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

Christian Badertscher⋆, Peter Gaži⋆⋆, Aggelos Kiayias⋆⋆⋆, Alexander Russell†, and Vassilis Zikas‡

July 19, 2019

Abstract. Proof-of-stake (PoS) has been shown to be a suitable replacement—in many respects—for the expensive proof-of-work mechanism introduced by the Bitcoin protocol. Nevertheless, one common and seemingly intrinsic shortcoming of all existing PoS blockchains in the permissionless "dynamic availability" setting introduced by Badertscher *et al.* [CCS 2018], where parties come and go without warning, is that they require explicit use of a common notion of time among the participants, i.e., a "global" clock that provides the correct time on demand.
We design and analyze a PoS blockchain protocol that we prove UC-secure without assuming access to a global time functionality. Central to our construction is a novel clock synchronization mechanism that enables joining parties to adjust their local clocks correctly, relying only on knowledge of the genesis block and the assumption that their local, initially desynchronized clocks advance at approximately the same speed. This is particularly challenging as we work in the dynamic availability setting which addresses optimal resilience under arbitrary and potential adversarial participation patterns. As a corollary of our construction, we obtain a permissionless PoS implementation of a global clock that may be used whenever access to global time is a requirement in a higher level protocol.

1 Introduction

Synchrony is one of the most important concepts in the theory and practice of distributed computing. Distributed protocols in the synchronous model advance in rounds and typically it is guaranteed that no party advances to round $\rho+1$ before every party has finished round ρ. In every round, a party might receive messages sent to it in the previous round and send messages to other parties intended (and guaranteed) to be delivered by the beginning of the following round. This means that synchronous protocols allow the protocol participants to maintain an implicit notion of common time by simply considering the current round index as a common clock (initially set to 0 at the protocol's onset).

In contrast, asynchronous settings do not provide such a common notion of round: while one party might already have completed its protocol, having received and reacted to all messages, another party may still be stuck waiting for messages somewhere in the middle of its protocol execution. While removing synchrony assumptions is naturally attractive and important, the asynchronous model comes at a high cost: The only way to get the same type of output-correctness guarantees afforded by the synchronous model is for parties to explicitly enforce a round structure, by some implicit or explicit acknowledgement mechanism where parties only advance to their next round if every party acknowledges that it is finished with the current round (cf. [29]). In this case, it is clear that an adversary can stall the computation forever by withholding acknowledgements on behalf of corrupted parties.

Alternatively, one may relax the input/output requirements to cope with asynchrony by not waiting for an acknowledgement from everyone; rather, players advance as soon as "enough" messages have been received.

⋆ University of Edinburgh and IOHK. christian.badertscher@ed.ac.uk.
⋆⋆ IOHK, peter.gazi@iohk.io.
⋆⋆⋆ University of Edinburgh and IOHK. akiayias@inf.ed.ac.uk. Research partly supported by EU Project No. 780477, PRIVILEDGE.
† University of Connecticut. acr@cse.uconn.edu. This material is based upon work supported by the National Science Foundation under Grant No. 1717432.
‡ University of Edinburgh and IOHK. vassilis.zikas@ed.ac.uk.

As argued in [4, 5], under the mild assumption that messages of honest parties are eventually delivered, this model allows for termination even if the adversary may exclude from the computation the inputs of several honest parties—at least as many as the corrupted parties. Furthermore, the number of corruptions that can be tolerated by such a protocol is also decreased with respect to their synchronous counterparts. Despite its inherent limitations, this model, which is known as *asynchronous with eventual delivery* or *guaranteed termination*, has been extensively studied in the literature [14, 9, 10, 28, 15, 32, 16].

This state of affairs suggests two approaches to distributed protocol design: either work on the stringent, but presumably more realistic, asynchronous with eventual delivery model, or assume synchrony and rely on an underlying *synchronizer* subsystem—which is typically a sophisticated distributed protocol on its own—to provide the required round structure. Building such synchronizers from weaker assumptions has been extensively studied in secure and fault-tolerant distributed computing, computer science theory, and cryptography. In typical scenarios [18, 30, 23] 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 [18, 30, 23]. This bound can be improved to $t < n/2$ by use of existentially unforgeable digital signatures [30]. A number of follow up works have investigated clock synchronization in various settings [20, 19, 1, 40, 31, 36] and we refer the reader to [39] for a survey (albeit somewhat outdated).

As the above results suggest, the synchronization problem is well understood across most models of interest. However, as with many concepts in secure distributed computation, this situation changed with the introduction of the so-called *permissionless model*, popularized by the Bitcoin protocol [35] which utilizes Proof-of-Work (PoW) and in turn motivated the design of other blockchain protocols such as Ethereum [8], and the introduction of alternative mechanisms including Proof-of-Stake (PoS), as in Ouroboros [17, 26, 2], Algorand [22], and Snow White [6]), Proof-of-Space, as in Spacemint [37]), and Proof-of-Space-Time [34].[1] Informally, this model posits a different perspective which is incomparable to the ones studied before: there is a large population of potential participants and everyone can become part of the protocol execution[2], contribute to its security, and enjoy its guarantees, but not every party needs to be informed that a new party is joining or leaving or is required to be online all the time to follow the protocol. More specifically, at any given moment, there will be a (comparably) small set of users which will have access to network communication with bounded delays but that set will be continuously evolving and may be arbitrarily small compared to the total set of users. This comes in sharp contrast to the models previously studied, in which parties would be aware of a fairly good estimate of the number as well as all the identities of the parties running the protocol because membership to this set is typically controlled, something that has lead to its characterization as the *permissioned model*.

The cryptographic study of the permissionless model—mainly in the context of proving security of Bitcoin and developing next-generation blockchains—revealed that for many problems the techniques used in the permissioned setting do not translate directly to the permissionless one. Not surprisingly, clock synchronization is one of these challenges. Informally, this is explained by the fact that existing synchronization techniques rely on knowledge of the total number of parties in the system, an assumption which one cannot make in a truly permissionless model. We refer to Remark 1 for further discussion. In fact, to our knowledge, all existing rigorous cryptographic analyses of blockchains, e.g., [38, 21, 3, 26, 17, 2], assume, implicitly or explicitly, perfect or loose clock-synchronization. Furthermore, the reliance of PoS blockchains on a global clock appears quite essential, as it is precisely this assumption which joining parties use to prevent the ad-

---

[1] Arguably there exist different flavors of "permissionlessness", with PoW and PoS based blockchains exhibiting two distinct perspectives. Still all such systems exemplify similar "peer-to-peer" operational characteristics in the sense that are supposed to continue to be functional despite parties coming and going without warning, even at significant ($> 1/2$) ratios.

[2] Note how these approaches differ in how parties become part of the execution. In PoW the ability to contribute stems from computational power, in PoS a party first has to obtain stake from an existing stakeholder. In this sense, PoW is slightly "more permissionless".

versary from exploiting, for example, the longest chain rule to "fast forward" them to a fictitious blockchain extending into the future.

In this work we carry out a systematic cryptographic study of clock synchronization in the permissionless PoS setting. We devise a novel protocol based on Ouroboros Genesis [2] to demonstrate that it is possible to build a composable PoS-based blockchain that enables parties to implement and maintain a bounded-drift clock in the delayed-delivery network model [38, 21]. Our new blockchain protocol thereby avoids the dependency on an external service providing timestamps such as NTP [33], which constitutes a great improvement in resiliency compared to previous works. Moreover, our protocol can serve as a cryptographically secure clock synchronization module in any application.

Our construction assumes that the inaugural parties, that initially commence the protocol execution, have access to local clocks that report values (corresponding to local time) that might be off by a parameter Δ and which advance at (roughly) the same speed. This is similar to the timing model of Kalai et al. [24]. Our protocol then 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 that depends on the network delay) with the rest of the parties running the protocol, and remain synchronized for as long as they faithfully execute the protocol. We conduct our analysis in the UC framework (where we model certain setup functionalities as being globally accessible) [12, 13], which 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 bounded delivery networks.

As mentioned above, one feature of our solution to obtain clock synchronization, which is perhaps equally interesting as the solution itself, is that synchronization is achieved by means of a PoS-based blockchain protocol. We describe our construction building on previous results in PoS and specifically, [2], as this latter work enables to express formally and succinctly the exact level of permissionlessness called "dynamic availability" that our synchronization mechanism captures. To signify the connection, we adopt the name *Ouroboros Chronos*, or simply *Chronos*, to refer to our protocol. The Chronos protocol exports the clock it implements to whoever participates in it. The only thing that a party needs to do in order to connect to the common clock, is join the protocol 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 guaranteed to be in (loose, up to a small constant offset) synchronization with all honest parties running the protocol as long as honest stake majority is maintained.

Despite sharing similarities in structure to the previous work [17, 26, 2] the design and analysis of Ouroboros Chronos involves several new ideas, which are crucial in achieving synchronization in the permissionless setting. We discuss these ideas in details in Section 1.1 where we give an overview of the design. Furthermore, aside from its usefulness for synchronization, Chronos is, to our knowledge, the first blockchain whose security proof does not rely on parties being given (loosely) synchronized clocks as an external setup or on knowing an approximation of the total participation which would allow them to use techniques from the permissioned literature as explained in Remark 1 below.

Remark 1 (The Inapplicability of Common Synchronization Techniques). The main tool used by synchronizers in the permissioned setting is appropriate counting of messages in combination with signatures to thwart malicious behavior. Unfortunately, in the permissionless model with dynamic availability counting messages does not work, as the parties have no way of knowing how many, or which parties are present at any given time. This is also a major factor that distinguishes the Algorand [22] approach to decentralized consensus from that we follow here (which is based on Ouroboros Genesis [2]). 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 allows Algorand to achieve complete agreement on the whole blockchain after each block, and even employ techniques from the permissioned synchronizers literature to achieve clock synchronization. It is evident, however, that this is a stronger assumption than dynamic availability.

1.1 Overview of our Techniques

The synchronization process of Ouroboros Chronos starts with the joining parties listening on the network for some time, collecting broadcasted chains and following a Genesis-inspired "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[3] 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 although the above rule was designed (and proved effective) in [2] for the model in which all parties share the same global clock—and therefore timestamps are accurate—it still offers a useful (albeit in itself insufficient) guarantee when no global clocks are assumed: If honest parties use the above rule, then any honest party will end up with some blockchain that, although arbitrarily long, is at worst forking from any blockchain held by an honest and already synchronized party by a bounded number of blocks (equal to a security parameter) with overwhelming probability. Thus, the above joining process can be seen as a generalization of the bootstrapping process of Ouroboros Genesis [2] to eliminate the need to use of the global clock. More concretely, we prove here 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 to 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.

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 will use the difference between its current local time and the (local) time recorded when this chain (and other control information) was received to reset 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 its local clock to a time sufficiently close to that of an honest and already synchronized party. Such an updating process is, however, far from trivial to design, let alone prove secure. To see why, consider the following naïve solution: 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. Let us look at some implications of this adjustment. A first observation is that due to the (bounded but otherwise adversarially controlled) delay in the message delivery, a message received by a party might have been sent up to Δ rounds[4] before. Hence the time that the party will set its clock to might be up to Δ rounds far from the clock of the sender (at the point of update). This delay-induced imprecision is inherent in clock synchronization and we will settle with the property that after the adjustment the clocks only need to be loosely synchronized, i.e., clocks of honest parties might be far but only by a bounded amount, where the bound is known and depends on Δ. In fact, this relaxation is common and believed to be necessary even in the permissioned model, when honest clocks might report times that differ by Δ [30, 23].[5]

However, the above simple solution is problematic, even when there is no delay (i.e., messages sent in some round are guaranteed to be delivered in the next round), in which case we should in principle be able to obtain perfect synchronization. The reason is that 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 (a reasonably accurate) estimate of the true time of creation 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, cf. [21, 17], and thus the time inaccuracy of any individual block can be significant. Thus the above simplistic approach fails in multiple ways.

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*

[3] Recall that, in Ouroboros, as in most permissionless blockchains, parties add their local time (stamp) to blocks they broadcast.

[4] Note that we use the term "round" here to express a step in the protocol execution — message delivery may take multiple such rounds subject to the maximum delay of the network, Δ.

[5] The model from [30] 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. Also this solution has several issues. First, it is not scalable. But scalability is not a big limitation, as it can be achieved, using existing ideas, e.g., by using the protocol history as input to a VRF to identify eligible parties (or, 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 (even very small) drift, this cannot be the case. And even assuming identical speed clocks, in the dynamic availability setting, every party might eventually drop off and rejoin, which means that, due to the network delay (as discussed above) the honest parties will end up with a small drift on their local clocks. An alternative approach would be to use the average instead of the majority, but once again, even a single adversarial timestamp (and there will be many such) can throw off the average arbitrarily far. Thus we need to use a function that is more stable against extreme values. Such a function is the *median* of the received timestamps. 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.

Although the above idea brings us closer to our solution, it still has a shortcoming. If the adversary is able to serve to different parties inconsistent timestamps (on behalf of eligible corrupted synchronization-beacon parties) then he can possibly force an opposing clock adjustment between joining participants that will increase their clock drift well beyond the drift of any pair of already synchronized parties. To resolve this, we need to make sure 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 is in tandem with a PoS-based blockchain which solves consensus in the permissionless setting. The idea is to use the blockchain to agree on beacon-value indices to be used for recalibration. Once again, the property discussed at the beginning of the section will ensure that even newly joining parties will eventually fall in the set that has consensus on the common prefix and therefore can use the blockchain for recalibrating.

Our final solution uses the above techniques and we establish the above properties by a careful sequence of probabilistic arguments: In a nutshell, we will use the VRF to assign timestamping-beacon parties to slots/rounds 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. 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, the above process has many moving parts and this is only a high level idea of our construction. Its low level details and design choices are carefully defined and analyzed in Sections 3-4 which constitutes the main technical contribution of this paper.

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 [2] 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 interested reader to [2] for details on this mechanism and examples of the different types of guarantees offered by Ouroboros Genesis.

Synchrony and Time. As discussed above, a common assumption in the analysis of blockchain protocols in the permissionless model is the availability of a global clock that allows parties to acquire the current round index. This assumption was captured in [27, 3] by means of a (global) clock functionality which, in a nutshell, behaves as follows: it maintains a round index, i.e., a clock counter, which it reports to parties registered to it upon request. To ensure that the round only advances when all parties have been given a chance to complete their current-round instructions, the clock accepts a special command from any party that it interprets as "I am done with my current round". Once every (honest) party sends this command, the clock increases its round index.

The above clock makes the synchrony assumptions required by synchronous protocols explicit. However, as clearly implied by our results, its induced assumption on synchrony is rather strong, and corresponds to every party having access to the same absolute clock. Arguably, the only way one can guarantee this in reality, especially in presence of parties that might come and go at will, is by assuming contiguous access to an Internet clock [33].

