Dynamic Ad Hoc Clock Synchronization

Citation for published version:
Badertscher, C, Gaži, P, Kiayias, A, Russell, A & Zikas, V 2021, Dynamic Ad Hoc Clock Synchronization. in *Advances in Cryptology – EUROCRYPT 2021.* Lecture Notes in Computer Science, vol. 12698, Springer, pp. 399-428. https://doi.org/10.1007/978-3-030-77883-5_14

Digital Object Identifier (DOI):
10.1007/978-3-030-77883-5_14

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Advances in Cryptology – EUROCRYPT 2021

Dynamic Ad Hoc Clock Synchronization

Christian Badertscher[1⋆], Peter Gaži[1], Aggelos Kiayias[1,2⋆⋆], Alexander Russell[1,3⋆⋆⋆], and Vassilis Zikas[4†]

[1] IOHK – `firstname.lastname@iohk.io`
[2] University of Edinburgh – `aggelos.kiayias@ed.ac.uk`
[3] University of Connecticut – `acr@cse.uconn.edu`
[4] Purdue University – `vzikas@cs.purdue.edu`

Abstract. Clock synchronization allows parties to establish a common notion of global time by leveraging a weaker synchrony assumption, i.e., local clocks with approximately the same speed. Despite intensive investigation of the problem in the fault-tolerant distributed computing literature, existing solutions do not apply to settings where participation is unknown, e.g., the ad hoc model of Beimel *et al.* [EUROCRYPT 17], or is dynamically shifting over time, e.g., the fluctuating/sleepy/dynamic-availability models of Garay *et al.* [CRYPTO 17], Pass and Shi [ASIACRYPT 17] and Badertscher *et al.* [CCS 18].

We show how to apply and extend ideas from the blockchain literature to devise synchronizers that work in such dynamic ad hoc settings and tolerate corrupted minorities under the standard assumption that local clocks advance at approximately the same speed. We discuss both the setting of honest-majority hashing power and that of a PKI with honest majority. Our main result is a synchronizer that is directly integrated with a new proof-of-stake (PoS) blockchain protocol, Ouroboros Chronos, which we construct and prove secure; to our knowledge, this is the first PoS blockchain protocol to rely only on *local* clocks, while tolerating worst-case corruption and dynamically fluctuating participation. We believe that this result might be of independent interest.

1 Introduction

Global clock synchronization [13,24,19] allows a set of mutually distrustful parties to approximate a global notion of "time," in such a manner that if some party believes that the global time is t then every party believes it to be $t \pm \epsilon$ for some small $\epsilon > 0$. This allows for an (approximately) synchronous (or partially synchronous) execution of distributed protocols which has placed the study of such

[⋆] Work done while the author was at the University of Edinburgh, Scotland.

[⋆⋆] Research partly supported by EU Project No. 780477, PRIVILEDGE.

[⋆⋆⋆] This material is based upon work supported by the National Science Foundation under Grant No. 1717432.

[†] Work done in part while the author was at the University of Edinburgh and while visiting the Simons Institute for the Theory of Computing, UC Berkeley. Work supported in part by IOHK.

synchronizers at a prominent position in theoretical computer science research. A number of works investigated feasibility across the spectrum of security/adversary models—from perfect to computational security and for different types of network synchronization assumptions [13,24,19,15,14,2,33,25,28,32]. We defer a full description of the current landscape of feasibility to the full version of this work [4] due to space constraints. The common assumption of such synchronizers is that the (honest) parties have local (initially desynchronized) clocks which advance at (roughly) the same speed.

Notwithstanding, existing synchronization techniques rely on accurate knowledge of the total number of parties present in the system and smart counting of received messages (or message chains). Consequently these techniques are inapplicable in the *ad hoc secure multi-party computation* setting of Beimel *et al.* [6], where the universe of parties is known but not all parties participate in the protocol and the identities of those that do participate are not known to the other parties. As discussed in [6], what makes this model challenging is the fact that it aims for non-interactive secure computation in the private simultaneous message (PSM) model [16,20]. Indeed, if one allows multiple rounds of interaction, then the parties assumed to be online can try to figure out the active identities, taking the problem's difficulty away.

In this work we study a synchronization challenge which arises in the *dynamic* variant of the ad hoc model, where not only the parties do not know who is actually playing the protocol, but the set of active participants might change in every round (this change is further allowed to be under adversarial control). This is not only a natural extension of [6] but is also motivated by real-world considerations in the blockchain setting. Indeed, the *sleepy model* of consensus proposed by Pass and Shi [31]—and later generalized in the UC setting [8,9] by Badertsher *et al.* [3] under the term *dynamic availability*—puts forth such a *dynamic ad hoc* model for capturing participation fluctuation in distributed ledger protocols. In a nutshell, these works allow for parties to (re)join the protocol at any time and to temporarily *sleep*—i.e., drop out of (certain processes of) the protocol —according to an arbitrary (or even adversarial) sleep pattern.

This dynamic ad hoc setting limits the power of existing synchronization techniques, since the lack of agreement of participation patterns makes counting ineffective for taking consistent decisions. The lack of such synchronization makes any distributed cryptography primitive [1] in this dynamic ad hoc setting reliant on a (possible imperfect) global notion of time. In fact, even the formal cryptographic analyses of proof-of-work (PoW) and proof-of-stake (PoS) blockchains have typically assumed a (partially) synchronous model with a notion of global time. For instance, standard references for the proven security of Bitcoin [17,18,29] implicitly use the fact that they can refer to a global round index in order to prove the desired properties of the protocol. Indeed, the common-prefix property is defined to require that if an honest party holds a chain at round ρ, then the prefix of this chain—obtained by removing the k most recent blocks—will eventually become prefix of the chain of any honest party (at some round $\rho' \geq \rho$). The assumption was made explicit in [5] by assuming a global clock in the global

UC setting [9]: this permits every party to query a common clock on demand and from that deduce the current round. A similar approach, assuming access to a global clock, was also adopted in the constructions of PoS blockchains, such as Sleepy Consensus [31], Snow White [11], and Ouroboros [23,12,3].

The natural question that we address in this work is the following: *Is global clock synchronization from standard assumptions possible in the dynamic ad hoc setting?* By "standard assumptions" in the above question we mean the common assumptions underlying traditional synchronizers—that is, local (initially desynchronized) clocks which advance at (roughly) the same speed and an honest majority of parties.[5] —along with standard cryptographic assumptions such as a public-key infrastructure (PKI) and existentially unforgeable digital signatures.

