, if P is an honest registered party, then remember that this party has received such a clock update (and the environment gets an activation). Otherwise, send $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-PUSH}, \text{sid}_C, \mathsf{P})$ from $\mathcal{A}$, remember that P is allowed to locally advance in this round if allowed by $\mathcal{G}_{\text{IMPERFLCLOCK}}$ (restricted by the maximal drift Δ_{clock}) and adjust δ_{P}.

Messages from the Ledger:

- Upon receiving $(\mathsf{Respond}, (\mathsf{start}, sid))$ from $\mathcal{G}_{\text{LEDGER}}$, send $(\mathsf{Respond}, (\mathsf{DefineOffset}, sid))$ in the name of $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$ to the adversary. Upon receiving the response $(\mathsf{DefineOffset}, sid, o_1, \ldots, o_n)$, $o_i \in [0, \ldots, \Delta]$, store the values and relay the answer to the simulated instance of $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$.
- Upon receiving $(\text{SUBMIT}, \texttt{BTX})$ from $\mathcal{G}_{\text{LEDGER}}$ where $\texttt{BTX} := (\texttt{tx}, \text{txid}, \tau, \mathsf{P})$ forward $(\text{MULTICAST}, \text{sid}, \texttt{tx})$ to the simulated network $\mathcal{F}_{\text{N-MC}}^{\text{tx}}$ in the name of P. Output the answer of $\mathcal{F}_{\text{N-MC}}$ to the adversary.
- Upon receiving $(\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ from $\mathcal{G}_{\text{LEDGER}}$, extract from $\boldsymbol{\mathcal{I}}_H^T$ the party P that issued this query. If P has already completed its round-task, then ignore this request. Otherwise, execute $\text{SIMULATEMAINTENANCE}(\mathsf{P}, \tau_L)$.

Simulator $\mathcal{S}_{\text{ledg}}$ (Part 2 - Black-Box Interaction)

Simulation of Functionality $\mathcal{F}_{\mathsf{INIT}}$ towards $\mathcal{A}$:

- The simulator relays back and forth the communication between the (internally emulated) $\mathcal{F}_{\mathsf{INIT}}^{\Delta_{\text{net}}}$ functionality and the adversary $\mathcal{A}$ acting on behalf of a corrupted party.
- If at time $\tau_L = 0$, a corrupted party $\mathsf{P} \in \mathcal{S}_{\text{initStake}}$ registers via $(\mathsf{ver\_keys}, sid, \mathsf{P}, v_{\mathsf{P}}^{\text{vrf}}, v_{\mathsf{P}}^{\text{kes}})$ to $\mathcal{F}_{\mathsf{INIT}}$, then input $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}_{\text{LEDGER}}$ on behalf of P.

Simulation of the Functionalities $\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{VRF}}$ towards $\mathcal{A}$:

- The simulator relays back and forth the communication between the (internally emulated) hybrids and the adversary $\mathcal{A}$ (either direct communication, communication to $\mathcal{A}$ caused by emulating the actions of honest parties, or communication of $\mathcal{A}$ on behalf of a corrupted party).

Simulation of the Network $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ (over which chains are sent) towards $\mathcal{A}$:

- Upon receiving $(\text{MULTICAST}, \text{sid}, (\mathcal{C}_{i_1}, U_{i_1}), \ldots, (\mathcal{C}_{i_\ell}, U_{i_\ell}))$ with a list of chains and corresponding parties from $\mathcal{A}$ (or on behalf some *corrupted* $P \in \mathcal{P}_{net}$), then do the following:
  1. Relay this input to the simulate network functionality and record its response to $\mathcal{A}$.
  2. Execute $\text{EXTENDLEDGERSTATE}(\tau_L)$
  3. Provide $\mathcal{A}$ with the recorded output of the simulated network.
- Upon receiving $(\text{MULTICAST}, \text{sid}, \mathcal{C})$ from $\mathcal{A}$ on behalf of some *corrupted* party P, then do the following:
  1. Relay this input to the simulate network functionality and record its response to $\mathcal{A}$.
  2. Execute $\text{EXTENDLEDGERSTATE}(\tau_L)$
  3. Provide $\mathcal{A}$ with the recorded output of the simulated network.
- Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ forward the request to the simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from $\mathcal{A}$: Forward the request to the simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and record the answer to $\mathcal{A}$. Before giving this answer to $\mathcal{A}$, query the ledger state $\texttt{state}$ and execute $\text{ADJUSTVIEW}(\texttt{state}, \tau_L)$.

– Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from $\mathcal{A}$: Forward the request to the simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and record the answer to $\mathcal{A}$. Before giving this answer to $\mathcal{A}$, query the ledger state `state` and execute $\text{AdjustView}(\texttt{state}, \tau_L)$.