In this work, we make an important relaxation to the synchrony assumed by blockchain protocols: parties do not have access to such an Internet-clock-style setup, but rather, they have local clocks that advance at roughly the same speed. This assumption can be captured by the straightforward global-setup version of the clock functionality introduced in [25]. The main difference to [25] is that, as a global setup, our clock-functionality is accessible to any party or functionality, and supports a registration/reregistration mechanism as in the global clock discussed in [3]. We note in passing that the clock functionality of [25] was explicitly described as the minimal assumption for synchrony.

Our (weaker) clock setup never exports a global-round counter to any party or functionality. In particular, exported information consists only of an indication of whether or not a new round has started. This can be expressed as in [25] by means of exporting a bit b that switches from b to $1 - b$ with every round switch. For sake of simplicity, we allow our clock to maintain a reference value τ—corresponding to global round—but the exported information to any party is the bit $\tau \bmod 2$.[6] For example, a party that joins an execution cannot infer the time of other parties in the system by observing this output. Still, the functionality allows the party to proceed "at the same speed" as other honest parties in the same session. For completeness, the above clock, which we denote by $\mathcal{G}_{\text{TICK}}$ to avoid confusion with the global clock from [3, 2], is defined in Section A. For simplicity, we restrict our attention to its "perfect" version, where a round-switch is reported to everyone at the very next request. As in [25], one can easily relax the clock to capture loose synchronization, by allowing the adversary to delay informing some parties. Our techniques can be easily extended to this loosely synchronized setting.

Remark 2 (Global vs. local time). We keep the explicit (redundant) counter τ_{sid} in the clock-functionality to be able to refer (in our analysis) to global reference time of the execution of a particular protocol session. In sharp contrast stands the notion of *local time* (also known as logical time), which is a party-specific variable and thus updated by the protocol logic. In our specific case, each party running Ouroboros Chronos maintains a local time-stamp `localTime` which defines the slot number for which it is going to produce the block in this round. As a consiquence, at each global time τ in the execution, parties might thus report different local time-stamps and it is up to the protocol to ensure that the difference of these time-stamps is reasonably bounded.

[6] We note in passing that such a reference value is somewhat redundant, and it was not included in the proposal by [25]. However, it will make the results easier to formulate, as some properties can be best expressed in terms of lifetime of the system (cf. Remark 2).

Modeling Peer-to-Peer Communication. We assume a diffusion network in which all protocol messages sent by honest parties are guaranteed to be fetched by protocol participants after a specific delay Δ. 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, even if the sender of the message is corrupted. Note that this is slightly different than the multicast-functionality from [3, 2] which only guaranteed this bounded delivery for messages sent by honest parties. Nonetheless such a seemingly stronger network can in theory be constructed by simple gossiping over multicast networks: 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 in Section A for completeness.

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 do not even require that the initialization of the initial stakeholders is complete in the same round, i.e., honest parties might start producing blocks even for logical slot 1 in different rounds of the (global) execution. To this aim, we weaken the functionality $\mathcal{F}_{\mathsf{INIT}}$ described in [2] to allow for bounded offsets in starting times. As for protocol initialization, the distinction with previous works is that our functionality $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$ does not enforce that all honest stakeholder receive the created genesis block in the same round, but merely guarantees delivery not more than Δ rounds apart. Looking ahead, the initialization of a protocol is only complete once the genesis block is received. 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. [11] 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").[7] The details of the $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$ functionality appear in Section A.1.

Further Hybrids. The protocol makes use of a VRF (verifiable random function) functionality $\mathcal{F}_{\mathsf{VRF}}$, a KES (key-evolving signature) functionality $\mathcal{F}_{\mathsf{KES}}$, 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 [17].

3 The New Protocol: Ouroboros Chronos

3.1 Overview and Main Challenges

The protocol Ouroboros Chronos inherits its basic mode of operation from Ouroboros Genesis. Recall that in Ouroboros Genesis the execution of the protocol is dependent on a global clock $\mathcal{G}_{\mathrm{CLOCK}}$ that provides to each party the current global time or slot number and allows them to have agreement on the slot number at any instant of the execution. In each slot, each party performs a private lottery to determine whether it is a leader of this slot. If a party P is determined as leader—whose probability is proportional to its stake in the system—it is allowed to create a block for this slot and to publish it via $\mathcal{F}_{\mathrm{N\text{-}MC}}$. The delivery times of this block are under limited adversarial control. Slots are further grouped into epochs, where at the beginning of a new epoch the stake distribution used in the lottery is updated. An epoch consists of a predefined number of R slots, where R is a protocol parameter whose value is chosen based on the security analysis (i.e., it can be seen as a function of a general security parameter).

Ouroboros Chronos operates effectively with much less agreement: the protocol still proceeds in rounds but each party maintains its own local time-stamp and performs the above round actions such as evaluating slot leadership according to its own local time. More technically, the setup $\mathcal{G}_{\mathrm{CLOCK}}$ is replaced by a

[7] 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.

synchronization module $\mathcal{G}_{\text{TICK}}$ that merely indicates that a new round has started. Hence, in order to realize a secure ledger functionality in the dynamic availability setting (where the level of participation varies without a predetermined estimate), it is indispensable that Ouroboros Chronos specifies actions to prevent a large drift in local time-stamps of the participants and at the same time offer newly joining parties the possibility to bootstrap the correct chain and a local time stamp that lies within a reasonable interval with existing participants' timestamps. Looking ahead, this will be achieved by emitting (and embedding in the blockchain) specific "synchronization beacons." In the following, we provide a more detailed overview of the actions and point out the differences between Ouroboros Chronos and Ouroboros Genesis. Before we discuss the exact operations, we take a look at the types of parties that our model of execution allows.

3.2 Party Types

The various basic and derived types of parties used in our analysis follow a similar categorization as used in [2]. For a concise overview, we refer to 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 considered 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 $\mathsf{JoinProc}$ 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 [2]), 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 [2]. 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 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`.

3.3 Technical Overview with Differences to Ouroboros Genesis

All operations are given as pseudo-code in the following. To underline the changes to Ouroboros Genesis we marked the lines that are new to Ouroboros Chronos in blue.

	Basic types of *honest* parties	
Resource	**Resource unavailable**	**Resource available**
random oracle $\mathcal{G}_{\mathrm{RO}}$	*stalled*	*operational*
network $\mathcal{F}_{\mathrm{N\text{-}MC}}$	*offline*	*online*
clock $\mathcal{G}_{\mathrm{TICK}}$	*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: $\quad$ *alert* $:\Leftrightarrow$ *operational* $\wedge$ *online* $\wedge$ *time-aware* $\wedge$ *synchronized* $\wedge$ *sign-capable*

active $:\Leftrightarrow$ *alert* $\vee$ *adversarial* $\vee$ *time-unaware*

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

Fig. 1. Party types.

3.3.1 Basic Operation

Ouroboros Chronos is a ledger-protocol and the main protocol is depicted in Section B.1. 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. Second, the ledger-specific inputs to submit new transactions (SUBMIT), to read the ledger state (READ), and to perform the round actions (MAINTAIN-LEDGER). Note that MAINTAIN-LEDGER inputs are the activations that "make the parties work" and perform their round actions in the main procedure LedgerMaintenance specified in Section B.2. 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.

Finally, the protocol allows a caller controlled access to the features of the global shared setups through this protocol instance. We present the relevant sub-protocols and procedures in handling all these calls in the sequel. We follow the typical stages of a party from registration to playing the lottery and perform the necessary round actions to maintain the ledger. A summary of the state variables appears in Section G.1.

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 [2] 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*.

3.3.2 Registration and Special Procedures

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. The registration handling is given in Section B.4. In contrast to Ouroboros Genesis, the protocol will initialize a party P's local time `P.localTime` to 0. Furthermore, 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 $\mathcal{G}_{\mathrm{TICK}}$ (either 0 or 1).

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 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. Only in this instance's second round (`P.localTime` $= 1$) it will retrieve the genesis block. 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 G.1. In particular, as specified by our setup $\mathcal{F}_{\mathsf{INIT}}$ if the genesis block is delivered shortly after the start of the system, then the party considers itself as synchronized.[8] Otherwise it has to invoke the joining procedure.

Joining. 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 introduction and analysis is a core contribution of this work and the procedure is explained in detail in Section 3.5.

3.3.3 Mode of Operation for Alert Parties

Recall that if a party is synchronized, i.e., part of the system since the beginning or completed the joining procedure, and if all resources are available, then the party is considered alert and runs through the standard round actions described in Section B.2:

- 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 $\mathcal{G}_{\text{TICK}}$ and comparing it to `lastTick`. The procedure is defined in Section B.6.
- Record the arrival times of the synchronization beacons the protocol sends out (call to ProcessBeacons). This feature will be discussed in detail in Section 3.4.
- 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). The procedures involved in chain selection are given in Section B.9, Section B.8, and Section B.7
- 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 Section B.10 and Section B.6.
- 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.

Below we provide more details on the most important aspects of the standard mode of operation.

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 $\mathcal{C}_{\mathsf{loc}}$, or more precisely by an abstract mapping that assigns a party (identified by its public keys) to its stake derived based on the encoded transactions in $\mathcal{C}_{\mathsf{loc}}$ (and the genesis block). 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* [17].

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

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

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" [17]:

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

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 B.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 [2], 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 $\mathsf{isvalidstate}(\vec{\mathtt{st}})$ 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. New to Ouroboros Chronos is the emission of synchronization beacons in the first $R/6$ slots of an epoch $\mathtt{ep}$. To be admissible to emit a beacon, the party evaluates the VRF again as in slot-leadership. To obtain an independent evaluation, we use a new constant called SYNC to obtain domain separation. If the returned value $y \leq T_{\mathsf{P}}^{\mathsf{ep}}$, the party will create a block header and send it on the broadcast network.[9]

Embedding synchronization beacons. Part of the staking procedure is to embed synchronization beacons in the first $2R/3$ slots of an epoch $\mathtt{ep}$. 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 $\mathtt{ep}$) 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 IsValidChain in Section B.7 which is the validity predicate of Ouroboros Genesis, equipped with the additional checks for beacon validity. Note that for a slot leader, we provide for simplicity an extra-predicate ValidSB in Section B.7 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 3.4.

3.3.4 Further Ledger Queries

We discuss further features exported by the ledger protocol Ouroboros Chronos and formally written in Section B.1.

[9] 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.

Submit transactions. As in [2] parties take as inputs transactions that serve as the inputs to the ledger.

Read state. As in [2], 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 "objective" time advancement (rounds) are given in the ideal ledger-functionality that Ouroboros Chronos realizes.

3.3.5 De-Registration and Re-Joining

If a party is alert, it can lose in several ways its status of being alert. Following [2], 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 B.1. The main issue is that such a party — when re-joining — will have to retrace what it missed. Compared to Ouroboros Genesis, this is slightly more 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 B.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 $\mathcal{G}_{\text{TICK}}$ or $\mathcal{F}_{\text{N-MC}}$ be the respective de-registration queries, then it considers itself as de-synchronized. Compared to Ouroboros Genesis, parties in Ouroboros Chronos are aware about their synchronization status. Any party that is de-synchronized will have to run through the main joining procedure of Section 3.5 to become synchronized.

3.4 The Synchronization Procedure of Ouroboros Chronos

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. Each building block is given with the reference to the code implementing it.

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 $\mathsf{P}.\mathtt{localTime} = i \cdot R$ holds for the first time. In the code, the synchronization procedure is invoked as the final step in a synchronization slot in Section B.2).

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

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

where $\mathsf{P}.\mathtt{localTime}$ 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 B.10.

3.) *Arrival times bookkeeping:* Every party P maintains an array $\mathsf{P}.\mathsf{Timestamp}_{SB}(\cdot)$ that assigns to each synchronization beacon $\mathtt{SB}$ a pair $(n, \text{flag}) \in \mathbb{N} \times \{\mathsf{final}, \mathsf{temp}\}$. Assume a beacon $\mathtt{SB}$ with $\mathsf{slotnum}(\mathtt{SB}) \in [j \cdot R + 1, \ldots, j \cdot R + R/6]$, $j \in \mathbb{N}$ and party P' is fetched by party P (for the first time). If the pair $(\mathsf{slotnum}(\mathtt{SB}), \mathsf{P}')$ 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 $(j+1) \cdot R$, $\mathsf{Timestamp}_{SB}(\mathtt{SB})$ is defined as the current slot number and the value is considered final, i.e., $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \triangleq (\mathsf{P}.\mathtt{localTime}, \mathsf{final})$.
- If party P has not yet passed synchronization slot $j \cdot R$ (and thus the beacon belongs logically to this party's next epoch), $\mathsf{Timestamp}_{SB}(\mathtt{SB})$ is defined as the current slot number $\mathsf{P}.\mathtt{localTime}$ and the decision is marked as temporary, i.e., $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \triangleq (\mathsf{P}.\mathtt{localTime}, \mathsf{temp})$. 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 B.3.

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 $I_{\text{sync}}(i) \triangleq [(i-1) \cdot R + 1, \ldots (i-1) \cdot R + R/6]$ 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 $(\mathsf{EvalProve}, \text{sid}, \eta_j \,\|\, \mathsf{P}.\mathtt{localTime} \,\|\, \mathtt{SYNC})$ to $\mathcal{F}_{\mathsf{VRF}}$ returned $(\mathsf{Evaluated}, \text{sid}, y, \pi)$ with $y < T_{\mathsf{P}}^{\mathsf{ep}}$ where $T_{\mathsf{P}}^{\mathsf{ep}}$ is the threshold in the current epoch as used for normal slot leader election. An alert slot leader P' on the other hand will include any valid synchronization beacon in its new block as long as $\mathsf{P}'.\mathtt{localTime}$ 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 B.10.

The remaining three steps are implemented as part of the core synchronization procedure in Section B.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[10] 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

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

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 [2].[11]

3.5 The Joining Procedure for New Parties of Ouroboros Chronos

Introducing synchronization slots into the protocol serves the main purpose of allowing newly joining parties to adjust their local time to a value that lies in the interval of time stamps reported by alert parties. Note that a newly joining party starts at local time 0 and can thus be arbitrarily off of the values reported by alert parties. With our new procedure, a newly joining party can not only bootstrap the current reliable ledger state, but also a reliable time-stamp.

The novel 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 we give an overview and rationale behind our procedure and formally prove its security in Section 4.4. The code for this procedure is given in Figure 2 containing the procedure JoinProc which is is invoked as part of LedgerMaintenance for newly joining parties.

Phase A: Once a newly joining party has all resources available, it will invoke its main round procedure and start the joining process. It will reset all its current 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 B.7). As we prove in Lemma 4, 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).

[11] An alert party reverting a previously reported state implies a common-prefix violation.