As discussed above, counting arguments of the sort used in classical synchronizers does not seem to help. Therefore, to answer this question we turn to techniques from the cryptographic literature on blockchain ledgers, which has already come a long way in addressing other security challenges that the dynamic ad hoc model creates. In fact, it is not hard to verify that in a resource-restricted scenario, such as the one created by assuming honest majority of hashing power, the above question can be answered by relying on a simplified version of the Bitcoin backbone protocol [17]. In particular, one can observe that the description of the Bitcoin blockchain (without difficulty recalibration) can rely on a purely *execution-driven* notion of time and explicit knowledge of current global time is not required. In the static difficulty setting, proving security in this way follows immediately from [17,29]. As a result, a synchronizer can be trivially inferred by defining the clock to be the current blockchain length in each party's local state.

The above observation is a good indication that blockchain techniques can help answering our question, but it unfortunately does not provide a satisfying answer, as it relies on a non-standard—from the perspective of synchronizers and/or general multi-party computation (MPC) literature—assumption, i.e., that the honest parties control the majority of the computing power per unit of time. To avoid such non-standard assumptions we turn to proof of stake (PoS). Here, an execution-driven notion of time similar to the aforementioned notion achieved by Bitcoin without difficulty recalibration can actually be achieved by certain PoS-based iterated-Byzantine Fault Tolerant (iBFT) ledger protocols such as Algorand [10]. Indeed, given access to the genesis block (which can be seen as an initial PKI) a party can use the index (sequence number) of the current block as global time. This suggests the following as a solution to our synchronization problem: The assumed PKI—which in the ad hoc model would include the keys of all parties, active or not—is interpreted as a genesis block where every key is associated with a unit of stake. Then a simplified version of the Algorand ledger protocol is executed, i.e., without any stake shift and where the contents of the blocks are independent messages, in particular they are not interpreted as transactions of any kind. Whenever a party becomes active in the computation,

[5] In the static ad hoc setting [6], this assumption becomes honest majority of active parties; and in the dynamic, it would be honest majority among the parties that are actively participating in any given round.

he uses the length of the blockchain as his global time. If a $(2/3+\epsilon)$-majority of active parties is honest (for some constant $\epsilon > 0$), it follows that in the above execution of simplified-Algorand, a $(2/3+\epsilon)$-majority of the (implicit) stake must be in honest hands and therefore security follows by the security proof of Algorand. It is not hard to verify that the protocol yields a good synchronizer, where, not surprisingly, the network delay lower-bounds the maximum skew of synchronized parties' clocks. However, the above solution works only under a concession which severely limits the nature of the dynamic ad hoc model. We need to demand explicit participation thresholds that are part of the protocol logic. Stated differently, each protocol participant at any given time must be aware of a sufficiently accurate estimate of how many parties are active at that time. To our knowledge, such a property, which in [10] is referred to as *lazy honesty*, is necessary for the security analysis of [10]. We note in passing that such a rigid participation restriction is not necessary for the Bitcoin blockchain or its PoS variants in the sleepy/dynamic-availability setting.

Although it does not solve our question, the above idea still points to the right direction: Concretely, if we could use the above idea but with a PoS protocol which does not rely on explicit participation bounds, e.g., [31,11,23,12,3], then we would have answered our question to the affirmative. And even better, our synchronizer would work assuming an honest majority ($1/2+\epsilon$ of parties for some suitably chosen ϵ), since the above protocols are secure w.r.t. such an assumption on the stake distribution. Unfortunately, unlike Algorand, these protocols use a notion of (approximate) global time hardwired in the protocol logic, and the protocol is unspecified without such global knowledge of time/round. In fact, as explained below, there does not seem to be a simple way to removing this dependence of global time, and replace it by local clocks —even, perfectly-coordinated ones that advance at exactly the same speed— while preserving the security guarantees.[6] The reason is that in these PoS blockchains a party's right to create a block is always associated with a concrete round (also called "slot"), and in order to verify that a block is created by an eligible party, that party must include a proof explicitly referring to the slot number. This means that a new party that joins the blockchain—or one that has been sleeping for long—cannot prune-off chains with adversarial timestamps so that it eventually adopts the right chain. Thus if a new party with an incorrect local time joins the protocol and sees a chain that includes blocks which appear to be far in the future (according to her local time), she cannot decide whether the chain is adversarial—in which case she needs to ignore or truncate it—or her local time is far behind absolute time. It is worth adding that these are not merely theoretical considerations: in a real world deployment the dependency on a global clock is typically met by using a global time synchronization service such as NTP [27] and hence the security of

[6] Of course, one could include such a notion of (approximate) global time in a trusted *checkpointing assumption* [11], but this defeats the purpose of decoupling the protocol from an explicitly assumed trusted source of global time when a party (re)joins, which is the main challenge of our work.

all these protocols becomes compromised if such service fails to deliver a truly reliable clock, a possibility that cannot be excluded [26].

Note that all previous PoS protocols which can operate in a participation-unrestricted setting [31,11,23,12,3] require an upper bound on the network delay Δ which is a necessary assumption, see [30], due to the participation uncertainty. However, knowledge of an upper bound on Δ does not help the parties in any direct way to assess the actual time (e.g., by locally counting time intervals of length Δ), as participation gaps can invalidate their local timer with respect to the implicit global execution-driven time.

It seems we have hit a deadlock: if the protocol itself crucially relies on global time, then how can it be used to remove global time and replace it with loosely synchronized clocks? Unfortunately, there seems to be no way to use these blockchain protocols (or their properties) in a black-box manner to realize a global clock from standard assumptions. Nonetheless, as we prove here, we can draw inspiration from these works to design new a new PoS blockchain protocol *from scratch*, so that it *does not rely on a global clock* and can be used as a synchronizer to obtain an approximate global clock from standard synchronizer assumptions. Our approach to building the new blockchain extends in a highly non-trivial manner ideas from the recent PoS literature.

Our blockchain protocol works not only for static stake, but can even accommodate stake transfers and new keys being generated (and potentially allocated stake) as in existing PoS blockchains. Thus, we actually not only solve the synchronizer problem in the dynamic ad hoc setting, but we provide the first full fledged PoS blockchain in the dynamic availability setting which relies *not* on global time but on the weaker and more realistic assumption of local (initially desynchronized) clocks which advance at (roughly) the same speed. We believe that this result might be both of independent interest for the distributed ledgers literature as well as of practical importance. We note in passing that given our new synchronizer, a potential alternative construction of a blockchain would be to use it in a black-box way to first realize a global clock, and then use this within an existing PoS blockchain. However, this would yield a highly suboptimal use of resources as it would effectively mean running two blockchains. This works shows that one does not need this redundancy and use our construction both as a PoS blockchain and as means to simulate a global clock (and potentially export it to other calling protocols) at the same time.