Simulation of the Network $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ (over which transactions are sent) towards $\mathcal{A}$:

– Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from $\mathcal{A}$ with list a transaction m from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$, then do the following:
  1. Submit the transaction to the ledger on behalf of this corrupted party, and receive for the transaction id txid.
  2. Forward the request to the internally simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$, which replies for each message with a message-ID mid
  3. Remember the association between mid and the corresponding txid
  4. Provide $\mathcal{A}$ with whatever the network outputs.
– Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ forward the request to the simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ and return whatever is returned to $\mathcal{A}$.
– Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ and return whatever is returned to $\mathcal{A}$.
– Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ and return whatever is returned to $\mathcal{A}$.

Simulation of the Network $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$ (over which beacons are sent) towards $\mathcal{A}$:

– Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from $\mathcal{A}$ with a beacon m from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$, then do the following:
  1. Forward the request to the internally simulated $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$, which replies for each message with a message-ID mid
  2. Remember the association between each mid and the corresponding beacon.
  3. Provide $\mathcal{A}$ with whatever the network outputs.
– Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ behave analogously to above for $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$.
– Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from $\mathcal{A}$ behave analogously to above for $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$.
– Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from $\mathcal{A}$ behave analogously to above for $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$.

Simulator $\mathcal{S}_{\text{ledg}}$ (Part 3 - Internal Procedures)

procedure $\text{SimulateMaintenance}(\mathsf{P}, \tau_L)$

Simulate the (in the UC interruptible manner) the maintenance procedure of party P as in the protocol in round $\tau_L' = \tau_L + \delta_{\mathsf{P}}$ when the party reports localtime $\mathsf{P}.\texttt{localTime}$, i.e., run $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \text{sid}, k, s, R, f)$ for this simulated party.

if party P gives up activation then **then**
  if party P has completed $\mathsf{JoinProc}(\cdot)$ and $\mathsf{updateInitTime}_{\mathsf{P},\tau_L'}$ is false **then**
    Execute $\text{AdjustTime}(\tau_L)$ and then set $\mathsf{updateInitTime}_{\mathsf{P},\tau_L'} \leftarrow \mathsf{true}$.
  end if
  if party P has reached the instruction $\mathsf{SelectChain}(\cdot)$ and $\mathsf{updateState}_{\mathsf{P},\tau_L'}$ is false **then**
    Execute $\text{ExtendLedgerState}(\tau_L)$ and then set $\mathsf{updateState}_{\mathsf{P},\tau_L'} \leftarrow \mathsf{true}$.
  end if
  if party P has reached the instruction $\mathsf{SyncProc}(\cdot)$ and $\mathsf{updateTime}_{\mathsf{P},\tau_L'}$ is false **then**
    Execute $\text{AdjustTime}(\tau_L)$ and then set $\mathsf{updateTime}_{\mathsf{P},\tau_L'} \leftarrow \mathsf{true}$.
  end if
  if party P has reached the instruction $\mathsf{FinishRound}(\mathsf{P})$ in round τ_L' **then**
    Send $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$ if $\mathcal{S}_{\text{ledg}}$ has received such an input in round τ_L'
  end if
  Return activation to $\mathcal{A}$
end if