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

1: Call $\mathsf{UpdateTime}(\mathsf{P}, R, f)$ // Align with newest round
2: **if** $\mathtt{localTime} > 1$ **then** // Set back to local round 1
3: Set $\mathtt{localTime} \leftarrow 1$
4: Set $\mathtt{ep} \leftarrow \lceil \mathtt{localTime}/R \rceil$, and $\mathtt{sl} \leftarrow \mathtt{localTime}$.
5: $\mathtt{fetchCompleted} \leftarrow \mathsf{false}$, $\mathtt{futureChains}, \mathtt{buffer} \leftarrow \emptyset$, $\mathsf{Timestamp}_{SB} \leftarrow$ empty array.
6: **end if**
7: // Phase B
8: **while** $\mathtt{localTime} \leq t_{\text{off}}$ **do**
9: **if** $\mathtt{fetchCompleted} = \mathsf{false}$ **then**
10: Call $\mathsf{FetchInformation}(k, \mathsf{P})$ and denote fetched chains by $\mathcal{N} := (\mathcal{C}_1, \ldots, \mathcal{C}_M)$
11: Call $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$ // Since $\mathtt{isSync} = \mathsf{false}$, all chains are considered
12: $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$
13: $\mathsf{FinishRound}(\mathsf{P})$ // Mark round actions as finished. Resume below upon next activation
14: **end if**
15: Call $\mathsf{UpdateTime}(\mathsf{P}, R, f)$ to update $\mathtt{localTime}$, $\mathtt{ep}$, and $\mathtt{sl}$ // $\mathtt{fetchCompleted}$ will reset.
16: **end while**
17: // Phases C
18: **while** $\mathtt{localTime} \leq t_{\text{off}} + t_{\text{minSync}} + t_{\text{stable}}$ **do**
19: **if** $\mathtt{fetchCompleted} = \mathsf{false}$ **then**
20: Call $\mathsf{FetchInformation}(k, \mathsf{P})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$.
21: Set $\mathtt{buffer} \leftarrow \mathtt{buffer} || (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and define $\mathtt{futureChains} \leftarrow \mathtt{futureChains} || (\mathcal{C}_1, \ldots, \mathcal{C}_M)$
22: Call $\mathsf{ProcessBeacons}$ to collect new beacons in this round.// All arrival times are temporary
23: Call $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathtt{futureChains}, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
24: $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$
25: $\mathsf{FinishRound}(\mathsf{P})$ // Mark round actions as finished. Resume below upon next activation
26: **end if**
27: Call $\mathsf{UpdateTime}(\mathsf{P}, R, f)$ to update $\mathtt{localTime}$, $\mathtt{ep}$, and $\mathtt{sl}$ // $\mathtt{fetchCompleted}$ will reset.
28: **end while**
29: // Phase D
30: Define the function $I_{\text{sync}}(j) : j \mapsto I_j := [(j-1)R + 1, \ldots, (j-1)R + 2R/3]$.
31: $\mathtt{syncBuffer}_{\text{valid}} \leftarrow \{\mathtt{SB}' \in \mathtt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathsf{sid}, \mathtt{SB}', \mathcal{C}_{\mathsf{loc}}, f, R) = \mathsf{true}\}$
32: Initialize $i := 0$. Now set i to be the minimum positive integer such that $\forall \mathtt{SB} \in \mathcal{C}_{\mathsf{loc}}[I_{\text{sync}}(i)] : \mathtt{SB} \in \mathtt{syncBuffer}_{\text{valid}} \wedge \mathsf{Timestamp}(\mathtt{SB}) > t_{\text{off}} + t_{\text{pre}}$ (if no interval exists, i is unchanged).
33: **if** $i \geq 1$ **then**
34: **for** at most $((t_{\text{stable}} + t_{\text{minSync}}) \operatorname{div} R))$ iterations **do**
35: $\mathcal{S}_i \leftarrow \{\mathtt{SB} \mid \exists B \in \mathcal{C}_{\mathsf{loc}}[I_{\text{sync}}(i)] : \mathtt{SB} \in B \wedge \mathsf{slotnum}(\mathtt{SB}) \in \{(i-1)R + 1, \ldots, (i-1)R + R/6\}\}$
36: **for** each $\mathtt{SB} = (\mathtt{sl}, \mathsf{P}, y, \pi) \in \mathcal{S}_i$ **do**
37: $Q_{\mathtt{SB}} \leftarrow \{\mathtt{SB}' = (\mathtt{sl}', \mathsf{P}', \cdot, \cdot) \in \mathtt{syncBuffer}_{\text{valid}} \mid \mathsf{P}' = \mathsf{P} \wedge \mathtt{sl}' = \mathtt{sl}\}$
38: **if** $Q_{\mathtt{SB}} \neq \emptyset$ **then**
39: $\min_{\mathtt{SB}} \leftarrow \min\{\mathsf{Timestamp}(\mathtt{SB}') \mid \mathtt{SB}' \in Q_{\mathtt{SB}}\}$
40: $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \leftarrow (\min_{\mathtt{SB}}, \mathsf{final})$
41: $\mathsf{recom}(\mathtt{SB}) \leftarrow \mathsf{slotnum}(\mathtt{SB}) - \mathsf{Timestamp}(\mathtt{SB})$
42: **else**
43: $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathtt{SB}\}$ // Negligible probability event in execution.
44: **end if**
45: **end for**
46: $\mathsf{shift}_i \leftarrow \mathsf{med}\,\{\mathsf{recom}(\mathtt{SB}) \mid \mathtt{SB} \in \mathcal{S}_i\}$
47: **for** each $\mathtt{SB}$ with $\mathsf{Timestamp}_{SB}(\mathtt{SB}) = (a, \mathsf{temp})$ **do**
48:) $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{temp})$
49: **end for**
50: Set $\mathtt{localTime} \leftarrow \mathtt{localTime} + \mathsf{shift}_i$; $\mathsf{EpochUpdate}(i) \leftarrow \mathsf{Done}$
51: **Break** if $\mathtt{localTime} \leq (i+1)R$. Otherwise, set $i \leftarrow i + 1$ and continue iteration.
52: **end for**
53: $\mathtt{isSync} \leftarrow \mathsf{true}$; $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathtt{futureChains}, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
54: **for** each beacon $\mathtt{SB} \in \mathtt{syncBuffer}_{\text{valid}}$ with $\mathsf{slotnum}(\mathtt{SB}) \leq (i+1)R$ **do**
55: Parse $\mathsf{Timestamp}_{SB}(\mathtt{SB})$ as (a, temp). Define $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \leftarrow (a, \mathsf{final})$
56: **end for**
57: If $\mathtt{localTime} \leq i \cdot R$ then set $t_{\text{work}} \leftarrow i \cdot R$ // If shifted back before the sync slot of i wait.
58: **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.

4 Security Analysis

In this section we establish the security properties of our protocol. Due to space constraints, all proofs are deferred to Appendix D.

4.1 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 [2]; the crucial difference is that it refers to the types of parties with respect to a *logical slot* as defined in Section 3.2.

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

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:*

- *the* alert stake ratio *be the fraction $\mathcal{S}^-(\mathcal{P}_{alert}[\mathtt{sl}])/\mathcal{S}^+(\mathcal{P}_{active}[\mathtt{sl}])$; and*
- *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 assumption also implies the existence of alert parties in every objective round.

4.2 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 for establishing the composable, UC-framework security guarantees that we are aiming for.

Similarly to [2], we only grant these guarantees to *alert* parties. More importantly for this work, the definitions from [2] 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 $\mathtt{sl}_1 + s \le \mathtt{sl}_2 \le \mathtt{sl}$, so $\mathtt{sl}_2$ is at least s slots ahead of $\mathtt{sl}_1$. Then $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \ge \tau \cdot s$. 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.

Finally, we will also consider a slight variant of chain quality called *existential chain quality*:

Existential Chain Quality ($\exists$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 $\mathtt{sl}_1 + s \le \mathtt{sl}_2 \le \mathtt{sl}$. Then $\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]$ contains at least one alertly generated block (i.e., block generated by an alert party).

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, 17, 2] to analyze it, providing only guarantees based on the logical time would be unsatisfactory, as the parties running the Ouroboros Chronos protocol desire persistence and liveness with respect to the objective time. We will address this translation from logical-time to objective-time guarantees later, when establishing the composable security in Theorem 5.

4.3 Setting with Static Registration

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 $\mathcal{G}_{\mathrm{RO}}$, $\mathcal{F}_{\text{N-MC}}$ or $\mathcal{G}_{\mathrm{TICK}}$). Similarly to [2], we refer to this setting as the *setting with static registration*; we will drop this assumption later.

Definition 3 (Clock skew and Skew_Δ). *Given an honest party* P*, we define its* skew *in slot* $\mathtt{sl}$ *(and denote* $\mathsf{Skew}^{\mathsf{P}}[\mathtt{sl}]$*) the difference between* $\mathtt{sl}$ *and the objective time* t *when* P *enters slot* $\mathtt{sl}$*. For any* $\Delta \ge 0$ *and a slot* $\mathtt{sl}$*, we denote by* $\mathsf{Skew}_\Delta[\mathtt{sl}]$ *the predicate that for all parties that are synchronized in slot* $\mathtt{sl}$*, their skew in this slot differs by at most* Δ*; formally*

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

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.

4.3.1 Single-Epoch Analysis with Δ-Bounded Skew

Before we can analyze the resynchronization procedure SyncProc, we first need to establish the blockchain security properties given in Section 4.2 for Ouroboros Chronos during a single-epoch execution, as the proper functioning of 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 with an additional assumption that 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*. The desired properties are established in the following theorem; its proof is given in a separate Appendix C.

Theorem 1. *Consider the single-epoch execution of the protocol Ouroboros-Chronos with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration and Δ-bounded skew. Let R denote the epoch length in slots, let f be the active-slot coefficient, let Δ be the upper bound on the network delay and let $\widetilde{\Delta} \triangleq 2\Delta$. 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*

$$\alpha \cdot (1-f)^{\widetilde{\Delta}+1} \geq (1+\epsilon)/2\,, \tag{4}$$

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

$$k > 192\widetilde{\Delta}/(\epsilon\beta) \qquad \text{and} \qquad R/6 \geq s = k/(4f) \geq 48\widetilde{\Delta}/(\epsilon\beta f) \tag{5}$$

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*

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

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*

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

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*

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

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*

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

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

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

4.3.2 Properties of SyncProc

Here we establish two key properties of the resynchronization procedure SyncProc given in Section B.11 that is being executed by all alert parties on the edge of any two epochs.

Lemma 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 the following lemma.

Lemma 2 (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 be the upper bound on message delay enforced by $\mathcal{F}_{\text{N-MC}}$; 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 $\mathit{Skew}_\Delta[\mathtt{sl}+1]$ will be satisfied.*

The proof of the above lemma relies on the following two intermediate claims:

(i) All alert parties use the same set of synchronization beacons in their execution of the procedure SyncProc between epochs ep and ep + 1,

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

Lemma 3 (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 be the upper bound on message delay enforced by $\mathcal{F}_{\text{N-MC}}$, and assume $\widetilde{\Delta} \triangleq 2\Delta \leq R/6$. Let $\mathtt{sl} \geq 1$ be the last slot of some epoch $\mathtt{ep} \geq 1$, i.e., such that $\mathtt{sl} \bmod R = 0$, and assume that $\mathsf{Skew}_\Delta[\mathtt{sl}']$ is satisfied for all slots in epoch 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 $\alpha \cdot (1-f) \geq (1+\epsilon)/2$, and if the property $\exists\mathsf{CQ}(R/3)$ is not violated during the execution up to slot sl, then in any invocation of SyncProc by an alert party during sl, the local variable shift computed on line 17 will satisfy $|\mathsf{shift}| \leq 2\Delta$, except with error probability $\exp(\ln L - \Omega(R))$ over the whole execution.*

4.3.3 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. We now show how these guarantees can be extended throughout the whole lifetime of the system consisting of many epochs. The following theorem is established 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 [17] together with the properties of SyncProc from Section 4.3.2 (again together with Theorem 1) for the inductive step.

Theorem 2 (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 network delay 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 $\mathcal{G}_{\text{RO}}$. If the assumptions (4) and (5) 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*

$$QL \cdot \left(\bar{\epsilon}_{\mathsf{CG}}\left(\tau_{\mathsf{CG}}, R/3; R, \epsilon\right) + \bar{\epsilon}_{\mathsf{CP}}\left(\tau_{\mathsf{CG}} R/3; R, \Delta, \epsilon\right) + \bar{\epsilon}_{\exists\mathsf{CQ}}\left(R/3; R, \epsilon\right)\right) \ , \tag{6}$$

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

$$\epsilon_{\mathsf{lift}} \triangleq QL \cdot \left[R^3 \cdot \exp\left(-\frac{(\epsilon\beta f)^2 R}{768}\right) + \frac{19R}{\epsilon^4} \cdot \exp\left(\widetilde{\Delta} - \frac{\epsilon^4 \tau_{\mathsf{CG}} R}{54}\right) + 3\bar{\epsilon}_{\mathsf{mv}}\right] \ . \tag{7}$$

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}.

4.4 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 4.3 remain valid also when new parties join the protocol later during its execution.

Definition 4 (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 (objective) round in which* P *becomes operational, time-aware and online for the first time.*

Lemma 4. *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 an (objective) round in which $\mathsf{P}_{\mathsf{join}}$ is in Phase C or D of its joining procedure. Let $\mathcal{C}_{\mathsf{join}}$ denote a chain held in round t by $\mathsf{P}_{\mathsf{join}}$, and let $\mathcal{C}_{\mathsf{alert}}$ denote a chain held in round $t' \triangleq t - \Delta$ by any party $\mathsf{P}_{\mathsf{alert}}$ alert in round t'.*

Then under the assumptions of Theorem 2 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.

Lemma 5. *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 2 and Lemma 4, 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 $SB \in \mathcal{S}_i^{\mathsf{P}_{\mathsf{join}}} = \mathcal{S}_i^{\mathsf{P}_{\mathsf{alert}}}$, the quantity

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

will differ by at most Δ between the two parties $\mathsf{P} \in \{\mathsf{P}_{\mathsf{join}}, \mathsf{P}_{\mathsf{alert}}\}$.

4.5 The Dynamic-Availability Setting

Using the above analysis of the joining procedure, we now generalize the results from previous sections to the *dynamic availability setting* [2], where the parties get arbitrarily registered and deregistered from their resources upon the decision of the environment. The error term in the theorem corresponds to violating the assumptions of Lemmas 4 and 5.