2 Overview of Our Techniques

At the core of our global synchronization procedure is a new PoS blockchain ledger protocol which (1) does not rely on global clocks but merely on local clocks with (approximately) the same speed, (2) accommodates dynamic ad hoc participation, and (3) assigns timestamps to each block so that they can be used by any external observer to deduce an (approximate) notion of global time/round (see Theorem 1). We refer to this new blockchain protocol as *Ouroboros Chronos*, or simply *Chronos*, and discuss it below. As discussed above, it would be sufficient for our synchronizer's needs to just design a blockchain that works in the static stake setting. Nonetheless, for full generality, we design Chronos to accommodate (and tolerate) stake-shift, which makes it the first fully-functional PoS blockchain, yielding the same guarantees as existing ones [10,11,3], but without reliance on a global clock or restricting dynamic participation.

First, observe that if all the parties running the blockchain protocol would be guaranteed to be around from its beginning and throughout its lifetime (i.e., in the static ad hoc model of [6]) then one could use an existing PoS blockchain for honest majority, e.g., [11,3] with the convention described above to assign one unit of stake per public key. A synchronizer could be derived from the length of the blockchain while the security assumptions and parties never joining or leaving the system would guarantee that parties stay synchronized. What makes the problem challenging and excludes the above solution is, thus, the combination of lack of a global clock with dynamic (ad hoc) participation. In the following we focus on how to redesign the mechanism of the above PoS protocols to allow (re)joining parties to get in sync with parties that have been around sufficiently long and are, therefore, already in-sync with each other—we refer to these latter parties as *alert*.

The central idea of our mechanism is the continuous recording of individually submitted clock readings and the clock adjustment of the alert parties' local clocks based on these readings at regular recalibration points. This mechanism is based on a VRF-based probabilistic sampling of the local clocks of all active parties using the blockchain to consistently record this operation over the protocol execution. As we demonstrate, this opens the opportunity for a safe (re)joining procedure; newly joining parties will be able to "hook" themselves into the next recalibration point and become fully alert.

In more details, here is how our new (re)joining procedure works: (Re)joining parties, start with listening on the network for some time, collecting broadcasted chains and following a "densest chain" chain-selection rule similar to [3]. Informally, this rule mandates that if two chains $\mathcal{C}$ and $\mathcal{C}'$ start diverging at some time t—according to the reported time-stamps in $\mathcal{C}$ and $\mathcal{C}'$—then choose the chain which is denser in a sufficiently long interval after that time. Our first key observation is that this rule offers a useful (albeit in itself insufficient) guarantee in our setting: the joining party will end up with some blockchain that, although *arbitrarily long*, is at worst forking from a chain held by an honest and already synchronized party by a bounded number of blocks (equal to a security parameter) with overwhelming probability. This observation is the key to start building our synchronization mechanism. More concretely, we prove that the above process guarantees to eventually prune-off all chains with bad prefixes, i.e., prefixes that do not largely coincide with the prefixes of the other already synchronized honest parties' chains. In fact, as we show, the parties can compute an upper bound on the time (according to their local clocks) they need to remain in the above self-synchronization state before they build confidence in the above guarantee, i.e., before they know that their locally held chain is consistent with a long and stable prefix that already-synchronized honest parties adopt.

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

Designing and analyzing such an updating process is challenging. Indeed, consider the following straw man attempt: The party resets its local clock so that the time reported in, say, the last block of the prefix is the time this block was received. Before discussing the limitations of this proposal, let us first discuss an inherent property when dealing with clock synchronization in the setting with Δ-bounded (but adversarially controlled) delay networks. A message received by a party might have been sent up to Δ rounds before, hence the time that the party will set its clock to might be up to Δ rounds away from the clock of the sender (at the point of update). This delay-induced imprecision is unavoidable, so when we assess a given proposal we accept that clocks only need to be "loosely" synchronized; specifically, clocks of honest parties might differ by a bounded amount, where the bound is known and depends only on Δ. In fact, this relaxation is common and believed to be necessary even in the permissioned model [24,19].[7]

However, the above simple solution is problematic even when there are no delays: Although the chain that the newly joining party recovered is guaranteed to have a prefix consistent with the already synchronized honest parties, individual blocks might be originating from the adversary and therefore contain a time stamp very different from the true sending time of that block. To make matters worse, the rate of honestly generated blocks in a chain of an honest party can be quite low as implied by the known bounds of chain quality [18,12], and thus the time inaccuracy of any individual block can be significant.

A second attempt would be to have in every round (or at regular intervals) every party use the credentials of all the coins it owns to broadcast a signed timestamp, i.e., every party acts as a verifiable *synchronization (or timestamping) beacon* on behalf of all the coins it owns. The joining party receives all these broadcasted timestamps, and uses their majority to compute the value of its clock. Still, this solution has drawbacks: The first is scalability; this is not severe, as existing ideas can be employed such as using the protocol history as input to a verifiable random function (VRF) to identify eligible parties (or, as in the case of Algorand, by using Bracha-style committees [7]) to send timestamping beacons in every synchronization round. The second, harder problem is that in order to use the majority, the local clocks of the parties that report time need to be perfectly synchronized so that their majority agrees. If their clocks have any small drift, this fails. Furthermore, even with identical speed clocks, dynamic participation allows parties to drop off and rejoin, which means that, due to the network delay

[7] The model from [24] 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.

the honest parties will end up with only loosely synchronized local clocks. Using the average instead of the majority function does not help out here either since a single adversarial timestamp can throw off the average arbitrarily far. Hence, taking the median of the received timestamps promises to be more stable against extreme values. Observe that as long as synchronized honest parties' local clocks are not far apart, the times they report will be concentrated to a sufficiently small time interval, and the median will fall in this interval.