end procedure

procedure ExtendLedgerState(τ_L)
  for each synchronized party $\mathsf{P} \in \mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ of round $\texttt{localTime}$ **do**
    Let $\mathcal{C}_{\text{loc}}^{(\mathsf{P})}$ be the party's currently stored local chain.
    // Note: In the following the internally simulated party state is not changed
    Determine the number of fetches $\rho^{(\mathsf{P})} \in \{0, 1\}$ this party is still going to make in this round $\tau_L' := \tau_L + \delta_{\mathsf{P}}$.
    If $\rho^{(\mathsf{P})} > 0$ then let $\mathcal{C}_1^{(\mathsf{P})}, \ldots, \mathcal{C}_k^{(\mathsf{P})}$ be the chains contained in the receiver buffer $\boldsymbol{M}^{(\mathsf{P})}$ of $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ with delay at most $\rho^{(\mathsf{P})}$.
    Re-evaluate $\mathcal{C}_{\mathsf{P}} \leftarrow \mathsf{SelectChain}()$ using the additoinal chains as well and let this resulting chain's encoded state be $\texttt{st}_{\mathsf{P}}$.
  end for
  Let $\texttt{st}$ be the longest state among all such states $\texttt{st}_{\mathsf{P}}$, $\mathsf{P} \in \mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ from above.
  Compare $\texttt{st}^{\lceil k}$ with the current state $\texttt{state}$ of the ledger
  if $|\texttt{state}| > |\texttt{st}^{\lceil k}|$ **then** // Only pointers need adjustments
    Execute AdjustView($\texttt{state}$)
  end if
  if $\texttt{state}$ is not a prefix of $\texttt{st}^{\lceil k}$ **then** // Simulation fails
    Abort simulation: consistency violation among synchronized parties. // Event $\mathsf{BAD\text{-}CP}_k$
  end if
  Define the difference diff to be the block sequence s.t. $\texttt{state}||\mathsf{diff} = \texttt{st}^{\lceil k}$.
  Parse $\mathsf{diff} := \mathsf{diff}_1||\ldots||\mathsf{diff}_n$.
  for $j = 1$ to n **do**
    Map each transaction $\texttt{tx}$ in this block to its unique transaction ID txid. If a transaction does not yet have a txid, then submit it to the ledger first and receive the corresponding txid from $\mathcal{G}_{\text{ledger}}$
    Let $\mathsf{list}_j = (\text{txid}_{j,1}, \ldots, \text{txid}_{j,\ell_j})$ be the corresponding list for this block diff_j
    if coinbase $\text{txid}_{j,1}$ specifies a party honest at block creation time **then**
      $\text{hFlag}_j \leftarrow 1$
    else
      $\text{hFlag}_j \leftarrow 0$
    end if
    Output $(\text{NEXT-BLOCK}, \text{hFlag}_j, \mathsf{list}_j)$ to $\mathcal{G}_{\text{ledger}}$ (receiving $(\text{NEXT-BLOCK}, ok)$ as an immediate answer)
  end for
  if Fraction of blocks with hFlag $= 0$ in the recent k blocks $> 1 - \mu$ **then**
    Abort simulation: chain quality violation. // Event $\mathsf{BAD\text{-}CQ}_{\mu,k}$
  else if State increases less than k blocks during the last $\frac{k}{\tau_{\mathsf{CG}}}$ rounds **then**
    Abort simulation: chain growth violation. // Event $\mathsf{BAD\text{-}CG}_{\tau_{\mathsf{CG}},k/\tau_{\mathsf{CG}}}$
  end if
  // If no bad event occurs, we can adjust pointers into this new state.
  Execute AdjustView($\texttt{state}||\mathsf{diff}$)
end procedure

procedure AdjustTime(P, τ_L)
  Let $\tau_L' := \tau_L + \delta_{\mathsf{P}}$ be the current (objective) round of this party.
  if P completed $\mathsf{JoinProc}$ in this round τ_L' **then**
    // Note that this party is about to become synchronized and to report time.
    Take the simulated timestamp $\texttt{time}_{\mathsf{P}}$ and send $(\text{SET-TIME}, \text{sid}, \mathsf{P}, \texttt{time}_{\mathsf{P}})$ to $\mathcal{G}_{\text{ledger}}$.
  end if
  if $\tau_L' = 0$ and P is an initial stakeholder U_i **then**
    Send $(\text{APPLY-SHIFT}, \text{sid}, (U_i, -o_i))$ to $\mathcal{G}_{\text{ledger}}$, where the o_i are the initial offsets by $\mathcal{A}$ as recorded above.
  else if $\mathsf{P}.\texttt{localTime} \bmod R = 0$ **then**
    Take the simulated timestamp $\texttt{time}_{\mathsf{P}}$ and the simulated shift shift of this party.
    if The range of timestamps of parties in $\mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ is invalid **then**

Abort simulation: time-range violation. // Event BAD-TIME-RANGE
else
Send $(\text{APPLY-SHIFT}, \text{sid}, (\mathsf{P}, \mathsf{shift}))$ to $\mathcal{G}_{\text{LEDGER}}$.
end if
end if
end procedure

M Glossary: Protocol Variables and Ledger Parameters

M.1 Main State Variables of Ouroboros Chronos

The variables defining the state of a protocol participant are summarized below.