Theorem 3 (Dynamic availability). *Consider an execution of the full protocol Ouroboros-Chronos in the dynamic-availability setting. Under the assumptions of Theorem 2 and Lemma 5, Ouroboros-Chronos achieves the same guarantees for common prefix (resp. chain growth, chain quality, existential chain quality) as given in Theorem 2 except for the negligible additional error probability*

$$\begin{aligned}\epsilon_{\mathsf{DA}} \triangleq\; &\epsilon_{\mathsf{CP}}(\max\{k\beta/64, \tau_{\mathsf{CG}} R/3\}) + \epsilon_{\mathsf{CG}}(\tau_{\mathsf{CG}}, s)\\ &+ \epsilon_{\exists\mathsf{CQ}}(\tau_{\mathsf{CG}} R/3) + \exp(\ln L - \Omega(R))\ .\end{aligned}$$

4.6 From Logical-Time to Objective-Time Guarantees

In this section, we show how to translate our statements, which basically are statements about the time the parties report, to statements as a function of objective time, i.e., guarantees that an "external" observer of the system could measure. The property under consideration for this is chain-growth.

Lemma 6 (Objective 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 objective 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*

$$\tau_{\mathrm{TG}} \triangleq \Big(1 - \frac{96\Delta + R\epsilon\beta f}{48R}\Big).$$

Then, under the assumptions of Theorem 3, 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. The lemma follows directly from Lemma 3 carried over to the dynamic-availability setting. In particular, the skew that the adversary can apply in every sequence of R objective rounds is at most 2Δ since no more synchronization slots can occur where the synchronized parties adjust their local time-stamps (and in between they increase at the same speed as the objective 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 $R \geq 144\widetilde{\Delta}/(\epsilon\beta f)$. ☐

Corollary 1. *Consider the event that the execution of Ouroboros Chronos under the assumptions of Theorem 3 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 the onset of an objective round* t. *Let further* t_1 *and* t_2 *be two previous objective rounds for which* $t_1 + \delta t \leq t_2 \leq t$. *Let* $\mathtt{sl}_1$ *and* $\mathtt{sl}_2$ *be the slot numbers that* P *reported at the end of objective rounds* t_1 *and* t_2*, respectively. Then it must hold that* $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \geq \tau_{\mathsf{CG},\mathrm{glob}} \cdot \delta t$ *whenever* $\delta t \geq \max\{s/\tau, 48\widetilde{\Delta}\}$.

Proof. By the previous Lemma, if the number of objective rounds elapsed is δt, then in the view of alert party P, at least $\tau_{\mathrm{TG}} \cdot \delta t \geq s$ slots were reported. Thus, by chain growth as of Definition 4.2, the increase in blocks between the reported logical slots $\mathtt{sl}_1$ and $\mathtt{sl}_2$ must be $\tau_{\mathrm{TG}} \cdot \delta t \cdot \tau = (\tau_{\mathrm{TG}} \cdot \tau) \cdot \delta t$. ☐

4.7 Composable Guarantees of Ouroboros Chronos and its Clock Properties

Based on the above analysis, we finally prove composable security of this proof-of-stake protocol by showing that it realizes a ledger functionality that additionally exports additional time-stamps and hence realizes a clock. This precisely gives the guarantees a higher-level protocol can rely on regarding the time-stamps of Ouroboros Chronos. The full details are given in Section E and we give here an overview.

The Export-Time Extension. We introduce the export time extension to the ledger functionality $\mathcal{G}_{\text{LEDGER}}$ of [2]. The requirement on the given guarantees (to alert parties) is that they should be sufficient for cryptographic protocols to work. We introduce a generic extension to the basic ledger functionality and represent time-stamps $\mathtt{time}_{\mathsf{P}}$ associated to party P as a pair (e, t), where t is the actual time stamp, and e refers to what we call a generation (in the real world, e corresponds to the number of adjustments a party has made to its local clock and t the logical time). An alert party's time t in (e, t) is guaranteed to increase during a generation with every tick of the reference speed. Once t hits a generation boundary, defined as multiples of a generation length parameter R_L (which in this work $R_L = R$), the generation 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 $\mathsf{shiftLB}$, $\mathsf{shiftUB}$). Furthermore the ledger enforces that any two alert parties with respective time-stamps (e, t) and (e', t'), satisfy the constraints $|t - t'| \leq \mathtt{timeSlack}_{\mathrm{total}}$ and $|t - t'| \leq \mathtt{timeSlack}_{\mathrm{ep}}$ if $e = e'$, and $|e - e'| \leq 1$ for the respective ledger parameters $\mathtt{timeSlack}_{\mathrm{ep}}$, $\mathtt{timeSlack}_{\mathrm{total}}$ 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 generations. We give an overview of the parameters in Table G.2.

On using the realized clock. To judge the applicability of our exported clock, we describe in Section F how higher-level protocols could use the exported clock in a synchronous computation. 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. However, each weakening of the parameters will result in a higher-level protocol to require specific reactions. This is obviously achievable by standard mechanisms if shifts are to be expected but still $\texttt{timeSlack}_{\text{ep}} = \texttt{timeSlack}_{\text{total}} = 0$. The parties will know that unexpected behavior could happen around the known generation boundary appropriately suspend their round operations and proceed at given later time after the boundary. 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 quantified. If parties can additionally 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 $\texttt{timeSlack}_{\text{ep}}$ and this allows a higher level protocol to deal with this bounded skew by standard mechanisms [25].

We conclude Section F by providing an overview on how the clock parameters do improve in more predictable and less adversarial networks, which in particular allows higher-level protocols to effectively emulate a timing service based on the timestamps provided by the Chronos protocol.

References

[1] 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.

[2] 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.

[3] 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.

[4] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th ACM STOC*, pages 52–61. ACM Press, May 1993.

[5] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In Jim Anderson and Sam Toueg, editors, *13th ACM PODC*, pages 183–192. ACM, August 1994.

[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] Vitalik Buterin. A next-generation smart contract and decentralized application platform, 2013. `https://github.com/ethereum/wiki/wiki/White-Paper`.

[9] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

[10] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology*, 18(3):219–246, 2005.

[11] 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.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[13] 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.

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

[15] Ran Cohen. Asynchronous secure multiparty computation in constant time. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 183–207. Springer, 2016.

[16] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 998–1021, 2016.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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`.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. `http://eprint.iacr.org/2017/454`.

[23] 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.

[24] 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`.

[25] 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.

[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] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.

[28] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2005.

[29] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. In Jon M. Kleinberg, editor, *38th ACM STOC*, pages 109–118. ACM Press, May 2006.

[30] 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.

[31] 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.

[32] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

[33] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition.* CRC Press, 2010.

[34] Tal Moran and Ilan Orlov. Proofs of space-time and rational proofs of storage. *IACR Cryptology ePrint Archive*, 2016:35, 2016.

[35] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `http://bitcoin.org/bitcoin.pdf`.

[36] 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.

[37] Sunoo Park, Krzysztof Pietrzak, Albert Kwon, Joël Alwen, Georg Fuchsbauer, and Peter Gazi. Spacemint: A cryptocurrency based on proofs of space. *IACR Cryptology ePrint Archive*, 2015:528, 2015.

[38] 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.

[39] 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.

[40] 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.

SUPPLEMENTARY MATERIAL

A Completing the Setup Functionality Description

The purpose of this section is to introduce the setup functionalities that have formally been weakened for the treatment in this work.

Synchrony. The clock functionality from [2] and replace by a weaker functionality $\mathcal{G}_{\text{TICK}}$ that instead of exporting global time, exports simply a bit via which parties can observe rounds passing. A party that joins the protocol execution has therefore no possibility to infer the time of existing parties. Still, the $\mathcal{G}_{\text{TICK}}$ offers the possibility to proceed "at the same speed" as all other honest parties in the same session. Our new functionality is given below where the only difference to the clock functionality in [2] is colored in blue.

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

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 session sid the clock maintains a variable τ_{sid}. For each identity $\mathsf{P} := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable d_{P}. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})}$ (all integer variables are initially 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 $(\text{CLOCK-READ}, \text{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary (on behalf of a corrupted party), or any ideal—shared or local—functionality), first compute $\text{tick} \leftarrow \tau_{\text{sid}} \mod 2$ and return $(\text{CLOCK-READ}, \text{sid}_C, \text{tick})$ 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 set $\tau_{\text{sid}} := \tau_{\text{sid}} + 1$ and reset $d_{(\mathcal{F},\text{sid})} := 0$ and $d_{\mathsf{P}} := 0$ for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$.

The network model. As outlined in the main body, we work in a setting where the adversary is required to the same multicast restrictions as an honest party (which gives a broadcast-like guarantee). Such a functionality can theoretically be achieved by a flooding protocol over a multicast network such as the one in [2]. Our network functionality is given below where the only difference to the one in [2] is colored in blue.

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

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 $\Delta_{\text{mid}_i} = \Delta$ for $i = 1, \ldots, n$ and set $\vec{M} := \vec{M}||(m, \text{mid}_1, D_{\text{mid}_1}, U_1)|| \ldots ||(m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, 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 (FETCH, sid) from $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P if P is corrupted):
  1. For all tuples $(m, \mathrm{mid}, D_{\mathrm{mid}}, \mathsf{P}) \in \vec{M}$, set $D_{\mathrm{mid}} := D_{\mathrm{mid}} - 1$.
  2. Let $\vec{M}_0^{\mathsf{P}}$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \mathrm{mid}, D_{\mathrm{mid}}, \mathsf{P})$ with $D_{\mathrm{mid}} = 0$ (in the same order as they appear in $\vec{M}$). Then, delete all entries in $\vec{M}_0^{\mathsf{P}}$ from $\vec{M}$ and in case some $(m, \mathrm{mid}, D_{\mathrm{mid}}, \mathsf{P})$ is in $\vec{M}_0^{\mathsf{P}}$, where P is honest, set $\Delta_{\mathrm{mid}'} = \Delta$ for any $(m, \mathrm{mid}', D_{\mathrm{mid}'}, \mathsf{P}')$ in $\vec{M}$ and replace this record by $(m, \mathrm{mid}', \min\{D_{\mathrm{mid}'}, \Delta\}, \mathsf{P}')$. Finally, send $\vec{M}_0^{\mathsf{P}}$ to P.

– **Adding adversarial delays.** Upon receiving $(\text{DELAYS}, \mathrm{sid}, (T_{\mathrm{mid}_{i_1}}, \mathrm{mid}_{i_1}), \ldots, (T_{\mathrm{mid}_{i_\ell}}, \mathrm{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\mathrm{mid}_{i_j}}, \mathrm{mid}_{i_j})$:
  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 $\vec{M}$, 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 $(\text{SWAP}, \mathrm{sid}, \mathrm{mid}, \mathrm{mid}')$ from the adversary, if mid and mid′ are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \mathrm{mid}, D_{\mathrm{mid}}, \cdot)$ and $(m, \mathrm{mid}', D_{\mathrm{mid}'}, \cdot)$ in $\vec{M}$. Return (SWAP, sid) to the adversary.

A.1 Genesis Block Distribution and Implicit Clock Initialization

Functionality $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$

The functionalsity $\mathcal{F}_{\mathsf{INIT}}$ is parameterized by 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, a variable $\mathsf{counter}$ to steer when a stakeholder receives the genesis block, and a variable `lastTick` to determine when a new round started. 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 $(\mathsf{DefineOffset}, sid, o_1, \ldots, o_n)$, set $\mathsf{offset}_i := \max(o_i, \Delta)$. It further sets $\mathsf{counter} \leftarrow 0$. Proceed handling the first input as specified in the following.
– On receiving any input by a registered party, the functionality first sends $(\text{CLOCK-READ}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{TICK}}$ to obtain the clock-tick d and if $\texttt{lastTick} \neq d$, it sets $\mathsf{counter} \leftarrow \mathsf{counter} + 1$ and sets $\texttt{lastTick} \leftarrow d$. Subsequently:
  - If $\mathsf{counter} = 0$ 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 $0 < \mathsf{counter} \leq \Delta$, then do the following
    * If any of the n initial stakeholders has not sent a request of the above form, i.e., a $(\mathsf{ver_keys}, sid, \mathsf{P}_i, v_i^{\mathrm{vrf}}, v_i^{\mathrm{kes}})$-message, to $\mathcal{F}_{\mathsf{INIT}}$ in the same round then $\mathcal{F}_{\mathsf{INIT}}$ outputs an error.
    * Otherwise, if the currently received input is a request of the form $(\mathsf{genblock_req}, sid, \mathsf{P})$ from any initial stakeholder $\mathsf{P} = \mathsf{P}_i$ for some $i \in [n]$, and $\mathsf{counter} \geq \mathsf{offset}_i$, $\mathcal{F}_{\mathsf{INIT}}$ sends $(\mathsf{genblock}, sid, (\mathbb{S}_1, \eta_1))$ to P.
    * Finally, if it is a request from a party that is not initial stakeholder, simply return $(\mathsf{genblock}, sid, (\mathbb{S}_1, \eta_1))$.
  - If $\mathsf{counter} > \Delta$ then do the following:
    * If any of the n initial stakeholders has not sent a request of the above form, i.e., a $(\mathsf{ver_keys}, sid, \mathsf{P}_i, v_i^{\mathrm{vrf}}, v_i^{\mathrm{kes}})$-message, to $\mathcal{F}_{\mathsf{INIT}}$ in the same round then $\mathcal{F}_{\mathsf{INIT}}$ outputs an error.
    * Return $((\mathsf{genblock}, sid, (\mathbb{S}_1, \eta_1)), \mathsf{Running})$ to the requestor.

B Completing the Chronos Protocol Description

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. Again, changes compared to Ouroboros Genesis are marked in blue.

B.1 The Main Protocol Instance

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

Global Variables:

- Read-only (parameters): R, k, f, s, t_{off}, t_{stable}, t_{minSync}, t_{pre}
- Read-write: $v^{\text{vrf}}_{\mathsf{P}}$, $v^{\text{kes}}_{\mathsf{P}}$, `localTime`, `ep`, `sl`, $\mathcal{C}_{\text{loc}}$, $T^{\text{ep}}_{\mathsf{P}}$, `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 $(\text{REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{TICK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol Registration-Chronos$(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\text{DE-REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{TICK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol Deregistration-Chronos$(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\text{IS-REGISTERED}, \mathrm{sid})$ return $(\text{REGISTER}, \mathrm{sid}, 1)$ if the local registry `Reg` indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}_{\text{LEDGER}}$ (and did not de-register since then). Otherwise, return $(\text{REGISTER}, \mathrm{sid}, 0)$.

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

- **If** $I = (\text{SUBMIT}, \mathrm{sid}, \mathtt{tx})$ **then** set $\mathtt{buffer} \leftarrow \mathtt{buffer}||\mathtt{tx}$, and send $(\text{MULTICAST}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{F}^{\Delta}_{\text{N-MC}}$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \mathrm{sid}, \mathrm{minerID})$ **then** invoke protocol LedgerMaintenance$(\mathcal{C}_{\text{loc}}, \mathsf{P}, \mathrm{sid}, k, s, R, f)$; if LedgerMaintenance halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\text{READ}, \mathrm{sid})$ **then** invoke protocol ReadState$(k, \mathcal{C}_{\text{loc}}, \mathsf{P}, \mathrm{sid}, R, f)$.
- **If** $I = (\text{EXPORT-TIME}, \mathrm{sid})$ **then** do the following: if `isSync` or `isInit` is false, then return $(\text{EXPORT-TIME}, \mathrm{sid}, \perp)$ to the caller. Otherwise call UpdateTime(P, R) and do:
  1. Define e to be the highest value s.t. EpochUpdate(e) = Done.
  2. Return $(\text{EXPORT-TIME}, \mathrm{sid}, (e, \mathtt{localTime}))$ to the caller.