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

Our solution follows the spirit of the above conclusion. In a nutshell, we will use the VRF to assign timestamping-beacon parties to slots according to their state. Parties who are synchronized and active when their assigned slot is encountered will broadcast a timestamp and a VRF-proof of their eligibility for the current timeslot (together, we call this a *synchronization beacon*). And to agree on the set of eligible parties that will be used (including the dishonest ones) these beacons will also be included in the blockchain by the already synchronized parties, similarly to transactions. Any party who joins and tries to get synchronized will gather chains and record any broadcasted beacons (and keep track of the local time these were received). Once the party is confident it has a sufficiently long prefix of the honest chain, it will retrospectively use this gathered information to extract the agreed-upon set of beacons, compute a good approximation of the clocks parties had when they broadcasted these beacons and apply a median rule to set its local clock to at most a small distance from other honest and synchronized parties. In order to ensure that already synchronized parties adjust in tandem with joining parties we will have them also periodically execute the synchronization algorithm—but of course using their local blockchain, which they know is guaranteed to have a large common prefix with any other honest and synchronized party. Evidently, to turn this high-level idea of our solution into a provably secure protocol requires appropriate design choices that we present in Section 4. Nonetheless, by a careful analysis (cf. Section 5) we can show that not only the above construction yields a PoS blockchain that does not rely on global time, but, also, the reported timestamps are (approximately) consistent among long-term (alert) participants and can, with a suitable encoding mechanism, be used to devise a synchronizer satisfied the guarantees of the following theorem.

Theorem 1. *There is a synchronizer protocol in the dynamic ad hoc setting, so that the following properties hold:*

1. *(Completeness) Any alert party in the protocol reports some time $t \in \mathbb{N}$.*
2. *(Approximate synchrony) For any two alert parties p_1 and p_2 reporting times t_1 and t_2, respectively, it holds $|t_1 - t_2| \leq 2\Delta$, where Δ is an upper bound on the network delay.*
3. *(Monotonicity) If an alert party reports times t_1 and then t_2 at two consecutive steps[8] in its execution, then $t_1 \leq t_2 \leq t_1 + 2\Delta$.*
4. *(Liveness) For any alert party, if time t_2 is reported 2Δ local rounds after time t_1, then $t_1 < t_2$.*

Note that the above theorem provides a clock that might make "jumps" (i.e., skip some rounds for certain parties). However, these jumps are bounded by 2Δ. Hence, it is straightforward to turn this clock into a clock that does not make jumps (albeit slower) and where synchronized parties are within a round from each other: Every party reports time $\lfloor \frac{t}{2\Delta} \rfloor$, where t is the value it sees from the above "jumpy" clock.

3 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 $\mathcal{C}_2$.

We discuss the model and the hybrid functionalities assumed in the protocol below. The formal descriptions are given in the full version of this work [4].

Relaxed synchrony. The synchrony assumption that parties advance at exactly the same pace can be captured by the global-setup variant of the clock functionality from [22]. This is a weaker version of the global clock used in previous analyses of blockchains [5,3] in that it does not keep a counter representing the global system time, but rather maintains for each party (resp. ideal functionality) an indicator bit d_{P} (resp. $d_{(\mathcal{F},\mathrm{sid})}$) of whether or not a new round has started. Each party's indicator is accessible by a standard CLOCK-GET command. All indicators are set to 0 at the beginning of each round; once any party or functionality finishes its round it issues a CLOCK-UPDATE command that updates his indicator

[8] In this context, a step in the execution corresponds to the action(s) a party takes during a single local round (i.e., between two "ticks" of its local clock.)

to 1. Once every party and functionality has updated its indicator, the clock resets all of them to 0; this switch allows the parties to detect that the previous round has ended and move on to the next round.

Arguably the above clock offers very strong synchronization guarantees, since once a round switches, every party is informed about it in the next activation. In [22] a relaxed version of this clock was introduced which allowed the adversary to delay notifying the parties about a round switch by bounded amount of fetch-attempts. This behavior relaxes the perfect nature of the clock, but it still ensures that no party advances to a next round before all parties have completed their current round.

In this work we consider parties that advance at roughly the same speed, which means that a party might advance its round even before another party has finished with its current round, and even multiple times, as long as its is ensured that no honest party is left too far behind. For this purpose we introduce an even more relaxed version of the (global-setup variant) of the clock from [22] which, intuitively, allows a party to advance to its next round multiple times *before* some honest parties have completed their current round, as long as the relative pace of advancement for any two honest parties stays below a drift parameter Δ_{clock}. We note in passing that a similar guarantee was formulated in the timing model [21]; however, the solution there notified the underlying model of computation which creates complications with the (G)UC composition theorem which would need to be reproved. To avoid such complications, in this work we capture the above relaxed synchrony assumption as a global functionality.[9] and call it $\mathcal{G}^{\Delta_{\text{clock}}}_{\text{ImperfLClock}}$.

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

This local indicator is used similarly to how synchronous protocols would use the perfect indicator in [22]; but we allow the adversary to control when this local indicator is updated under the restrictions that (a) d^{Imp}_{P} cannot advance in the middle of P's round, (b) it cannot fall behind the baseline induced by the indicator d_{P}, and (c) it cannot advance ahead of the baseline by more than Δ_{clock}. This is achieved by the imperfect clock keeping track of the relative difference/distance $\mathtt{drift}_{\mathsf{P}}$ between the number of local advances of each registered P from the baseline updates; this distance is increased whenever d^{Imp}_{P} is reset (by the adversary) to 0 and decreased whenever the baseline indicator $d_{\mathsf{P}} \in \{0,1\}$ is

[9] In [22] a functionality corresponding to the timing-model assumptions [21] was proposed along with a reduction to the (local) clock functionality. However, both the fact that their clock functionality is local and that their reduction uses a complete network of (known) bounded-delay authenticated channels—which we do not assume here—makes that result incompatible with our model and goals.

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

Modeling peer-to-peer communication. We assume a diffusion network, denoted by and we denote it by $\mathcal{F}_{\text{N-MC}}^{\Delta_{\text{net}}}$, in which all messages sent by honest parties are guaranteed to be fetched by protocol participants after a specific delay Δ_{net}. Additionally, the network guarantees that once a message has been fetched by an honest party, this message is fetched by any other honest party within a delay of at most Δ_{net}, even if the sender of the message is corrupted. We note that this network model is not substantially stronger than in previous works [5,3], which use a network functionality providing bounded-delay message delivery. Our model is equivalent via an unconditional reduction: echoing received messages. In practice, this reduction of course needs to be applied prudently to avoid saturating the network. This is exactly done by the relevant networking protocols: e.g. in Bitcoin, when a new block is received its hash is advertised and then propagated and validated by the network as needed. Chronos can use the same mechanism.

Genesis block distribution and weak start agreement. Our model allows parties' local time-stamps to drift apart over the course of an execution; additionally the model makes no assumption that the initialization of the initial stakeholders is completed in the same round, i.e., honest parties might start staking at different rounds of the execution. To this aim, we weaken the functionality $\mathcal{F}_{\mathsf{INIT}}$ adopted by [3] to allow for bounded delays when initial stakeholders receive the genesis blocks. Namely, our $\mathcal{F}_{\mathsf{INIT}}^{\Delta_{\text{net}}}$ functionality merely guarantees genesis block delivery to initial stakeholder not more than Δ_{net} rounds apart from each other; the offsets are under adversarial control.

