# Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability

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

February 22, 2019

**Abstract.** Proof-of-stake-based (in short, PoS-based) blockchains aim to overcome scalability, efficiency, and composability limitations of the proof-of-work paradigm, which underlies the security of several mainstream cryptocurrencies including Bitcoin.

Our work puts forth the first (global universally) composable (GUC) treatment of PoS-based blockchains in a setting that captures—for the first time in GUC—arbitrary numbers of parties that may not be fully operational, e.g., due to network problems, reboots, or updates of their OS that affect all or just some of their local resources including their network interface and clock. This setting, which we refer to as *dynamic availability*, naturally captures decentralized environments within which real-world deployed blockchain protocols are assumed to operate.

We observe that *none* of the existing PoS-based blockchain protocols can realize the ledger functionality under dynamic availability in the same way that Bitcoin does (using only the information available in the genesis block). To address this we propose a new PoS-based protocol, "Ouroboros Genesis", that adapts one of the latest cryptographically-secure PoS-based blockchain protocols with a novel *chain selection rule*. The rule enables new or offline parties to safely (re-)join and bootstrap their blockchain only from a trusted copy of the genesis block without any additional advice—such as checkpoints—or assumptions regarding past availability. We say that such a blockchain protocol can "bootstrap from genesis."

We prove the GUC security of Ouroboros Genesis against a fully adaptive adversary controlling less than half of the total stake. Our model allows adversarial scheduling of messages in a network with delays and captures the dynamic availability of participants in the worst case. Importantly, our protocol is effectively independent of both the maximum network delay and the minimum level of availability—both of which are run-time parameters. Proving the security of our construction against an adaptive adversary requires a novel martingale technique that may be of independent interest in the analysis of blockchain protocols.

## 1 Introduction

The primary real-world use of blockchains, thus far, has been to offer a platform for decentralized cryptocurrencies with various capabilities [26, 5]. A unique feature of blockchain protocols (in contrast to standard multiparty computation) from which the setting draws much of its appeal is the fact that the parties that run the protocol may engage only in passing with the protocol and need not identify themselves to other protocol participants. In fact, the Bitcoin blockchain protocol remains robust in the presence of a Byzantine adversary even if parties arbitrarily desynchronize, join at any moment of the execution, or go offline for arbitrary periods of time (a set of execution features that we will refer to as *dynamic availability*), as long as a majority of hashing power is always following the protocol.

Motivated by this novel setting, several applications have recently emerged that use blockchains (or the cryptocurrencies that build on top of them) as enablers for other cryptographic protocols. For example, a

⋆ University of Edinburgh and IOHK, `christian.badertscher@ed.ac.uk`. Work done while at ETH Zurich.

⋆⋆ IOHK, `peter.gazi@iohk.io`.

⋆ ⋆ ⋆ University of Edinburgh and IOHK. `akiayias@inf.ed.ac.uk`. Research partly supported by EU Project No. 780477, PRIVILEDGE.

† University of Connecticut and IOHK. `acr@cse.uconn.edu`. This material is based upon work supported by the National Science Foundation under Grant No. 1717432.

‡ University of Edinburgh and IOHK. `vassilis.zikas@ed.ac.uk`.

number of recent work [2, 4, 24, 23, 1] describe how blockchain-based cryptocurrencies can be used to obtain a natural notion of fairness in multi-party computation against dishonest majorities; or to allow parties to play games of chance—e.g., card games like poker, lottery-based games, etc.—without the need of a trusted third party; or how to use blockchains as bulletin boards in electronic voting. Such developments—in conjunction with the direct applicability to cryptocurrencies—have led to a pressing need for a general, formal security analysis of the functionality that blockchain protocols provide.

Recently, Badertscher et al. [3] put forth the first composable analysis of Bitcoin, by proving that it implements, in a universally composable (UC) manner, an immutable transaction ledger. This improved on previous works [17, 27] that provided a game-based security analyses and rigorously described an ideal ledger that provides an answer to the question: What is the goal that Bitcoin aims to achieve? The advantage of such a UC treatment of blockchains is that it allows for a modular design and security analysis of the above cryptographic applications of blockchains.