Handling calls to the shared setup:

- Upon receiving $(\text{CLOCK-READ}, \mathrm{sid}_C)$ forward it to $\mathcal{G}_{\text{TICK}}$ and output $\mathcal{G}_{\text{TICK}}$'s response.
- Upon receiving $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$, record that a clock-update was received in the current round. If the party is registered to all its setups, then do nothing further.
  Otherwise, do the following operations *before concluding this round*:
  1. If this instance is currently time-aware but otherwise stalled or offline, then call UpdateTime(P, R) to update `localTime` and evolve the KES signing key by sending $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, 0, \mathtt{localTime})$ to $\mathcal{F}_{\mathsf{KES}}$. If the party has passed a synchronization slot, then set $\mathtt{isSync} \leftarrow \mathsf{false}$.
  2. If this instance is only stalled but $\mathtt{isSync} = \mathsf{true}$, then additionally execute FetchInformation$(\mathsf{P}, \mathrm{sid})$, extract all new synchronization beacons $\mathcal{B}$ from the fetched chains, and invoke ProcessBeacons$(\mathsf{P}, \mathrm{sid}, \mathcal{B})$ to correctly process all message that are due in this round and set $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$. Also, complete any unfinished interruptible execution of this round.
  3. Forward $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{TICK}}$ to finally conclude the round.
- Upon receiving $(\text{EVAL}, \mathrm{sid}_{RO}, x)$ forward the query to $\mathcal{G}_{\text{RO}}$ and output $\mathcal{G}_{\text{RO}}$'s response.

B.2 Ledger Maintenance

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

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

1: **if** `isInit` is false **then** invoke Initialization-Chronos($\mathsf{P}, \mathrm{sid}, R$); if Initialization-Chronos halts then halt (this will abort the execution)
end if
2: // From here the variables $v_p^{\mathrm{vrf}}, v_p^{\mathrm{kes}}$, `localTime`, `ep`, `sl`, $\mathcal{C}_{\mathsf{loc}}$, `isSync`, T_p^{ep}, `fetchCompleted`, t_{work} can be used to read from as they are guaranteed to be initialized.
3: **if** `isSync` and stalled before (and now up and running) **then**
4: SimulateClockAdjustments(P, R, k, f, s)
5: **end if**
6: **if not** `isSync` **then**
7: Call JoinProc($\mathsf{P}, \mathrm{sid}, R, k, f, s, t_{\mathrm{off}}, t_{\mathrm{stable}}, t_{\mathrm{minSync}}$)
8: **end if**
9: // normal operation when alert
10: Call FetchInformation(P, sid) and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$.
11: Set `buffer` $\leftarrow$ `buffer`$||(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and define `futureChains` $\leftarrow$ `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 $\mathcal{B} \leftarrow \{\mathtt{SB}_1, \ldots, \mathtt{SB}_n\}$ contained in $\mathcal{C}_1, \ldots, \mathcal{C}_M$ and not yet contained in `syncBuffer`.
15: Call ProcessBeacons($\mathsf{P}, \mathrm{sid}, \mathcal{B}$)
16: Let $\mathcal{N}_0$ be the subsequence of `futureChains` s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq$ `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($\mathsf{P}, \mathrm{sid}, k, \mathsf{P}, R, f$) to update the values $\mathbb{S}_{\mathsf{ep}}, \alpha_{\mathsf{P}}^{\mathsf{ep}}, T_{\mathsf{P}}^{\mathsf{ep}}$, 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` mod $R = 0$ **then**
25: Call SyncProc($\mathsf{P}, \mathrm{sid}, R$)
26: **end if**
end if
27: Call FinishRound(P) // Mark normal round actions as finished.

B.3 Process Beacons and Arrival Times

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

1: **if not** `fetchCompleted` **then**
2: Send (FETCH, sid) to $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$. denote the ith response from $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$ 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 $\mathsf{Timestamp}_{SB}(\mathtt{SB}) = \bot$ **do**
5: `syncBuffer` $\leftarrow$ `syncBuffer` $\cup \{\mathtt{SB}\}$
6: Let ep be the epoch number $\mathsf{slotnum}(\mathtt{SB})$ belongs to
7: **if** `isSync` $\wedge$ (EpochUpdate($ep - 1$) = 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** $\texttt{isSync}$ **then**
15: Remove from $\texttt{syncBuffer}$ all beacons such that $\mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}, \mathcal{C}_{\mathsf{loc}}, f, R)$ returns false.
16: $\texttt{syncBuffer}_{\text{valid}} \leftarrow \{\mathtt{SB}' \in \texttt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}', \mathcal{C}_{\mathsf{loc}}, f, R) = \mathsf{true}\}$
17: Let $L = (\mathtt{SB}_1, \ldots, \mathtt{SB}_n)$ be a canonical ordering of $\texttt{syncBuffer}_{\text{valid}}$
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 $\texttt{syncBuffer}$ 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 $\mathcal{Z}$).

B.4 Registration and De-registration

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

1: **if** $\mathcal{G} \in \{\mathcal{G}_{\text{TICK}}, \mathcal{G}_{\text{RO}}\}$ **then** send $(\text{REGISTER}, \mathrm{sid})$ to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
2: **end if**
3: **if** $\mathcal{G} = \mathcal{G}_{\text{LEDGER}}$ **then**
4: **if** the party is not registered with $\mathcal{G}_{\text{TICK}}$ or $\mathcal{G}_{\text{RO}}$ **then** or already registered with all setups ignore this input
5: **else**
6: **for** each $\mathcal{F} \in \{\mathcal{F}_{\mathsf{INIT}}^{\Delta}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}}\}$ **do**
7: Send $(\text{REGISTER}, \mathrm{sid})$ to $\mathcal{F}$, set its registration status as registered with $\mathcal{F}$, but do not output the received values.
end for
8: Send $(\text{CLOCK-READ}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{TICK}}$ and receive $(\text{CLOCK-READ}, \mathrm{sid}_C, \mathrm{tick})$ and set $\texttt{lastTick} \leftarrow \mathrm{tick}$
9: Send $(\text{REGISTER}, \mathrm{sid})$ to $\mathcal{F}_{\text{N-MC}}^{\Delta}$.
10: 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 $(\text{REGISTER}, \mathrm{sid}, \mathsf{P})$ once completing the registration with all the above resources $\mathcal{F}$.
end if
end if

And De-registration:

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

1: If the party is alert, set $\texttt{lastTimeAlert} \leftarrow \texttt{localTime}$
2: **if** $\mathcal{G} \in \{\mathcal{G}_{\text{TICK}}, \mathcal{G}_{\text{RO}}\}$ **then**
3: **if** $\mathcal{G} = \mathcal{G}_{\text{TICK}}$ **then** set $\texttt{isSync} \leftarrow \mathsf{false}$
4: Send $(\text{DE-REGISTER}, \mathrm{sid})$ 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 $(\text{DE-REGISTER}, \mathrm{sid})$ to $\mathcal{F}_{\text{N-MC}}^{\Delta}$, set its registration status as de-registered with $\mathcal{F}_{\text{N-MC}}^{\Delta}$ and output $(\text{DE-REGISTER}, \mathrm{sid}, \mathsf{P})$.
end if

B.5 Initialization

Protocol Initialization-Chronos(P, sid, R)

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

1: Send (KeyGen, sid, P) to $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$; receiving (VerificationKey, sid, $v_{\mathsf{P}}^{\mathrm{vrf}}$) and (VerificationKey, sid, $v_{\mathsf{P}}^{\mathrm{kes}}$), respectively.
// The following branch is executed on the first maintenance query after registration of this instance
2: **if** `localTime` $= 0$ **then**
3: Send (ver_keys, sid, 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 (genblock_req, sid, 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 (genblock_req, sid, P) to $\mathcal{F}_{\mathsf{INIT}}$.
17: **end while**
18: Receive the genesis block (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: Set $T_p^{\mathsf{ep}} \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha_p^{\mathsf{ep}})$ as the threshold for stakeholder P for epoch `ep`, where α_p^{ep} is the relative stake of stakeholder P in $\mathbb{S}_{\mathsf{ep}}$ and ℓ_{VRF} denotes the output length of $\mathcal{F}_{\mathsf{VRF}}$.
21: If $\mathcal{F}_{\mathsf{INIT}}$ did not mark the returned value as as Running then additionally set `isSync` $\leftarrow$ true (the execution just started).
end if
22: Set `isInit` $\leftarrow$ true and `fetchCompleted` $\leftarrow$ false, $t_{\mathrm{work}} \leftarrow 0$, `lastTimeAlert` $\leftarrow 0$
23: `buffer` $\leftarrow \emptyset$, `futureChains` $\leftarrow \emptyset$
24: EpochUpdate($\cdot$) $\leftarrow$ empty table (initial symbol $\perp$), EpochUpdate(0) $\leftarrow$ Done

B.6 Fetching information and stake distribution; time update

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

Protocol FetchInformation(P, 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 (FETCH, sid) to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$; denote the ith response from $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ by (FETCH, 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 $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$; denote the ith response from $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ by $(\textsc{fetch}, \mathrm{sid}, b_i)$.
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: $\quad$ $\mathtt{buffer} \leftarrow \mathtt{buffer} || (\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
12: $\quad$ $\mathtt{futureChains} \leftarrow \mathtt{futureChains} \cup \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$
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 $\mathcal{Z}$).

Protocol UpdateStakeDist(k, P, R, f)

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

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

And, finally, the time update procedure:

Protocol UpdateTime(P, R)

$\quad$ // Precondition: Only executed if time-aware
1: Send $(\textsc{clock-read}, \mathrm{sid}_C)$ to $\mathcal{G}_{\textsc{tick}}$ and receive $(\textsc{clock-read}, \mathrm{sid}_C, \mathrm{tick})$
2: **if** $\mathtt{lastTick} \neq \mathrm{tick}$ **then**
3: $\quad$ $\mathtt{lastTick} \leftarrow \mathrm{tick}$
4: $\quad$ $\mathtt{localTime} \leftarrow \mathtt{localTime} + 1$
5: $\quad$ $\mathtt{fetchCompleted} \leftarrow \mathsf{false}$
6: **end if**
7: Set $\mathtt{ep} \leftarrow \lceil \mathtt{localTime}/R \rceil$, and $\mathtt{sl} \leftarrow \mathtt{localTime}$.

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

B.7 Chain Verification and Beacon Validity

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 $\mathsf{isvalidstate}(\vec{\mathsf{st}}) = 0$ **then**
$\quad$ **return** false
end if
if isSync and $(\exists B \in \mathcal{C} : \mathsf{slotnum}(B) > \mathtt{localTime})$ **then**
$\quad$ **return** false
end if
for each epoch $\mathtt{ep}$ **do**
$\quad$ // Derive stake distribution and randomness for this epoch from $\mathcal{C}$
$\quad$ // In the following, $H(\cdot)$ stands for an RO evaluation for simplicity.
$\quad$ Set $\mathbb{S}^{\mathcal{C}}_{\mathsf{ep}}$ to be the stakeholder distribution at the end of epoch $\mathtt{ep} - 2$ in $\mathcal{C}$.
$\quad$ Set $\alpha^{\mathsf{ep},\mathcal{C}}_{\mathsf{P}'}$ to be the relative stake of any party P' in $\mathbb{S}^{\mathcal{C}}_{\mathsf{ep}}$ and $T^{\mathsf{ep},\mathcal{C}}_{\mathsf{P}'} \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha^{\mathsf{ep},\mathcal{C}}_{\mathsf{P}'})$.
$\quad$ Set $\eta^{\mathcal{C}}_{\mathsf{ep}} \leftarrow H(\eta^{\mathcal{C}}_{\mathsf{ep}-1} \,\|\, \mathtt{ep} \,\|\, v)$ 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 $\mathbf{G}$.
$\quad$ **for** each block B in $\mathcal{C}$ from epoch $\mathtt{ep}$ **do**
$\quad\quad$ Parse B as $(h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$.
$\quad\quad$ // 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 $(\mathsf{Verify}, sid, \eta_{\mathsf{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST}, y, \pi, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$,
denote its response by $(\mathsf{Verified}, sid, \eta_{\mathsf{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST}, y, \pi, b_1)$.
Send $(\mathsf{Verify}, sid, \eta_{\mathsf{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$,
denote its response by $(\mathsf{Verified}, sid, \eta_{\mathsf{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, b_2)$,
Set $\mathsf{badvrf} \leftarrow \left(b_1 = 0 \vee b_2 = 0 \vee y \geq T^{\mathsf{ep},\mathcal{C}}_{\mathsf{P}'}\right)$.
// 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 $\exists \mathtt{SB} \in B \wedge \mathsf{slotnum}(B) > (\mathsf{ep} - 1)R + 2R/3$ **then**
Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
else if $\exists \mathtt{SB} \in B : \mathsf{slotnum}(\mathtt{SB}) > \mathsf{slotnum}(B) \vee \mathsf{slotnum}(\mathtt{SB}) \notin [(\mathsf{ep} - 1)R + 1, \mathsf{ep} \cdot R]$ **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 $(\mathsf{Verify}, sid, \eta_{\mathsf{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{SYNC}, y, \pi, v^{\mathrm{vrf}}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$.
Denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Verified}, sid, \eta_{\mathsf{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{SYNC}, y, \pi, b_4)$,
if $(b_4 = 0)$ or $(y \geq T^{\mathsf{ep},\mathcal{C}}_{\mathsf{P}'})$ **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.

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. $\mathcal{C}$, 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 $T^{ep',\mathcal{C}}_{\mathsf{P}'} \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha^{ep',\mathcal{C}}_{\mathsf{P}'})$.
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 (Verify, $sid, \eta_{ep'} \| \texttt{sl}' \| \texttt{SYNC}, y, \pi, v^{\text{vrf}}_{\mathsf{P}'}$) to $\mathcal{F}_{\mathsf{VRF}}$.
Denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by (Verified, $sid, \eta_{ep'} \| \texttt{sl}' \| \texttt{SYNC}, y, \pi, b_1$),
if $b_1 = 0$ or $y \geq T^{ep',\mathcal{C}}_{\mathsf{P}'}$ **then**
  return false
end if
return true

B.8 Select Chain

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

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

B.9 The Genesis Chain Selection Rule

Algorithm maxvalid-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 $\texttt{sl}_{j'}\}$
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** $\mathcal{C}_{\mathsf{max}}$.