Further hybrids. The protocol makes use of a VRF (verifiable random function) functionality $\mathcal{F}_{\mathsf{VRF}}$, a KES (key-evolving signature) functionality $\mathcal{F}_{\mathsf{KES}}$, and a (global) random oracle functionality $\mathcal{G}_{\text{RO}}$ (to model ideal hash functions).

3.1 Dynamic (Ad Hoc) Participation

To support a fine-grained dynamic participation model, we follow the approach of [3] and categorize the parties into *party types*. Recall that the dynamic participation model allows to capture the security of the protocol in a realistic fashion, by considering that some parties might be stalling their computation, some might accidentally lose network access and hence disappear unannounced, and others might lose track of the passage of time due to some failure. In our model, we formally let the environment be in charge of connecting and disconnecting to its resources. (This is done by equipping the functionalities, global setups, and

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

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

Derived types:

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

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

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

Fig. 1. Party types.

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.) The various basic and derived types of parties are summarized in Figure 1.

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

Additionally, an honest party is called *synchronized* if it has been continuously connected to all its resources for a sufficiently long interval to make sure that, roughly speaking, (i) it holds a chain that shares a common prefix with other synchronized parties (synchronized state) and (ii) its local time does not differ by much from other synchronized parties (synchronized time). Our protocol's resynchronization procedure JoinProc will guarantee the party that after executing it for the prescribed number of rounds, it will achieve both properties (i) and (ii) above. In addition, such a party will eventually become sign-capable in future rounds (in case the KES is "evolved" too far into the future due to a desynchronized 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 [3]), 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 [3]. 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 now, namely from single points in an execution to the natural numbers, which we refer to as *logical slots* in this context. As we see in Section 4, to each logical slot, a leader election process is associated, which every honest party will run when its local clock `localTime` equals `sl` for the first time. The definition of party types w.r.t. logical slots is 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`.

4 The Blockchain Protocol

At a high level, the protocol we present is a Nakamoto-style proof-of-stake based protocol for the so-called semi-synchronous setting; this is the same model used for standard analyses of Bitcoin. In this model, parties have a somewhat accurate common notion of elapsed time (rather than absolute time information) and the network has an upper bound on the delay which is not known to the parties. At a very high-level the protocol attempts to imitate a process which resembles a situation in which state (including time) is continuously passed on to currently alert stakeholders. The honest majority of active stake assumption that is explicit in [12,3] will then ensure that the adversary cannot destroy this state by using his ability to tune participation.

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

Here is how such an inaugural participant (i.e., a participant who is assigned stake at the outset of the computation by $\mathcal{F}_{\mathsf{INIT}}$) executes the protocol. With access to the provided genesis block, which reveals an initial record $\mathbb{S}_1 = \big((\mathsf{P}_1, v_1^{\mathrm{vrf}}, v_1^{\mathrm{kes}}, s_1), \ldots, (\mathsf{P}_n, v_n^{\mathrm{vrf}}, v_n^{\mathrm{kes}}, s_n)\big)$ that associates each participant

P_i[11] to its chosen public keys used for verification purposes of the staking process and its initial stake s_i, each party begins the so-called first epoch of the staking procedure and sets its local clock `localTime` to the value 1. The party has to execute a certain set of tasks per round. Note that two inaugural parties have only a somewhat accurate notion of elapsed time and receiving the genesis block might be delayed, it might very well be that a party P_1 has executed three rounds, while P_2 has only executed one so far, or has not even received the genesis block. The bounds on the clock drifts and the network delay however ensure that the difference of the number of completed protocol rounds does not drift too far apart.

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

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

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

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

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

chain growth, and chain quality properties for specific parameters, as long as the leader-election per slot is to the advantage of honest protocol participants.[14]

At the epoch boundary to the second epoch, two important things happen. First the stake-distribution and the epoch randomness change: they are derived from specific blocks contained in the guaranteed common prefix established by the first epoch. In particular, we must ensure that at the time the stake distribution is fixed, the epoch randomness cannot be predicted to ensure the freshness of the slot leader election lottery for the second epoch. The second critical update at the epoch boundary is the local time: each party performs a local-clock adjustment, outlined in Section 4.1, which ensures that after the adjustment parties are still close together, where "close" means within $\Delta = \Delta_{\text{net}} + \Delta_{\text{clock}}$ (two sources of bounded variance contribute to this: delay and drift) and that performed shifts of the local clock remain small (which is crucial for security). The desired property follows from the common-prefix guarantee (enabling an agreement on beacons), the honest majority assumption (enabling small clock shifts), and the network properties and clock properties (which ensure correlated arrival times). With some additional considerations detailed in Section 4.1, the protocol proceeds executing the above round tasks for the entire second epoch until the next boundary is met. This iterated process, where one epoch bootstraps the next, is backed by an inductive security argument, following previous works [3,12,23], that shows how the overall security is a consequence of the first epoch achieving the desired blockchain properties to serve as a good basis for the second, etc.

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

The bootstrapping procedure for newcomers is quite involved due to a combination of obstacles: First, the joining party needs to obtain a blockchain that shares some common prefix with the common prefix established by the existing parties. This is achieved by having the joining party listen to the network for some rounds, and picking the "best" chain $\mathcal{C}$ it sees in the following sense: when compared with any other seen valid chain $\mathcal{C}'$, $\mathcal{C}$ contains more blocks in an interval of slots of size s starting from the forking point of $\mathcal{C}$ and $\mathcal{C}'$. We prove that based on the honest-majority assumption, such a densest chain must share a large common prefix with the chains honest parties currently hold. However, $\mathcal{C}$ could still be adversarially crafted and for example be much longer than what honest parties agreed on by extending into the future, hence a reliable ledger state cannot yet be computed. However, it will become possible once the joining party succeeds in bootstrapping also an accurate time-stamp in the Δ interval

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

of honest participants' timestamps, which is the second obstacle to overcome. After the party is guaranteed to be hooked to a large prefix of the honest parties' common-prefix, it begins recording all synchronization beacons it receives on the network for a long enough period of time, a parameter of the system. The length of the waiting time is set in order to ensure that, after the newly joining party started listening to the network, the parties at least once seeded the slot-leadership lottery with a fresh nonce that was unpredictable at the time of joining the system. After an additional waiting time, the agreed-upon set of beacons (with proofs referring to the fresh lottery) will be part of the common prefix and eventually be part of what is known to the joining party. We prove that based on this agreement on beacons found in the blockchain, the clock-adjustments procedure by the current participants in the system can be retraced and will yield a clock adjustment to the newly joining party's local clock that will directly push it into the interval of existing honest participants' local clock. At this point, the party runs the normal chain-selection mechanism, essentially cutting off blocks in its logical future and obtains a reliable ledger state as well.