Notwithstanding, the wide adoption of Bitcoin has revealed some serious efficiency and (in)composability issues. The efficiency issues stem from the fact that it relies on *proof-of-work* (in short, *PoW*), a cryptographic puzzle-solving procedure with increasing difficulty as more parties join the system.[1] Composability issues are due to the fact that the puzzle-solving procedure can, in principle, be useful also for other protocols—independent from the Bitcoin mining process.[2] This means that one cannot exclude the possibility that an adversarial miner, participating in such an independent protocol $\pi$ and in Bitcoin in parallel, can potentially double the value of his effort, by using the same hash query both for $\pi$ and for Bitcoin.

The demand for blockchain solutions that do not suffer from the above issues gave rise to an exciting recent line of work that propose to use alternative resources to achieve consensus and maintain a robust ledger. The most popular such resource is *stake* in the system. Informally, instead of requiring a party to invest computing power in order to be allowed to extend the blockchain, parties are given the chance to do so according to their *stake* in the system, e.g., the number of coins they own. This paradigm, often referred to as *proof-of-stake* (in short *PoS*), has yielded a number of proposals for PoS-based blockchains.[3]

Several of these PoS-based proposals originated from the cryptographic community, e.g., Algorand [19], Snow White [13], and Ouroboros/Ouroboros Praos [22, 14]. As such they are accompanied by a formal security proof that they achieve a well defined set of desirable properties. Alas, these protocols severely restrict the dynamic availability of participants: Snow White [13] and Ouroboros/Ouroboros Praos [22, 14] require an honest blockchain to be delivered as trusted "advice" to any joining party—a form of "checkpointing", while Algorand [19] requires the explicit knowledge of a good estimate of the number of offline parties. Furthermore, all these works focus on a property-based specification of the provided security guarantees, i.e., they prove that they achieve a desirable set of properties. Such property-based definitions are known not to ensure, in general, the composability of the proposed schemes [8, 10, 20].

This leaves the following questions open:

– *Can PoS offer the same level of dynamic availability guarantees as PoW? In particular, joining without checkpointing advice or the knowledge of other parties' online/offline status?*
– *What is the ideal functionality implemented by PoS-based blockchains? How does it compare to the one implemented by PoW?*
– *Does PoS suffer from the same incomposability issues as PoW?*

Our work addresses all the above questions. We put forth the first UC treatment of PoS-based blockchains. Our model captures for the first time dynamic availability and provides a fine-grained classification of failures that determine all different settings that an honest protocol participant may find itself in during the protocol

[1] In Feb. 2019 each single Bitcoin block requires more than $2^{60}$ operations to be performed, cf. `https://en.bitcoin.it/wiki/Difficulty`.

[2] The concept of merged mining is an illustration of this fact from a positive angle; cf. `https://en.bitcoin.it/wiki/Merged_mining_specification`.

[3] In fact, as a response to the criticism about the bottlenecks of PoW, the second most adopted decentralized blockchain, Ethereum [5], has announced a plan to gradually transition from a PoW-based to a PoS-based protocol.

execution. Given that none of the existing PoS protocols provide such strong guarantees, we describe and analyze a new protocol based on Ouroboros Praos [14]. The major structural change towards our new protocol, which we call Ouroboros Genesis, is a novel chain selection rule that enables joining parties to "bootstrap" only from a trusted copy of the genesis block. We prove that the protocol UC-securely implements the natural ledger functionality proposed in [3]—the very same functionality shown to be possessed by Bitcoin. We prove security in the setting of dynamic availability under the assumption of standard cryptographic primitives, an initialization functionality that is akin to public-key registration and a global random oracle which is a natural abstraction of deterministic hash functions. Our contributions are discussed in more detail below.

**Our Contributions.** Our work provides a Universally Composable (UC) [8] treatment of proof-of-stake-based blockchains. To obtain a tight abstraction of the real-world setting and stronger composability guarantees, our treatment is in the UC model with *global setups*, a.k.a. GUC [9]; note that all our statements trivially apply also to the standard UC model by considering global setups as ideal UC functionalities.[4]

DYNAMIC AVAILABILITY. Our first contribution is capturing in an accurate manner the guarantees that any such protocol can give to freshly joining parties and/or parties with temporary connectivity/availability issues, a setting that we call dynamic availability. More concretely, our model distinguishes several classes of honest parties, based on:

- whether the party has access to all three resources it needs (i.e., is registered with the respective functionalities): the random oracle, the clock, and the network (see Fig. 1 for the detailed enumeration of these classes);
- whether the party has been able to follow the protocol in such a state for sufficiently long to be guaranteed to hold a synchronized view of the ledger (we call such parties *synchronized*). More specifically, we will describe a parameter called `Delay` (a function of the network delay $\Delta$) that determines the time sufficient for a party with access to all resources to become fully synchronized with the state of the protocol. We stress that `Delay` is a parameter that is unknown to the protocol participants, thus it is impossible for a party to determine whether it is already synchronized.