B.10 The Staking Procedure

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

B.11 Code of the Synchronization Procedure

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

1: // Only called when: P is alert, $\mathtt{localTime} \mod R = 0$ and $\mathtt{localTime} > 0$
2: Set $i \leftarrow \mathtt{localTime}$ div R
3: **if** (**not** $\mathsf{EpochUpdate}(i) = \mathsf{Done}$) **then**
4:     $\mathsf{EpochUpdate}(i) \leftarrow \mathsf{Done}$ // Remember that clock adjustment has happened
5:     $\mathcal{B}_i \leftarrow \mathcal{C}_{\mathsf{loc}}[(i-1)R : (i-1)R + 2R/3]$

6: $\mathcal{S}_i \leftarrow \{\mathtt{SB} \mid \exists B \in \mathcal{B}_i : \mathtt{SB} \in B \wedge \mathsf{slotnum}(\mathtt{SB}) \in \{(i-1)R, \ldots, (i-1)R + R/6\}\}$
7: **for** each $\mathtt{SB} = (\mathtt{sl}, \mathsf{P}, y, \pi) \in \mathcal{S}_i$ **do**
8: // Find representative beacon and compute recommendation.
9: Find unique $\mathtt{SB}' = (\mathtt{sl}, \mathsf{P}, \cdot, \cdot) \in \mathtt{syncBuffer}$. If inexistent, set $\mathtt{SB}' \leftarrow \bot$.
10: **if** $\mathtt{SB}' \neq \bot$ **then**
11: Set $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \leftarrow \mathsf{Timestamp}_{SB}(\mathtt{SB}')$ // Assign correct value
12: $\mathsf{recom}(\mathtt{SB}) \leftarrow \mathsf{slotnum}(\mathtt{SB}) - \mathsf{Timestamp}(\mathtt{SB})$
13: **else**
14: $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathtt{SB}\}$ // Negligible probability event in execution.
15: **end if**
16: **end for**
17: $\mathsf{shift}_i \leftarrow \mathsf{med}\,\{\mathsf{recom}(\mathtt{SB}) \mid \mathtt{SB} \in \mathcal{S}_i\}$
18: **for** each $\mathtt{SB}$ with $\mathsf{Timestamp}_{SB}(\mathtt{SB}) = (a, \mathsf{temp})$ **do**
19: $\mathsf{Timestamp}_{SB}(\mathtt{SB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{final})$
20: **end for**
21: **if** $\mathsf{shift}_i > 0$ **then** // Move fast forward
22: $\mathsf{newTime} \leftarrow \mathtt{localTime} + \mathsf{shift}_i$
23: $\mathcal{M}_{\text{chains}} \leftarrow \mathcal{M}_{\text{sync}} \leftarrow \emptyset$
24: **while** $\mathtt{localTime} < \mathsf{newTime}$ **do**
25: $\mathtt{localTime} \leftarrow \mathtt{localTime} + 1$
26: Let $\mathcal{N}_0$ be the subsequence of $\mathtt{futureChains}$ s.t.
27: $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \mathtt{localTime}$
28: Remove each $\mathcal{C} \in \mathcal{N}_0$ from $\mathtt{futureChains}$.
29: Call $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$
30: Call $\mathsf{UpdateStakeDist}(k, \mathsf{P}, R, f)$
31: 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).
32: **end while**
33: Send $(\textsc{multicast}, \mathrm{sid}, \mathcal{M}_{\text{chains}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ and $(\textsc{multicast}, \mathrm{sid}, \mathcal{M}_{\text{sync}})$ to $\mathcal{F}^{\mathsf{sync}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
34: **end if**
35: **if** $\mathsf{shift}_i < 0$ **then** // Need to wait
36: Set $t_{\text{work}} \leftarrow \mathtt{localTime}$
37: 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".
38: **end if**
39: **end if**

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

B.12 Reading the Ledger State

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

1: If $\mathtt{isInit}$ or $\mathtt{isSync}$ is false output the empty state $(\textsc{read}, \mathrm{sid}, \varepsilon)$ (to $\mathcal{Z}$). Otherwise, do the following:
2: Call $\mathsf{FetchInformation}(k, \mathsf{P})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$.
3: Set $\mathtt{buffer} \leftarrow \mathtt{buffer} || (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and define $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$
4: Call $\mathsf{UpdateTime}(\mathsf{P}, R)$
5: Call $\mathsf{ProcessBeacons}(\mathsf{P}, \mathrm{sid})$
6: Let $\mathcal{N}_0 := \{\mathcal{C} \in \mathcal{N} \cup \mathtt{futureChains} \mid \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \mathtt{localTime}\}$
7: Let $\mathcal{N}_1 := \{\mathcal{C} \in \mathcal{N} \mid \exists B \in \mathcal{C} : \mathsf{slotnum}(B) > \mathtt{localTime}\}$
8: $\mathtt{futureChains} \leftarrow (\mathtt{futureChains} \setminus \mathcal{N}_0) \cup \mathcal{N}_1$
9: $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$
10: Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ to update $\mathcal{C}_{\mathsf{loc}}$

11: Extract the state $\vec{\mathtt{st}}$ from the current local chain $\mathcal{C}_{\mathsf{loc}}$.
12: Output $(\textsc{read}, \mathrm{sid}, \vec{\mathtt{st}}^{\lceil k})$ (to $\mathcal{Z}$). // $\vec{\mathtt{st}}^{\lceil k}$ denotes the prefix of $\vec{\mathtt{st}}$ with the last k state blocks chopped off

B.13 Simulate Clock Adjustments

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

1: $\mathsf{simulatedTime} \leftarrow \mathtt{lastTimeAlert}$
2: **for** $\mathtt{localTime} - \mathtt{lastTimeAlert}$ iterations **do**
3: Let $\mathcal{N}_0$ be the subsequence of $\mathtt{futureChains}$ s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall B \in \mathcal{C} : \mathsf{slotnum}(B) \leq \mathsf{simulatedTime}$
4: Remove each $\mathcal{C} \in \mathcal{N}_0$ from $\mathtt{futureChains}$.
5: Emulate $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, k, s, R, f)$ with simulated time $\mathsf{simulatedTime}$ (instead of $\mathtt{localTime}$)
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 $\mathtt{localTime}$)
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 $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, 0, \mathtt{localTime})$ to $\mathcal{F}_{\mathsf{KES}}$
15: Set $t_{\mathrm{work}} \leftarrow \mathsf{localTime}$
16: Set $\mathtt{localTime} \leftarrow \mathsf{simulatedTime}$

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

B.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 $\mathcal{G}_{\text{TICK}}$. 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.[12]

Protocol FinishRound(P)

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

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

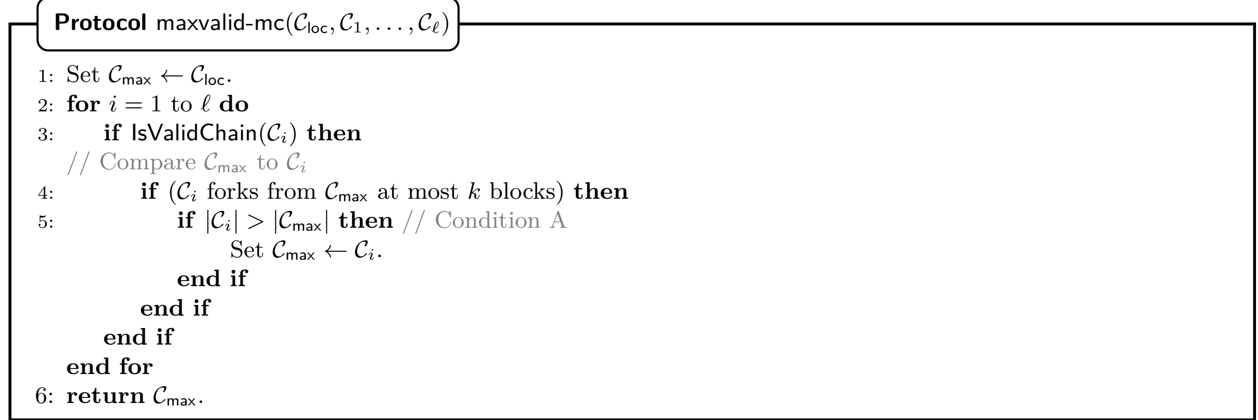
In this section we prove Theorem 1. First, in Section C.1, we introduce a simplified chain-selection rule that will make our protocol easier to analyze. In Section C.2 we draw the connection between a single-epoch execution of this simplified protocol and the formalism of characteristic strings and forks that we later employ. We then analyze the distribution of the characteristic strings induced by an execution of the

[12] Note that in the ideal world, it is the ledger functionality which is registered with $\mathcal{G}_{\text{TICK}}$ and enforces the same principal time-evolving behavior as in the real world.

simplified Ouroboros Chronos in Section C.3, and establish the implications of that for the properties CP, CG and CQ in Section C.4. Finally, in Section C.5, we finally replace the simplified chain-selection rule with the actual one, concluding the proof of Theorem 1.

C.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, 17, 2], 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. B.8; we will denote this variant $\text{Ouroboros-Chronos}_{\mathsf{mc}}$ for conciseness.


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

1: Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_{\mathsf{loc}}$.
2: **for** $i = 1$ to ℓ **do**
3: **if** IsValidChain($\mathcal{C}_i$) **then**
// Compare $\mathcal{C}_{\mathsf{max}}$ to $\mathcal{C}_i$
4: **if** ($\mathcal{C}_i$ forks from $\mathcal{C}_{\mathsf{max}}$ at most k blocks) **then**
5: **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
6: **return** $\mathcal{C}_{\mathsf{max}}$.

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

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. B.8). The latter rule allows for so-called *bootstrapping from genesis* [2] (hence "bg") and so we adopt it for Ouroboros Chronos as well, the consequences for our analysis are discussed in Section C.5.

C.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.

Definition 5 (Characteristic string [26, 17, 2]). *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 $w \in \{0, 1, \bot\}^R$ *of S to be the random variable so that*

$$w_j = \begin{cases} \bot & \text{if } \mathrm{P}(j) = \emptyset, \\ 0 & \text{if } |\mathrm{P}(j)| = 1 \text{ and the assigned party is alert,} \\ 1 & \text{otherwise.} \end{cases} \tag{8}$$

For such a characteristic string $w \in \{0, 1, \bot\}^*$ we say that the index j is uniquely alert *if $w_j = 0$,* empty *if $w_j = \bot$, 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 6 (Δ-fork [17]). *Let $w \in \{0,1,\bot\}^k$ and Δ be a non-negative integer. Let $A = \{i \mid w_i \neq \bot\}$ denote the set of potentially active indices, and let $H = \{i \mid w_i = 0\}$ 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 [17]. 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 7.

Definition 7 (Execution tree [17]). *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 $B = (h, st, \mathsf{sb}, \mathtt{sl}, crt, \rho, \sigma)$ 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 7. *Consider a single-epoch execution $\mathcal{E}$ of* Ouroboros-Chronos$_{\mathsf{mc}}$ *with static registration and Δ-bounded skew, where Δ is the 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 (with respect to the objective 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 rounds later (again, with respect to the objective time) 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 6, as well as the requirement that $\mathsf{range}(\ell) = \{0\} \cup A$, are satisfied for the same reasons as given in [17, 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 `sl`, it will act based upon it and if it creates any block for slot `sl'`, its depth will be strictly larger than the depth of any block created by P for the slot `sl` by the description of the protocol. □

To maintain readability, in the following tratment we will omit the (negligible) failure probability caused by random-oracle collisions that are mentioned in the second statement of Lemma 7.

C.3 Protocol-Induced Distribution of the Characteristic String

Badertscher *et al.* [2] identified the following property of a characteristic string distribution to be of particular interest.

Definition 8 (The characteristic conditions [2]). *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* $k \geq 1$,

$$\begin{aligned}
&\Pr[W_k = \perp \mid W_1, \ldots, W_{k-1}] \geq (1 - f)\,,\\
&\Pr[W_k = 0 \mid W_1, \ldots, W_{k-1}, W_k \neq \perp] \geq \gamma\,, \textit{and hence}\\
&\Pr[W_k = 1 \mid W_1, \ldots, W_{k-1}, W_k \neq \perp] \leq 1 - \gamma\,.
\end{aligned}$$

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 $\mathsf{Ouroboros\text{-}Chronos_{mc}}$ satisfies $(f; \gamma)$-characteristic conditions for parameters recorded in the next lemma.

Lemma 8 (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 [2, Corollary 2] with respect to logical slots rather than objective-time rounds. □

C.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 [2, 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 9 and 10.

Definition 9 (Tines, length, and viability [2]). *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 10 (Divergence [26, 17]). *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*

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

We extend this notation to the fork F by maximizing over viable tines: $\mathrm{div}_\Delta(F) \triangleq \max_{t_1,t_2} \mathrm{div}(t_1, t_2)$, 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: $\mathrm{div}_\Delta(w) \triangleq \max_{F \vdash_\Delta w} \mathrm{div}_\Delta(F)$.

Theorem 4 ([2, 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*

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

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 8, 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 8, using the first statement of Lemma 7 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 [2, Corollary 4] (resp., [2, Corollary 5], [2, Lemma 11]) with respect to logical slots instead of the objective time, and using the first statement of Lemma 7 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

$$s = 48\widetilde{\Delta}/(\epsilon\beta f) \qquad \text{and} \qquad \tau = \beta f/16 \tag{9}$$

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 $\mathsf{Ouroboros\text{-}Chronos_{mc}}$ 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

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

Corollary 5 (Existential Chain Quality). *Let $W = W_1, \ldots, W_r$ denote the characteristic string induced by the protocol $\mathsf{Ouroboros\text{-}Chronos}$ 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 $s \geq 12\widetilde{\Delta}/(\epsilon\beta f)$,

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

C.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 B.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 [2] 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

$$\bar{\epsilon}_{\mathsf{mv}} \triangleq \exp\left(\ln R - \Omega(k)\right) + \bar{\epsilon}_{\mathsf{CG}}(\beta f/16, k/(4f)) + \bar{\epsilon}_{\exists\mathsf{CQ}}(k/(4f)) + \bar{\epsilon}_{\mathsf{CP}}(k\beta/64) \,, \tag{10}$$

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

Proof (sketch). The corollary follows by the same reasoning as Theorem 2 in [2], again by using the first statement of Lemma 7 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.

D Further Analytic Details and Omitted Proofs

D.1 The Proof of Lemma 1

Proof (of Lemma 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

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

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. □

D.2 The Proof of Lemma 2

Proof. We first establish that under the lemma assumptions, the following holds:

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

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

$$\mu(\mathsf{P}_i, \mathtt{SB}) = (\mathsf{slotnum}(\mathtt{SB}) - t) - \delta_{\mathsf{P}_i,\mathtt{SB}} \,, \tag{11}$$

where t is the objective time in which $\mathtt{SB}$ was sent, and $\delta_{\mathsf{P}_i,\mathtt{SB}} \in [\Delta]$ is the number of (objective) rounds that $\mathtt{SB}$ was delayed in its transit from $\tilde{\mathsf{P}}$ to P_i, establishing (ii). Note that (11) 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 (11) is equal to $\mathsf{Skew}^{\tilde{\mathsf{P}}}[\mathsf{slotnum}(\mathtt{SB})]$.