4.1 The Protocol with Static Ad hoc Participation

Towards a modular description of our protocol, let us first focus on how the protocol would work in the static ad hoc setting, where all parties are alert. In particular, we discuss what such alert parties need to do in order to accommodate synchronzation of joining and rejoining parties. The description of what joining and rejoining parties do—i.e., how they use the help of alert parties to get in-sync—is the included in Section 4.2. Every alert party runs the following round instructions. For the pseudo-code of all involved tasks (and more detailed explanations), we refer to the full version of this work [4].

1. Fetch information from the network over which transactions, beacons, and blocks are sent and further update the current time-stamp and epoch number. A party locally advances its time-stamp whenever it realizes that a new (local) round has started by a call to $\mathcal{G}_{\textsc{ImperfLClock}}$.
2. Record the arrival times of the synchronization beacons produced by all protocol participants. This is discussed in more detail below.
3. Process the received chains: as some chains might have been created by parties whose time-stamps are ahead of local time, the future chains are stored in a specific buffer for later usage (and importantly, not discarded). Among the remaining chains, the protocol will decide whether any chain is more preferable than the local chain using a chain-selection rule inspired by Ouroboros Genesis [3] which we thus refer to as the Genesis rule. An important property of the Genesis rule is that chain selection is secure without requiring a moving checkpoint: roughly speaking, a chain $\mathcal{C}_1$ is preferred over $\mathcal{C}_2$ if they have a large common history, except possibly the last k blocks (where k is some parameter) and $\mathcal{C}_1$ is longer. If however, they fork even before, chain $\mathcal{C}_1$ is preferred if it is block density is higher compared to $\mathcal{C}_2$ in a carefully selected interval of size s slots after the forking point.

4. Run the main staking procedure 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. We provide more details on some of these aspects below.
5. If the end of the round coincides with the end of an epoch, the *synchronization procedure* (denoted $\mathsf{SyncProc}$) is executed.

While the above only gives a broad overview of different tasks per round, we cover some of those in more detail below.

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

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

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

and is parametrized by a quantity $f \in (0,1]$ called the *active slots coefficient* [12].

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

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

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

Note that by (2), a party with relative stake $\alpha \in (0,1]$ becomes a slot leader in a particular slot with probability $\phi_f(\alpha)$, independently of all other parties. We clearly have $\phi_f(1) = f$, hence f is the probability that a hypothetical party controlling all 100% of the stake would be elected leader for a particular slot. Furthermore, the function ϕ has an important property called "independent aggregation" [12]:

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

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 code of the staking procedure is not given here due to space constraints. Briefly, 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 [3], we abstract block formation and transaction validity into predicates $\mathsf{blockify}_{\mathsf{OC}}$ and $\mathsf{ValidTx}_{\mathsf{OC}}$. The function $\mathsf{blockify}_{\mathsf{OC}}$ takes as input a plain sequence of transactions and outputs a block, whereas $\mathsf{ValidTx}_{\mathsf{OC}}$ takes as input a single transaction and the ledger state. A transaction is said to be valid with respect to the ledger state if and only if it fulfills the predicate. The transaction validity predicate $\mathsf{ValidTx}_{\mathsf{OC}}$ induces a natural transaction validity on blockchain-states that we succinctly denote by the predicate $\mathsf{isvalidstate}(\mathtt{st})$ that decides that a state is valid if it can be constructed sequentially by adding one transaction at a time and viewing the already added transactions as part of the state.

Eligibility to emit synchronization beacons. An alert party emits so-called *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 for slot-leadership. To obtain an independent evaluation, we use a new constant called SYNC to obtain domain separation. If the returned value $y \leq T_{\mathsf{P}}^{\mathsf{ep,bc}}$, where in this case we can simply use a linear scaling of the domain, i.e., we define the threshold

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

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

Embedding synchronization beacons in blocks. 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. Validity is decided according to a predicate whose description appears as part of the protocol's code in the full version [4].

Details of the synchronization process. At the end of an epoch, parties run the synchronization procedure based on the beacons recorded in this epoch. The entire synchronization can be logically partitioned into seven logical building blocks. The first five items are definitions and necessary preparatory tasks in order to have the synchronization procedure perform its tasks at the end of an epoch.

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.

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

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

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 following, let $\mathsf{slotnum}(\cdot)$ be the function that returns the first element (the reported slot number) of a beacon.

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 $(n, \mathrm{flag}) \in \mathbb{N} \times \{\mathsf{final}, \mathsf{temp}\}$. 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.

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

5.) *Emitting Beacons and inclusion into the chain:* An alert party sends out a synchronization beacon during a synchronization interval (i.e., if the current local time reports a slot number that falls into a synchronization interval) if and only if the VRF evaluation $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \parallel \mathsf{P}.\mathtt{localTime} \parallel \mathtt{SYNC})$ to $\mathcal{F}_{\mathsf{VRF}}$ returned $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$ with $y < T_{\mathsf{P}}^{\mathsf{ep}}$ where $T_{\mathsf{P}}^{\mathsf{ep,bc}}$ is the beacon threshold in the current epoch as defined in equation 4. An alert slot leader P' on the other hand will include any valid synchronization beacon in its new block as long as $\mathsf{P}'.\mathtt{localTime}$ 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 and was describe in the previous paragraph.

The remaining three steps are implemented as part of the core synchronization procedure $\mathsf{SyncProc}$.

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[15] 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.

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

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

4.2 (Re)Joining Procedures

De-Registration and Re-Joining. If a party is alert, it can lose in several ways its status of being alert. If a party loses access to the random oracle only, then it will still be able to observe the protocol execution and record message arrivals. The main issue is that such a party—when it is fully operational again—will have to retrace what it missed. This is slightly complicated due to the adjustments to the local clock in the course of the execution. However, the party has all reliable information to actually retrace the actions as if it was present as a passive observer all the time. This special procedure SimulateClockAdjustments is given in the full version of this work [4] and it is invoked as part of the main round tasks before performing the actions as an alert party (again).

On the other hand, if any alert party loses access to $\mathcal{G}_{\text{ImperfLClock}}$ or $\mathcal{F}_{\text{N-MC}}$ by the respective de-registration queries, or if it joins anew only late in the execution, then it considers itself as de-synchronized. Parties are aware of their synchronization status, and any party that is de-synchronized will have to run through the main joining procedure that we call JoinProc in order to become alert. Due to lack of space, we cannot provide the code of this procedure and refer to [4]. Below we give an overview of this procedure.