This detailed classification allows us to precisely capture the following derived party classes that will be essential for describing the security guarantees provided by our protocol:

- *Alert parties* are parties that have access to all the required resources, and are also synchronized. These parties enjoy full security guarantees and we will require a lower bound on their number (see below) to ensure security.
- *Potentially active parties* (or *active* for short) are all parties that, broadly speaking, might potentially act in the current time slot of the protocol execution. This includes honest parties that have access to all the required resources as well as adversarial parties. Note that honest parties who lose access to clock are also potentially active, as they are not able to reliably evolve their private state, and hence it cannot be excluded that if they later get corrupted, they might act retroactively.
- *Inactive Parties* are all other parties, such as honest parties that cannot access some of the necessary resources to engage with the protocol, e.g., their network connection. Per protocol rules, such parties abstain from active participation until that access is regained.

Our objective is to realize the ledger functionality given the following two conditions:

(A) *The ratio $\alpha$ of the number of alert over the active parties is above* $1/2$*; the difference is by a constant that is sufficiently large to absorb the partial synchrony delay parameter $\Delta$.*
In particular (and similar to the Bitcoin blockchain, see [17, 27]) the protocol will use a parameter $f$ and will permit a meaningful security guarantee provided that a suitable relation between $f$ and $\Delta$ is satisfied. For a fixed choice of $f$, the larger the delay $\Delta$ is, the larger the difference between $1/2$ and the alert over active ratio should be. Note that this condition is simply a translation of the standard honest-majority assumption into our dynamic-availability setting.

[4] Informally, the main difference between ideal functionalities and global setups is that the former are bound to a calling protocol and only expose their functionality to this protocol, whereas the latter can be accessed by any protocol.

(B) *The ratio $\beta$ of the number of active over all parties is bounded from* 0 *by some arbitrary constant that is unknown to the protocol participants.*
This necessary assumption provides an upper bound on the probability that *no* party will be able to act in the protocol at a given time (if it fails the protocol may halt).

It is instructive to consider what role various party classes play in the above assumptions. In particular, let us stress that the stake of inactive parties, such as those that have lost (or given up) their network connection or their access to the random oracle, does not count as adversarial; in other words, they do not affect the ratio considered in Assumption (A) above.

The above guarantees (and the corresponding assumptions) are arguably natural. However, none of the existing PoS protocols provides them without additional assumptions and/or restrictions to the adversary's capabilities. Concretely, existing solutions and proposals [22, 13, 14, 6] either forfeit dynamic availability and assume honest parties are regularly online or rely on an assumption that joining (or resuming) parties are implicitly given access to a *checkpointing* functionality, which serves them a trusted recent honest chain and is supposed to be implemented either "for-free" by the environment or by some fortuitous network connection to existing honest parties. This solves the problem of joining parties getting up to speed with the correct chain—which is the main challenge here–but is arguably a strong assumption. To see this, note that given such a functionality parties only need to deregister and register in order to obtain eventual consensus, which completely trivializes the main goal of a blockchain protocol. One can attempt to avoid such trivialization by restricting the interval between deregistration and reregistration, or even forbidding it, but this makes the assumption somewhat artificial and excludes natural scenarios from the analysis, such as short-term unavailability (e.g. due to a system crash, network outage, maintenance, or update restart). We also note that even with the assumption of such an additional checkpointing functionality, there is no existing PoS solution which can tolerate both the optimal threshold of adversarial stake ratio approaching 1/2, and full adaptivity in corruptions and in the (re-)joining schedule, i.e., (re-)registration/deregistration.

Note that the PoS protocol Algorand [19] does not require such checkpointing or parties being regularly online, however it requires a good estimate on participation to be fixed in the protocol, thus forfeiting dynamic availability as well. This requirement stems from the fact that the core of the protocol runs a Byzantine agreement sub-protocol that requires to be able to know (bounds on) the level of expected participation and hence estimate in advance the number of messages that are required to proceed with key protocol decisions.