Given the above properties, we finish the proof by invoking Lemma 1 for the tuples

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

for two arbitrary alert parties P_1 and P_2. By property (i) both tuples are of the same size, and by property (ii) they satisfy the Δ-bound required by Lemma 1. Therefore we obtain

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

and since for each $i \in \{1,2\}$ we have $\mathsf{med}\left(\left(\mu(\mathsf{P}_1, \mathtt{SB})\right)_{\mathtt{SB} \in \mathcal{S}_i^{\mathsf{P}_1}} \right) = \mathsf{Skew}^{\mathsf{P}_i}[\mathtt{sl}+1]$, Lemma 2 follows. □

D.3 The Proof of Lemma 3

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 $\widetilde{\Delta} \leq R/6$ and by the first statement of Lemma 7. 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 from equation (11) for $\mu_{\mathsf{P},\mathtt{SB}_1}$ and $\mu_{\mathsf{P},\mathtt{SB}_2}$ where we use $\delta_{\mathsf{P},\mathtt{SB}_1} \leq \Delta$ and $\delta_{\mathsf{P},\mathtt{SB}_2} \geq 0$, 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 $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$. □

D.4 The Proof of Theorem 2

Proof (of Theorem 2, 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 SyncProc in the last slot of every epoch.

This proof partly follows the treatment in Section 5 of [17] 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 $\mathcal{F}_{\mathsf{INIT}}$. More precisely, $\mathcal{F}_{\mathsf{INIT}}$ ensures $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ for the slot $\mathtt{sl} = 1$, but $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$ is always maintained within an epoch as all alert parties advance their local clock by exactly one slot per objective round. 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 2 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 [17, 2], 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 first (and hence all) slot 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 for $\mathtt{ep}+1$ to obtain guarantees on CP, CG, and CQ and complete the induction step.

Finally, the bound (7) is obtained by instantiating (6) with the concrete bounds of Theorem 1. □

D.5 The Proof of Lemma 4

Proof (sketch). Notice that the chain-selection procedure maxvalid-bg given in Fig. B.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. B.7. 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 [2]. 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 t_{join} it 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 [2, 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 t. Moreover, since $\mathsf{P}_{\mathsf{join}}$ is still holding $\mathcal{C}_{\mathsf{join}}$ at round 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 t satisfies $\mathcal{C}'^{\lceil k}_{\mathsf{virt}} \preceq \mathcal{C}_{\mathsf{join}}$. Finally, if any alert party $\mathsf{P}_{\mathsf{alert}}$ held in round t' a chain $\mathcal{C}_{\mathsf{alert}}$ that would violate $\mathcal{C}_{\mathsf{alert}}^{\lceil k} \preceq \mathcal{C}_{\mathsf{join}}$, by our network assumption it would be seen by $\mathsf{P}_{\mathsf{virt}}$ by round t and hence adopted, concluding the proof. □

D.6 The Proof of Lemma 5

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.

For clarity, 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 $t_{\mathsf{start}}^{(j)}$ denote the (objective) round 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

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

thanks to the fact that there is a synchronization interval starting at the first slot of every R-slot epoch, the bound of Lemma 3 and the assumptions Skew_Δ and (5).

Let now i^* denote the minimal i such that $t_{\mathsf{start}}^{(i)} \geq t_{\mathsf{join}} + t_{\mathrm{off}} + t_{\mathrm{pre}}$, i.e., $t_{\mathsf{start}}^{(i^*)}$ occurs at least t_{pre} rounds *after* the beginning of $\mathsf{P}_{\mathsf{join}}$'s Phase C. According to (12) we have

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

by the values of t_{pre} and t_{minSync} (cf. Table 1). Therefore $t_{\mathsf{start}}^{(i^*)}$ 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 $\mathsf{JoinProc}$. This follows as by round $t_{\mathsf{start}}^{(i^*)}$, $\mathsf{P}_{\mathsf{join}}$ has been already recording the timestamps of all received synchronization beacons for t_{pre} rounds, and hence, has recorded into its $\mathsf{Timestamp}$ 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 $\mathsf{JoinProc}$. 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 2: 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 network delay, as in (12)). Assuming $\mathsf{CG}(\tau_{\mathsf{CG}}, R/3)$ and $R \geq 3k\tau_{\mathsf{CG}}$ (which follows from (5)), 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 4, 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 a similar reasoning as claim (ii) in the proof of Lemma 2, and relies on our network model guarantees. ◻

D.7 The Proof of Theorem 3

Proof (of Theorem 3, 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 $\mathcal{G}_{\mathrm{TICK}}$. 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 4.4, just like newly joining parties. The claims (b) and (c) of Lemma 5 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 Lemma 1 in the same way as done for synchronized parties in Lemma 2 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 C, which was designed in [2] 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. □

E Realization of the Ledger Functionality with Export-Clock Extension (in the $\mathcal{G}_{\mathbf{tick}}$-world)

To realize the ledger functionality, we must among other things relate the global reference time—objective rounds that—to the logical time experience by the protocol to translate chain growth into liveness guarantees by the ledger. We do this in Section 4.6. We then move on to the overview of the ledger and the security statement in Section E.1.

E.1 The Ledger Functionality and Security Theorem

As in [2], 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, but complex enough to provide a sufficiently detailed clock. Compared to [2], we have thus two major differences:

1. The ledger uses $\mathcal{G}_{\text{TICK}}$ to maintain its 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 [2] builds upon the general functionality [3]. 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 $\mathsf{predict\text{-}time}(\cdot)$ to ensure that the ideal world execution advances with the same pace (relatively to the pacemaker) as the protocol.

Any party can input a transaction to the ledger; upon reception transactions are validated using a predicate $\mathsf{Validate}$ 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 $\vec{N}_i$, is mapped to the ith state block via function $\mathsf{Blockify}(\vec{N}_i)$. $\mathsf{Validate}$ and $\mathsf{Blockify}$ are two of the ledger's parametrization algorithms. The third algorithm is the predicate $\mathsf{predict\text{-}time}$ 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 $\mathsf{FinishRound}$ in the protocol.

Party sets maintained by the ledger. In accordance with the classification of parties in Section 3.2, 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, except perhaps the GRO, until the current time. Formally, here, "sufficiently long" refers to $\texttt{Delay}$-many rounds, where $\texttt{Delay}$ 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 $\texttt{state}$ 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 [2] except that we establish a slightly better bound for transaction liveness guarantees. In nutshell, the basic mode of operation of $\mathsf{ExtendPolicy}$ 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, $\mathsf{ExtendPolicy}$ 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 G.2. The ledger functionality and its $\mathsf{ExtendPolicy}$ are further provided in the appendix for the sake of self-containment.

The Export-Time Extension. We introduce the export time extension to $\mathcal{G}_{\text{LEDGER}}$. The requirement on the given guarantees (to alert parties) should replace the need for a global clock by external protocols.

We introduce a generic extension to the basic ledger functionality: first, we represent a times-stamp $\texttt{time}_{\mathsf{P}}$ associated to party P as a pair (e, t), where t is the actual time stamp, and e refers to what we call a generation.[13] An alert party's time t in (e, t) is guaranteed to increase during a generation with every tick of the reference speed. Once t hits a generation boundary, defined as multiples of a generation length parameter R_L[14], the generation 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 $\mathsf{shiftLB}$, $\mathsf{shiftUB}$). 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 generations.

We give an overview of the parameters in Table G.2 and provide the functionality in Section E.2.

Security statement. Based on the above, we show that Ouroboros Chronos realizes the extended ledger functionality $\mathcal{G}_{\text{LEDGER}}$ with a concrete choice of parameters. The proof is deferred to Appendix E.4.

Theorem 5. *Let k be the common-prefix parameter, R the epoch-length parameter (constrained as required by Theorem 2) and Δ be the network delay. Let τ_{CG} be the chain growth coefficient as of Theorem 1, let $\tau_{\mathsf{CG},\text{glob}}$ be the derived (objective time) chain-growth coefficient as of Corollary 1, and let μ be the chain quality coefficient as of Theorem 1. Under the constraints[15] of Theorem 3, the protocol Ouroboros Chronos realizes the ledger functionality, i.e., there exists a simulator that simulates the protocol execution in the*

[13] In the real world, e would be the number of adjustments a party has made to its local clock.

[14] In this work, this is always exactly the length of normal epoch, i.e., $R_L = R$. However, the two concepts are logically different.

[15] 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 [3] and [2].

ideal world perfectly except with negligible probability in the parameter k for $R \geq \omega(\log k)$, for the ledger parameters

$$\texttt{windowSize} = k; \qquad \texttt{Delay} = t_{\text{join}}; \qquad \texttt{Delay}_{\texttt{tx}} = 2\Delta;$$
$$\texttt{maxTime}_{\texttt{window}} \geq \frac{\texttt{windowSize}}{\tau_{\mathsf{CG}} \cdot \tau_{\mathsf{CG},\text{glob}}}; \qquad \texttt{advBlcks}_{\texttt{window}} \geq (1-\mu)\texttt{windowSize},$$

and the clock-parameters

$$\mathsf{shiftLB} = -2\Delta; \qquad \mathsf{shiftUB} = \Delta; \qquad R_L = R$$
$$\texttt{timeSlack}_{\text{total}} = 2\Delta; \qquad \texttt{timeSlack}_{\text{ep}} = \Delta,$$

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

E.2 The Formal Description of the Functionality

For completeness, we describe here the ledger functionality in the $\mathcal{G}_{\text{TICK}}$ 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 a generation number and t is the actual timestamp. Initially, the generation 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 generation 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.

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 $\texttt{state}$, $\texttt{NxtBC}$, $\texttt{buffer}$, τ_L, and $\vec{\tau}_{\texttt{state}}$, as described above. The variables are initialized as follows: $\texttt{state} := \vec{\tau}_{\texttt{state}} := \texttt{NxtBC} := \varepsilon$, $\texttt{buffer} := \emptyset$, $\tau_L = 0$. 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 $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$).

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 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 $\mathcal{G}_{\text{TICK}}$ whenever $\mathcal{H} \neq \emptyset$.

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 $\mathcal{G}_{\text{TICK}}$ and stores the variable internally as $\texttt{lastTick}$ 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: If during round $\tau_L = 0$, the ledger did not receive a registration from each initial stakeholder, i.e., $\mathsf{P} \in \mathcal{S}_{\text{initStake}}$, the functionality halts.

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

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 $(\textsc{clock-read}, \text{sid}_C)$ to $\mathcal{G}_{\textsc{tick}}$ and upon receiving response $(\textsc{clock-read}, \text{sid}_C, \text{tick})$ 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,$\mathcal{G}_{\textsc{tick}}$, 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 $t \leq \tau_L$ 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 $|e - e'| \leq 1$. Otherwise, assign to $\texttt{time}_{\mathsf{P}}$ the 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 $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T || (I, \mathsf{P}, \tau_L)$;

   (b) Compute $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) := \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$ and if $\vec{N} \neq \varepsilon$ set $\texttt{state} := \texttt{state} || \mathsf{Blockify}(\vec{N}_1) || \ldots || \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\texttt{state}} := \vec{\tau}_{\texttt{state}} || \tau_L^\ell$, 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 $\texttt{time}_{\mathsf{P}} \leftarrow (e, t + d)$. If P is stalled and $t + d \operatorname{div} R_L > e$ 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, $\mathcal{G}_{\textsc{ledger}}$ executes the corresponding code from the following list:
   - *Submitting a transaction:*
     If $I = (\textsc{submit}, \text{sid}, \texttt{tx})$ 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 $(\textsc{submit}, \texttt{BTX})$ to $\mathcal{A}$.
   - *Reading the state:*
     If $I = (\textsc{read}, \text{sid})$ is received from a party $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$ with $\texttt{time}_{\mathsf{P}} = (e, t)$ s.t. $e \geq 0$ and $t \geq 0$, then return $(\textsc{read}, \text{sid}, \varepsilon$ to the requester. Else, set $\texttt{state}_p := \texttt{state}|_{\min\{\texttt{pt}_p, |\texttt{state}|\}}$ and return $(\textsc{read}, \text{sid}, \texttt{state}_p)$ to the requester. If the requester is $\mathcal{A}$ then send $(\texttt{state}, \texttt{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.
   - *Maintaining the ledger state:*
     If $I = (\textsc{maintain-ledger}, \text{sid}, \text{minerID})$ is received by an honest party $\mathsf{P} \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ then send $(\textsc{clock-update}, \text{sid}_C)$ to $\mathcal{G}_{\textsc{tick}}$. Else send I to $\mathcal{A}$. Additionally, before losing the activation, if $\mathsf{P} \in \mathcal{H} \setminus \mathcal{P}_{DS}$, parse $\texttt{time}_{\mathsf{P}}$ as (e, t) and if $t \operatorname{div} R_L > e$ then set $\texttt{time}_{\mathsf{P}} \leftarrow (e + 1, t)$ (and otherwise, do not update the timestamp).

- *The adversary proposing the next block:*
  If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \dots, \text{txid}_\ell))$ is sent from the adversary, update $\texttt{NxtBC}$ as follows:
  (a) Set $\text{listOfTxid} \leftarrow \epsilon$
  (b) For $i = 1, \dots, \ell$ do: if there exists $\texttt{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, U_j) \in \texttt{buffer}$ with ID $\text{txid} = \text{txid}_i$ then set $\text{listOfTxid} := \text{listOfTxid}||\text{txid}_i$.
  (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 $I = (\text{SET-SLACK}, (U_{i_1}, \widehat{\texttt{pt}}_{i_1}), \dots, (U_{i_\ell}, \widehat{\texttt{pt}}_{i_\ell}))$, with $\{\mathsf{P}_{i_1}, \dots, \mathsf{P}_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ 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 $I = (\text{DESYNC-STATE}, (U_{i_1}, \texttt{state}'_{i_1}), \dots, (U_{i_\ell}, \texttt{state}'_{i_\ell}))$, with $\{U_{i_1}, \dots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ 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 when the shift happens:*
  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 $\texttt{time}_\mathsf{P} \leftarrow \texttt{time}$

E.3 Extend Policy

For completeness, we state here the extend policy of [2] and mark in blue the minor modification. Note that the default block mechanism DefaultExtension is identical to [2] and thus omitted.