Description of JoinProc. Introducing synchronization slots into the protocol serves the main purpose of enabling a novel joining procedure that newly joining (or resynchronizing) parties can execute to bootstrap an actual reliable time-stamp and ledger state, where a reliable time-stamp is one that lies in the interval of time stamps reported by alert parties. The joining procedure is divided into several phases where the party gathers reliable information, identifies a good synchronization interval and finally applies the shift(s) that will allow it to report a local time-stamp that is sufficiently close to the alert parties in the system. The procedure refers to a couple of parameters. Their concrete values is not necessary to understand its dynamics.

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

Phase B: In the second activation upon a MAINTAIN-LEDGER command, the party will jump to phase B and continue to do so until and including round t_{off}.

During this interval, the party applies the Genesis chain selection rule $\mathsf{maxvalid\text{-}bg}$ 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. As we prove in the security analysis, 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).

5 Security Analysis

We begin by setting down notation and defining the conventions we adopt for measuring stake ratios. The following definition is adapted from [3]; the crucial difference is that it refers to the types of parties with respect to a *logical slot* as defined in Section 3.1.

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: (i.) the* alert stake ratio *be the fraction of stake* $\mathcal{S}^{-}(\mathcal{P}_{alert}[\mathtt{sl}])/\mathcal{S}^{+}(\mathcal{P}_{active}[\mathtt{sl}])$*; and (ii.) the (potentially)* participating stake ratio *be* $\mathcal{S}^{-}(\mathcal{P}_{active}[\mathtt{sl}])$.

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

5.1 Blockchain Security Properties

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

Similarly to [3], we only grant these guarantees to *alert* parties. More importantly for this work, the definitions from [3] need to be adjusted to take into account the fact that the local clocks of the parties are not synchronized. To this end, we choose now to define the properties below with respect to the *logical* timestamps (i.e., slot numbers) contained in blocks, and the local clocks of the parties. Namely, we refer to logical slots below, and a party is considered to *be on the onset* of slot $\mathtt{sl}$ (or *enter* slot $\mathtt{sl}$) if her local clock just switched to $\mathtt{sl}$.

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

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

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

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

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

While these definitions based on the logical time allow us to talk about the logical structure of the forks created by the parties and reuse parts of the technical machinery given in [23,12,3] to analyze it, providing only guarantees based on the logical time would be unsatisfactory, as the parties running Chronos desire persistence and liveness with respect to a more "real-time" notion (that we define in a moment). We will address this translation from logical-time to real-time guarantees later in Section 5.2.

For many of the security arguments it will be convenient to define a notion of *nominal time*; even though inaccessible to alert parties, we will use it in the proofs to express time-relevant properties of an execution.

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

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

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

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

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

Note that in the static-registration setting (where parties do not join or leave), all honest parties are synchronized (and hence are considered for $\mathsf{Skew}_{\Delta}[\mathtt{sl}]$).

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

5.2 Proving the Blockchain Properties

We phrase here the asymptotic version of our main result, its concrete-security variant is proven in the full version [4].

Theorem 2. *Consider an execution of the protocol* $\mathsf{Chronos}$ *in the dynamic-availability setting and let* κ *denote a security parameter. Let* f *be the active-slot coefficient and* R *the epoch length, let* Δ *be the upper bound on the sum of the maximum network delay and maximum local clock drifts, and let* $\widetilde{\Delta} \triangleq 2\Delta$*. Let* $\alpha, \beta \in [0, 1]$ *denote a lower bound on the alert and participating stake ratios throughout the whole execution, respectively. Assume that for some* $\epsilon \in (0, 1)$ *we have*

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

and that the $\mathsf{maxvalid\text{-}bg}$ *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, all blockchain properties $\mathsf{CP}(k'), \mathsf{CG}(\tau, s')$*,* $\mathsf{CQ}(\mu, k')$*,* $\exists\mathsf{CQ}(s')$ *(for concrete coefficients* τ *and* μ *defined in the proof) hold except with negligible probability in* κ *whenever* s' *and* k' *as well as the chain-selection parameters* k *and* s *of* $\mathsf{maxvalid\text{-}bg}$ *are functions in* $\omega(\log \kappa)$*.*

Note that the bound on s implies that the epoch length R has a lower bound in $\omega(\log \kappa)$ in such an asymptotic treatment.

Outline of the proof. We only give a brief overview of the proof and refer to the full version of this work [4]. To handle the proof complexity, the proof is divided into a sequence of logical steps:

1. A proof that the blockchain properties CP, CG, CQ, and $\exists\mathsf{CQ}$ hold in a static registration setting (where parties do not join or leave) and for a single epoch. In view of an inductive proof, this serves as the security base case.
2. Once we can rely on the blockchain properties, we can as a second step analyze the synchronization procedure and prove that no matter what the adversary does, the parties will always stay close together when transitioning from one epoch, say i where the security properties hold, to the next and that the clock-adjustments are very small. Two properties are important:

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

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

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

3. By an inductive argument, if we start with a bounded-skew initial epoch (which is guaranteed by the weak start agreement), the above two steps allow us to conclude the security of the (multi-epoch) blockchain protocol, but without parties joining.
4. A party joining the network acts like an observer of the network (i.e., it does not interfere with the protocol) and becomes synchronized after extracting enough information from the network, at which point it can start to be an active protocol participant. This step of the security proof can hence be conducted based on the previous analysis. Our analysis shows two properties of the joining process of $\mathsf{P}_{\mathsf{join}}$ that hold as long as the established properties CP, CG, $\exists\mathsf{CQ}$ remain satisfied throughout the joining process:
   (a) After Phase B, $\mathsf{P}_{\mathsf{join}}$ will be holding a chain $\mathcal{C}_{\mathsf{join}}$ that satisfies $\mathcal{C}_{\mathsf{alert}}^{\lceil k} \preceq \mathcal{C}_{\mathsf{join}}$ with respect to any $\mathcal{C}_{\mathsf{alert}}$ held by an alert party at least Δ time steps ago.
   (b) In Phase D, $\mathsf{P}_{\mathsf{join}}$ correctly identifies an epoch i^* for which it has collected all the beacons that alert parties had used in their execution of $\mathsf{SyncProc}$ after epoch i^*, and based on these beacons mimics the synchronization procedure so that starting with epoch i^*+1, $\mathsf{P}_{\mathsf{join}}$ does not violate Skew_{Δ} as it becomes alert.
5. At this point, we are ready to derive the CP, CG,CQ, and $\exists\mathsf{CQ}$ guarantees for the entire protocol in a fully dynamic world, where parties join any time, might be temporarily stalled, and disappear unannounced. This can be argued based on a case distinction on different party types (cf. Section 3.1) and quantify their impact on the security guarantees established above. This concludes the proof.