A PoS Blockchain with Bootstrapping from Genesis. Given the deficiencies of existing protocols to handle dynamic availability we present a new protocol, Ouroboros Genesis, that is based on a recent PoS protocol, Ouroboros Praos [14]. The novelty of our protocol lies in its chain selection rule that instantiates the so-called maxvalid procedure in [17, 22, 3, 14] in a way that allows the parties to identify a chain whose prefix has been part of the prefix of a recent honest chain, *using only knowledge of the genesis block.* For this reason we refer to this process as *bootstrapping from genesis.*

Concretely, we prove that Ouroboros Genesis (G)UC-securely realizes the ledger functionality in the dynamic-availability setting, as long as the two essential conditions (A) and (B) introduced above are satisfied. Towards this goal, we develop a new technique which non-trivially extends the martingale argument from [29] so that we can use it to analyze an adaptive adversary in the presence of a worst-case (adversarial) joining-schedule. This technique is of independent interest, as it might prove useful for analyzing other PoS-based blockchains. Overall our security proof maintains the same cryptographic assumptions as [14].

Global UC formalisation. As a technical contribution, along the way we also provide a full specification of the real-world resources needed for PoS as ideal functionalities and global setups in GUC. Concretely, following the paradigm of [3], we capture protocols in the (semi-) synchronous model as (G)UC protocols with access to a global clock functionality, and to a network with eventual (bounded) delivery. A delicate deviation from [3], which also formally demonstrates the stronger composability guarantees that PoS offers, is with respect to how we abstract the calls to the hash function. Concretely, we assume the protocol participants have access to a *global* random oracle setup (in short, GRO). This captures the abstraction of hash functions as *publicly available* random functions. This should be contrasted to their abstraction as a UC functionality proposed in [3], which is less composable. Intuitively, a (deterministic) hash function can be queried by any party, whereas a UC random-oracle functionality is available only to its calling protocol $\pi$; this implicitly restricts access to this functionality (and therefore the hash function it is supposed to abstract) on the specific protocol $\pi$.[5]

In fact, a closer consideration of the idiosyncrasies of PoWs reveals that abstracting hash-queries as calls to a GRO is not an option for PoW-based blockchains. This is true because of two issues: (1) at an intuitive level this would imply that the environment (i.e., other protocols) could make queries to the GRO and then share them with the adversary, which, as discussed above, gives "free" out-of-band computing resources to the adversary; (2) at the more technical level, the non-programmability of the GRO allows the environment to check that the simulator creates blocks that indeed carry sufficient work; but since the simulator needs to also simulate the hash queries of honest parties, this would only be feasible if he had a much larger query-budget than the adversary, which is not possible as the GRO needs to behave identically in the real and ideal world. We note in passing that in [11] a version of the GRO was proposed that reduces the power of the environment to check on the simulator; this GRO—which is arguably not the most realistic abstraction of hash functions—would still not work for PoWs because of the first issue. Demonstrating that PoS-based schemes can be proved in a model where hash functions are abstracted as GROs (which is not the case for the PoW setting) sheds light on the comparison between PoW and PoS.

Our results and analysis thus categorically address the three questions posed above: Ouroboros Genesis can realise the same ledger functionality as Bitcoin, $\mathcal{G}_{\text{LEDGER}}$ [3], in a setting with dynamic availability using standard cryptographic assumptions. In particular, the realization proof is carried out in a model where hash-functions are abstracted as global random oracles which shows that the consensus step does not suffer from the composability issues of PoW protocols. We note that providing explicit realizations of all the hybrid functionalities in the GRO model is beyond the scope of the current paper. Nevertheless, we observe that PoS protocols can effectively drop-in replace PoW based ones with the only added requirement being the initialization functionality that should provide a key registration service as opposed to merely a common random string.