Variable	Description
`localTime`,`sl`	The party's current time-stamp. In the staking context we call the time slot.
`ep`	The epoch that `sl` belongs to.
$\mathcal{C}_{\mathsf{loc}}$	The local chain the party adopts based on which it does staking and exports the ledger state.
`isInit`	A variable to keep track of whether initialization is complete.
t_{work}	A value to steer when the party executes the staking procedure for the next time.
`buffer`	The buffer of transactions.
`futureChains`	A buffer to store chains that are not yet processed, for example because they contain blocks that belong to the logical future of this party.
$\mathsf{Timestamp}_{SB}(\cdot)$	A map that assigns to each synchronization beacon a pair (a, b), where a is a numerical value (the arrival time) and b is an indication of whether a is final or not.
$\mathsf{Timestamp}(\cdot)$	Shorthand for the first (and numerical) element of the pair $\mathsf{Timestamp}_{SB}(\cdot)$.
`lastTick`	The last tick received from $\mathcal{G}_{\text{PERFLCLOCK}}$. Used to infer when a round change occurs.
`isSync`	A party stores its synchronization status, as it can infer when its time and state become reliable.
$\mathsf{EpochUpdate}(\cdot)$	An function table to remember which clock adjustments have been done already. Used to update beacon arrival times.
`fetchCompleted`	A variable to store whether the round messages have been fetched.
`lastTimeAlert`	The local time stamp the party was alert the last time. Used for rejoining if the party was only stalled.
$T_{\mathsf{P}}^{\mathsf{ep}}, T_{p}^{\mathsf{ep},\mathsf{bc}}$	The thresholds of this party to evaluate slot leadership (and beacon production and validity) in (current) epoch `ep`.
$v_{\mathsf{P}}^{\text{vrf}}$, $v_{\mathsf{P}}^{\text{kes}}$	The public keys of this party to interact with $\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{VRF}}$.

Fig. 4. Overview of the main state variables of Ouroboros Chronos.

M.2 Main Ledger Elements

The state variables of the new ledger functionality are given below.

Core Ledger Parameter	**Description**
`windowSize`	The window size (number of blocks) of the sliding window. In the realization statement, it is typically set to the common-prefix parameter.
Validate	Decides on the validity of a transaction with respect to the current state. Used to clean the buffer of transactions. If the protocol fixes a validation predicate, say $\mathsf{ValidTx}_{\mathsf{OC}}$, then the realization statement holds with $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) := \mathsf{ValidTx}_{\mathsf{OC}}(\texttt{tx}, \texttt{state})$.
Blockify	The function to format the ledger state output. If the protocol fixes a particular function, say $\mathsf{blockify}_{\mathsf{OC}}$, the ledger will use the same in the realization proof.
predict-time	The function to predict the real-world time advancement. Ouroboros Chronos has a predictable time-advancement $\mathsf{predict\text{-}time}_{OC}$ as it can inferred by design when the protocol will call FinishRound in each round, when given a fixed number of activations that depends on the local time-stamp of this party.
`Delay`	A general delay parameter for the time it takes for a newly joining (after the onset of the computation) miner to become synchronized. In this paper, it corresponds to the duration of the joining procedure.

Policy Parameter (ExtendPolicy)	**Description**
$\texttt{maxTime}_{\texttt{window}}$	Minimal Growth: In $\texttt{maxTime}_{\texttt{window}}$ rounds at least `windowSize` blocks have to be inserted into the ledger state. The value in the realization proof will depend on the chain-growth property.
$\texttt{advBlcks}_{\texttt{window}}$	A limit $\texttt{advBlcks}_{\texttt{window}}$ of adversarial blocks (i.e., contributed blocks that do not need to employ higher standards) in each window of `windowSize` state blocks. This ensures a minimal fraction of blocks that contain all old and valid transactions. The value in the realization proof will depend on the chain-quality property.
$\texttt{Delay}_{\texttt{tx}}$	An extra parameter to define when a transaction is old. In this work, this will be much less than `Delay` as it will only depend on the network delay.

Export-Time Parameter	**Description**
$\texttt{time}_{\mathsf{P}}$	A variable that will represent the (idealized) clock value that the party reports as its local time. A party will export pairs (e, t), where t is the current local time, and e is the epoch.
shiftLB, shiftUB	Limits on the shift values an adversary can impose at epoch boundaries
R_L	The parameter characterizing epoch boundaries: if a party's timestamp (e, t) is such that $t = iR_L + 1$ (for the first time), then the party moves to the next epoch.
$\texttt{timeSlack}_{\text{total}}$	An upper bound between t and t' of two synchronized parties P and P' reporting (e, t) and (e', t') as their respective time $\texttt{time}_{\mathsf{P}}$ and $\texttt{time}_{\mathsf{P}'}$, respectively.
$\texttt{timeSlack}_{\text{ep}}$	An upper bound between t and t' of two synchronized parties P and P' reporting (e, t) and (e', t') as their respective time $\texttt{time}_{\mathsf{P}}$ and $\texttt{time}_{\mathsf{P}'}$, respectively whenever $e = e'$

Fig. 5. Overview of main ledger elements such as parameters and state variables. As in [5, Definition 2], we always assume that $\mathsf{blockify}_{\mathsf{OC}}$ and $\mathsf{ValidTx}_{\mathsf{OC}}$ do not disqualify each other.