From Logical-Time to Real-Time Guarantees for Chain Growth. Recall that eventually, we are interested in a ledger that provides consistency and liveness and they typically follow black-box from the blockchain properties above. However, since in our protocol, parties emulate a global time themselves, we must make related logical time advancement with the nominal time, which is especially important for liveness. Since parties adjust their timestamps at the boundary of every epoch, an external observer that takes nominal time as the baseline, would conclude that parties are slightly off. To quantify the general relationship, we introduce a concrete discount factor τ_{TG}. We state the informal lemma here, which is proven with a concrete expression for τ_{TG} in the full version [4].

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

It is important to point out that the τ_{TG} is close to 1 for typical parameter choices and that the lower bound on δt does depend on Δ and not on the security parameter. We are ready to state chain-growth with respect to nominal time. Again, the formal statement with concrete bounds is given in the full version [4].

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

6 The Synchronizer

We now explore the properties of the time-stamps that are recorded by our blockchain protocol and how to export a clock based on them. Recall that in the view of each party P, blocks feature *extended* local timestamps $\mathtt{time}_{\mathsf{P}}$, equal to the pair $\mathtt{time}_{\mathsf{P}} = (e, t)$, where t is the time-value, and e is the number of non-monotone adjustments to t, i.e., the number of epoch switches that P has observed (and hence the synchronization procedure was executed). The following lemma (proven in the full version [4]) captures the properties of these timestamps.

Lemma 2 (Quality of Exported Time-Stamps). *Consider an execution of the full protocol* Chronos *in the dynamic-availability setting, let* P *be a party and let the sequence* $(e_1, t_1), \ldots, (e_n, t_n)$ *denote the updates that* P *makes to its exported time-stamp between two arbitrary instances in the execution where in between* P *is synchronized throughout. Then the timestamps satisfy the following properties:*

1. *No reported time stamp* t_i *is further than* 2Δ *slots apart from any other alert party's time value and no other alert party reports an e-value that differs by more than* 1*. If another alert party reports the same e-value, then the exported times are at most* Δ *apart.*
2. *Any subsequence of the same epoch* $(e, t_j), \ldots, (e, t_k)$*,* $k > j$ *has monotone increasing time-stamps with increments of* 1 *happening whenever* P *locally completes a round in the execution.*
3. *The only non-monotone behavior of the exported time can occur at most once per epoch, namely at the epoch boundary* $(e, t) \to (e+1, t')$ *with* $t \mod R = 0$*, and it holds* $t' \leq t + \Delta$ *and* $t' \geq t - 2\Delta$*.*

Having established this final piece, we can couple it with the statements above, notably with Theorem 2—which guarantees that we have achieved a blockchain protocol in the dynamic availability setting with all required properties—which overall assures that we have a protocol that outputs reliable, accurate two-dimensional time-stamps in the dynamic availability setting: any party and any observer is able to compute a reliable time-stamp, no matter when he or she joined or started observing the system.

Proof of Theorem 1[The synchronizer]. Theorem 1 follows as a simple corollary of the above. In fact, we just need to map the above 2D time-stamps to the natural numbers: an alert party, obtaining sequences of (2-dimensional) time-stamps from the underlying protocol over the course of an execution, say $E = (e_1, t_1), \ldots, (e_n, t_n)$, simply maps this to an integer by the map $\tau_i \leftarrow \max_{j \in [i]}\{t_j\}$. This integer time-stamp satisfies the abstract properties 1. to 4. demanded by Theorem 1. Clearly, the outputs are natural numbers, then by property 1. of Lemma 2 we obtain the bound between time-stamps of $2\varDelta$, and by combining properties 2. and 3. of Lemma 2, the third and fourth properties of Theorem 1 follow, i.e., the final sequence of time values are non-decreasing and guaranteed to increase after a constant number of local rounds have elapsed (since the underlying 2D timestamps never roll back more than $2\varDelta$ in the second coordinate).

In the full version of this paper [4], we additionally give a UC proof of the protocol that follows in a straightforward way from the above properties. The protocol UC-realizes a functionality that combines a ledger with a clock. We analyze in a modular way further settings, including optimistic network models with known expectation and variance of delay to show that it is possible to approximate real-time progression extremely accurately.

References

1. Marcin Andrychowicz and Stefan Dziembowski. PoW-based distributed cryptography with no trusted setup. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 379–399. Springer, Heidelberg, August 2015.
2. Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Optimal clock synchronization under different delay assumptions (preliminary version). In Jim Anderson and Sam Toueg, editors, *12th ACM PODC*, pages 109–120. ACM, August 1993.
3. 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.
4. Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros chronos: Permissionless clock synchronization via proof-of-stake. Cryptology ePrint Archive, Report 2019/838, 2019. `https://eprint.iacr.org/2019/838`.
5. Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.

6. Amos Beimel, Yuval Ishai, and Eyal Kushilevitz. Ad hoc PSM protocols: Secure computation without coordination. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 580–608. Springer, Heidelberg, April / May 2017.
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. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
9. 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.
10. Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.
11. Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *LNCS*, pages 23–41. Springer, 2019.
12. Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.
13. 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.
14. 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.
15. 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.
16. Uriel Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation (extended abstract). In *26th ACM STOC*, pages 554–563. ACM Press, May 1994.
17. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.
18. 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`.
19. 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.
20. Yuval Ishai and Eyal Kushilevitz. Private simultaneous messages protocols with applications. In *ISTCS 1997*, pages 174–184. IEEE Computer Society, 1997.
21. Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. Concurrent composition of secure protocols in the timing model. *Journal of Cryptology*, 20(4):431–492, October 2007.

22. 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.
23. 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.
24. 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.
25. 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.
26. Aanchal Malhotra, Matthew Van Gundy, Mayank Varia, Haydn Kennedy, Jonathan Gardner, and Sharon Goldberg. The security of ntp's datagram protocol. In Aggelos Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 405–423. Springer, 2017.
27. David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition.* CRC Press, 2010.
28. 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.
29. 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.
30. Rafael Pass and Elaine Shi. Rethinking large-scale consensus. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 115–129. IEEE Computer Society, 2017.
31. Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 380–409. Springer, Heidelberg, December 2017.
32. 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.
33. 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.