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

function ExtendPolicy($\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}}$)
  // First, create a default honest client block as alternative:
  $\vec{N}_{\texttt{df}} \leftarrow$ DefaultExtension($\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}}$) // Extension if adversary violates policy.
  Let τ_L be current ledger time (computed from $\vec{\mathcal{I}}_H^T$)
  // The function must not have side-effects: Only modify copies of relevant values.
  Create local copies of the values $\texttt{buffer}$, $\texttt{state}$, and $\vec{\tau}_{\texttt{state}}$.
  // Now, parse the proposed block by the adversary
  Parse $\texttt{NxtBC}$ as a vector $((\text{hFlag}_1, \texttt{NxtBC}_1), \cdots, (\text{hFlag}_n, \texttt{NxtBC}_n))$
  $\vec{N} \leftarrow \varepsilon$ // Initialize Result
  // Determine the time of the state block which is $\texttt{windowSize}$ blocks behind the head of the state

if $|\texttt{state}| \geq \texttt{windowSize}$ **then**
  Set $\tau_{\texttt{low}} \leftarrow \vec{\tau}_{\texttt{state}}[|\texttt{state}| - \texttt{windowSize} + 1]$
else
  Set $\tau_{\texttt{low}} \leftarrow 1$
end if
oldValidTxMissing $\leftarrow$ false // Flag to keep track whether old enough, valid transactions are inserted.
for each list $\texttt{NxtBC}_i$ of transaction IDs **do**
  // Compute the next state block
  // Verify validity of $\texttt{NxtBC}_i$ and compute content
  Use the txid contained in $\texttt{NxtBC}_i$ to determine the list of transactions
  Let $\vec{\texttt{tx}} = (\texttt{tx}_1, \dots, \texttt{tx}_{|\texttt{NxtBC}_i|})$ denote the transactions of $\texttt{NxtBC}_i$
  if $\texttt{tx}_1$ is not a coin-base transaction **then**
    return $\vec{N}_{\texttt{df}}$
  else
    $\vec{N}_i \leftarrow \texttt{tx}_1$
    for $j = 2$ to $|\texttt{NxtBC}_i|$ **do**
      Set $\texttt{st}_i \leftarrow \mathsf{blockify}_{B}(\vec{N}_i)$
      if $\mathsf{ValidTx}_{B}(\texttt{tx}_j, \texttt{state}||\texttt{st}_i) = 0$ **then**
        return $\vec{N}_{\texttt{df}}$
      end if
      $\vec{N}_i \leftarrow \vec{N}_i||\texttt{tx}_j$
    end for
    Set $\texttt{st}_i \leftarrow \mathsf{blockify}_{B}(\vec{N}_i)$
  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 $\texttt{BTX} = (\texttt{tx}, \text{txid}, \tau', \mathsf{P}) \in \texttt{buffer}$ of an honest party P with time $\tau' < \tau_{\texttt{low}} - \texttt{Delay}_{\texttt{tx}}$ **do**
      if $\mathsf{ValidTx}_{B}(\texttt{tx}, \texttt{state}||\texttt{st}_i) = 1$ but $\texttt{tx} \notin \vec{N}_i$ **then**
        oldValidTxMissing $\leftarrow$ true
      end if
    end for
  end if
  $\vec{N} \leftarrow \vec{N}||\vec{N}_i$
  $\texttt{state} \leftarrow \texttt{state}||\texttt{st}_i$
  $\vec{\tau}_{\texttt{state}} \leftarrow \vec{\tau}_{\texttt{state}}||\tau_L$
  // Must not proceed with too many adversarial blocks
  $i \leftarrow \max\{\{\texttt{windowSize}\} \cup \{k \mid \texttt{st}_k \in \texttt{state} \wedge \text{ proposal of } \texttt{st}_k \text{ had hFlag} = 1\}\}$ // Determine most
  // recent honestly-generated block in the interval behind the head.
  if $|\texttt{state}| - i \geq \texttt{advBlcks}_{\texttt{window}}$ **then**
    return $\vec{N}_{\texttt{df}}$
  end if
  // Update $\tau_{\texttt{low}}$: the time of the state block which is $\texttt{windowSize}$ blocks behind the head of the
  // current, potentially already extended state
  if $|\texttt{state}| \geq \texttt{windowSize}$ **then**
    Set $\tau_{\texttt{low}} \leftarrow \vec{\tau}_{\texttt{state}}[|\texttt{state}| - \texttt{windowSize} + 1]$
  else
    Set $\tau_{\texttt{low}} \leftarrow 1$
  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 $\tau_{\texttt{low}} > 0$ **and** $\tau_L - \tau_{\texttt{low}} > \texttt{maxTime}_{\texttt{window}}$ **then** // A sequence of blocks cannot take too much time.
  return $\vec{N}_{\texttt{df}}$
else if $\tau_{\texttt{low}} = 0$ **and** $\tau_L - \tau_{\texttt{low}} > 2 \cdot \texttt{maxTime}_{\texttt{window}}$ **then** // Bootstrapping cannot take too much time.

```
        return N⃗_df
    else if oldValidTxMissing then // If not all old enough, valid transactions have been included.
        return N⃗_df
    end if
    return N⃗
end function
```

E.4 The Proof of Theorem 5

Proof (of Theorem 5). For the simulation part, we observe that compared to the simulator of the protocol Ouroboros Genesis in [2], 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 E.5.

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 $\mathsf{BAD\text{-}CP}, \mathsf{BAD\text{-}CQ}, \mathsf{BAD\text{-}CG}$, and $\mathsf{BAD\text{-}TIME\text{-}RANGE}$ 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 $\text{hFlag} = 1$) (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.) This 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 3 it is clear that $\mathsf{shiftLB} = -2\Delta$ and $\texttt{timeSlack}_{\text{ep}} = \Delta$ by Lemma 2. 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 3 as done in Lemma 9. By the preceeding analysis, the probability that any constraint is violated, and thus $\mathsf{BAD\text{-}TIME\text{-}RANGE}$ 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 3. Let* $\texttt{sl}$ *be the synchronization slot of epoch* $\texttt{ep}$ *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* $\mathsf{Skew}_\Delta[\texttt{sl}]$ *is not violated in this execution. Then it holds that*

1. *The shift* shift *party* P *computes is upper bounded by the maximal recommendation* $\mathsf{recom}(SB)$, $SB \in \mathcal{S}_i^{\mathsf{P}}$ *for which* $\mathsf{slotnum}(SB) = s$ *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 time stamp in this round.*

Proof (Sketch). The first part of the claim follows from the observation that $\mathsf{Timestamp}(\texttt{SB})$ recorded by P is received after P' created the beacon. Thus, the term $\mathsf{slotnum}(\texttt{SB}) - \mathsf{Timestamp}(\texttt{SB})$. If at the objective time

the beacon was created the creator reported slot s, 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. 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 3 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. 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 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 $\mathsf{slotnum}(\mathsf{SB}) - \mathsf{Timestamp}(\mathsf{SB})$. Analogous to the above case, if at the objective time the beacon was created the creator reported slot s, party P' local timestamp (used to measure arrival times in epoch ep) differs by at most d to the creator's timestamp, then the recommendation is lower bounded by $d - \Delta$. 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. ☐

E.5 Simulator used in the UC realization

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 [2] 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 $\mathcal{F}_{\mathsf{VRF}}$, $\mathcal{F}_{\mathsf{KES}}$, $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$, $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$, $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$, and $\mathcal{F}_{\text{N-MC}}^{\mathsf{sync}}$.
- 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 [2]).
- 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 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 (objective) 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 ($\mathsf{P}.\mathtt{isInit}$ becomes true).
- We assume that the simulator queries upon any activation for the sequence $\vec{\mathcal{I}}_H^T$, and the current time τ_L from the ledger. We note that the simulator is capable of determining $\mathsf{predict\text{-}time}(\cdot)$ of $\mathcal{G}_{\text{LEDGER}}$.

Messages from the Clock:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ from $\mathcal{G}_{\text{TICK}}$, 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 $\mathcal{A}$.

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 $(\mathsf{DefineOffset}, sid, o_1, \ldots, o_n)$, store the avlues and relay the ansower to the simulated instance of $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$.
- Upon receiving $(\text{SUBMIT}, \mathtt{BTX})$ from $\mathcal{G}_{\text{LEDGER}}$ where $\mathtt{BTX} := (\mathtt{tx}, \text{txid}, \tau, \mathsf{P})$ forward $(\text{MULTICAST}, \text{sid}, \mathtt{tx})$ to the simulated network $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ 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 $\vec{\mathcal{I}}_H^T$ 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 $\mathcal{S}_{\text{ledg}}^{\Delta}$ (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) $\mathcal{F}_{\mathsf{INIT}}^{\Delta}$ functionality and the adversary $\mathcal{A}$ acting on behalf of a corrupted party.
- If at time $\mathtt{localTime} = 0$, 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 $\mathtt{state}$ and execute $\text{AdjustView}(\mathtt{state}, \tau_L)$.
- 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 $\mathtt{state}$ and execute $\text{AdjustView}(\mathtt{state}, \tau_L)$.

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

- Upon receiving $(\textsc{multicast}, \text{sid}, m)$ 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 $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$, 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 $(\textsc{fetch}, \text{sid})$ from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ forward the request to the simulated $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\textsc{delays}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\textsc{swap}, \text{sid}, \text{mid}, \text{mid}')$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ and return whatever is returned to $\mathcal{A}$.

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

- Upon receiving $(\textsc{multicast}, \text{sid}, m)$ 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 $\mathcal{F}^{\mathsf{sync}}_{\text{N-MC}}$, 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 $(\textsc{fetch}, \text{sid})$ from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ behave analogously to above for $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
- Upon receiving $(\textsc{delays}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from $\mathcal{A}$ behave analogously to above for $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
- Upon receiving $(\textsc{swap}, \text{sid}, \text{mid}, \text{mid}')$ from $\mathcal{A}$ behave analogously to above for $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.

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

procedure SimulateMaintenance(P, `localTime`)
    Simulate the (in the UC interruptible manner) the maintenance procedure of party P as in the protocol in round τ_L when the party reports localtime P.`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 AdjustTime(τ_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 ExtendLedgerState(τ_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 AdjustTime(τ_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 $(\textsc{clock-update}, \text{sid}_C, \mathsf{P})$ 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 `localTime` **do**
    Let $\mathcal{C}_{\mathsf{loc}}^{(\mathsf{P})}$ 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 τ_L.
    If $\rho^{(\mathsf{P})} > 0$ then let $\mathcal{C}_1^{(\mathsf{P})}, \dots, \mathcal{C}_k^{(\mathsf{P})}$ be the chains contained in the receiver buffer $\vec{M}^{(\mathsf{P})}$ 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 $\vec{\mathtt{st}}_{\mathsf{P}}$.
  end for
  Let $\vec{\mathtt{st}}$ be the longest state among all such states $\vec{\mathtt{st}}_{\mathsf{P}}$, $\mathsf{P} \in \mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ from above.
  Compare $\vec{\mathtt{st}}^{\lceil k}$ with the current state `state` of the ledger
  if $|\mathtt{state}| > |\vec{\mathtt{st}}^{\lceil k}|$ **then** // Only pointers need adjustments
    Execute AdjustView(`state`)
  end if
  if `state` is not a prefix of $\vec{\mathtt{st}}^{\lceil k}$ **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. $\mathtt{state}||\mathsf{diff} = \vec{\mathtt{st}}^{\lceil k}$.
  Parse $\mathsf{diff} := \mathsf{diff}_1 || \dots || \mathsf{diff}_n$.
  for $j = 1$ to n **do**
    Map each transaction `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 $\mathsf{list}_j = (\text{txid}_{j,1}, \dots, \text{txid}_{j,\ell_j})$ 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 $\text{hFlag} = 0$ 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($\mathtt{state}||\mathsf{diff}$)
end procedure

procedure AdjustTime(P, τ_L)
  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 $\mathtt{time}_{\mathsf{P}}$ and send $(\text{SET-TIME}, \text{sid}, \mathsf{P}, \mathtt{time}_{\mathsf{P}})$ 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 $\mathsf{P}.\mathtt{localTime} \bmod R = 0$ **then**
    Take the simulated timestamp $\mathtt{time}_{\mathsf{P}}$ 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

F 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.

F.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 generation boundaries. However, the behavior is limited and predictable based on the clock parameters. For example, parties could stall their operations until the generation boundary switched 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 $\texttt{timeSlack}_{\text{ep}}$.

F.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 $\mathcal{G}_{\text{TICK}}$), then each alert party reports time $t = \tau - m \cdot e$ (where e is the generation number) and thus the clock parameters of the ledger would in this case be $\mathsf{shiftLB} = \mathsf{shiftUB} = -m$ and $\texttt{timeSlack}_{\text{ep}} = 0$, $\texttt{timeSlack}_{\text{total}} = m$. Even more, in this case, any external protocol can recompute the objective time (as recorded by $\mathcal{G}_{\text{TICK}}$) 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 $\texttt{timeSlack}_{\text{ep}} = 2d$, $\texttt{timeSlack}_{\text{total}} = m+3d$. 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 (as recorded internally by $\mathcal{G}_{\text{TICK}}$) 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.

G Summaries of main protocol state variables and parameters

G.1 Overview of the main state variables of Ouroboros Chronos.

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 $\mathcal{G}_{\text{TICK}}$. 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.
$T_{\mathsf{P}}^{\mathsf{ep}}$	The threshold of this party to evaluate slot leadership 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.

G.2 Overview of main ledger elements such as parameters and state variables.

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 $\mathsf{Validate}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) := \mathsf{ValidTx}_{\mathsf{OC}}(\mathtt{tx}, \mathtt{state})$.
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 from the honest inputs when the protocol will call FinishRound after finishing its round operations.
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**
$\mathtt{maxTime}_{\mathtt{window}}$	Minimal Growth: In $\mathtt{maxTime}_{\mathtt{window}}$ 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.
$\mathtt{advBlcks}_{\mathtt{window}}$	A limit $\mathtt{advBlcks}_{\mathtt{window}}$ 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.
$\mathtt{Delay}_{\mathtt{tx}}$	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**
$\mathtt{time}_{\mathsf{P}}$	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 generation boundaries
R_L	The parameter characterizing generation boundaries (similar to epochs): 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 generation.
$\mathtt{timeSlack}_{\mathrm{total}}$	An upper bound between t and t' of two synchronized parties P and P' reporting (e, t) and (e', t') as their respective time $\mathtt{time}_{\mathsf{P}}$ and $\mathtt{time}_{\mathsf{P}'}$, respectively.
$\mathtt{timeSlack}_{\mathrm{ep}}$	An upper bound between t and t' of two synchronized parties P and P' reporting (e, t) and (e', t') as their respective time $\mathtt{time}_{\mathsf{P}}$ and $\mathtt{time}_{\mathsf{P}'}$, respectively whenever $e = e'$

Fig. 5. Overview of main ledger elements such as parameters and state variables. As in [3, Definition 2], we always assume that $\mathsf{blockify}_{\mathsf{OC}}$ and $\mathsf{ValidTx}_{\mathsf{OC}}$ do not disqualify each other.