**Related Work.** A number of recent works have studied—in a rigorous cryptographic manner—the security of existing and newly proposed blockchain protocols both PoW-based, e.g, [17, 27, 3] and PoS-based, e.g., [22, 14, 28, 13, 19]. In the PoW-based setting, [3] describes and proves the composable security guarantees of the most representative protocol, namely Bitcoin; furthermore, the security proof tolerates an adaptive adversary and achieves optimal resilience—the adversary can control any percentage less than 50% of the network's total computing power. In contrast, in the PoS-based setting, no simulation-based (UC) proof existed, and different proposed schemes tolerate different types of adversaries in terms of adaptivity. For example, Ouroboros [22] achieves only "semi-adaptive" security (corruptions with delay), whereas among the adaptively secure ones, Algorand [19] needs less than 1/3 of the stake of the system to be held by malicious parties, whereas Show White [13] and Ouroboros Praos [14] achieve the optimal 1/2 bound, at the cost of needing a checkpointing functionality to accommodate joining parties.

The idea of parties that are muted for some time but do receive their messages was first proposed in [28] where those parties were referred to as *sleepers*. Our modeling of such parties differs from (and in fact strictly generalizes) that of [28] in various ways: first, instead of describing them by means of whether they are paused or not, we characterize them by means of the availability of their resources, making clear how those parties enter this state. Furthermore, our notion is only affecting the PoS session that is being executed and thus, in our composable setting, such parties are not restricted as to how they should behave within other protocols that they concurrently participate in. To emphasize this distinction and the fact that they may be continuing to operate in other protocol sessions we use the term "stalled" for these parties (cf. Fig. 1). In addition to the modeling distinctions, our model allows us to obtain more general statements regarding the adaptivity of the adversary. Concretely, we can tolerate fully adaptive adversaries and worst-case registration/deregistration scheduling. In contrast, [13] tolerates semi-adaptive adversaries, whose corruption only takes effect after a certain number of rounds. Interestingly, there is no need for distinguishing a class of parties called deep-

[5] For readers familiar with the programmability issues of random oracles, e.g. [16, 11], the UC-functionality abstraction corresponds to a programmable RO that is only accessible by the protocol, whereas the random-oracle global setup is both non-programmable and publicly available.

sleepers in [13] (i.e., those that are in sleepy mode for a prolonged time) that required checkpointing via a safe initialisation string and thus the related impossibility result in [13] can be circumvented. Taking advantage our bootstrapping from genesis chain selection rule, all parties that are stalling, even for prolonged periods of time, can safely resynchronize without the assistance of a trusted initialisation exactly as in the case of PoW-based protocols.

**Outline of the remainder of the paper.** In Section 2 we provide a formal description of our model of computation, including our real and ideal world functionalities and setups. In Section 3 we describe Ouroboros Genesis as a (G)UC protocol called Ouroboros-Genesis. The security analysis of the protocol, i.e., the proof that it UC-securely realizes the ledger functionality is given in Section 4. The proof starts by considering the interaction of the old chain selection procedure from [14] (called maxvalid-mc here, the protocol using it is dubbed Ouroboros-Praos) with honest parties assumed to be connected to the network and synchronized (Section 4.2), gradually incorporates the new maxvalid-bg procedure which allows the protocol to bootstrap from the genesis block (Section 4.3), along with proofs that this procedure is sufficient to provide all the guarantees offered to newly joining and temporarily offline parties (Section 4.4). Finally, there results are transformed into the full UC statement in Section 4.5.

## 2 The Model

This section includes the main components of the computation model including the real and ideal functionalities used in this work. We assume the reader has some familiarity with the universal composition (UC) framework [8]. In addition to the new functionalities, we make use of the number of already existing functionalities from the literature. For completeness we nonetheless include these functionalities in Section A of the supplementary material.

UC defines security via the simulation paradigm: the protocol execution in the real world is compared to an ideal execution, where the parties have access to an ideal functionality $\mathcal{F}$ which abstracts the goals of the protocol. In the ideal world honest parties act as simply relayers between their environment $\mathcal{Z}$ and the functionality $\mathcal{F}$ (i.e., they run the so called *dummy* protocol [8]). Informally, security requires that the attack of any adversary against the (real-world) protocol can be simulated in the ideal world. More concretely, for any real-world adversary $\mathcal{A}$ there should exists an ideal-world simulator $\mathcal{S}$ that corrupts the same parties as $\mathcal{A}$ and makes the ideal-world execution indistinguishable from the real-world in the eyes of any environment $\mathcal{Z}$.

Importantly, the (real-world) protocol might be given access to some functionalities (often called *hybrids*), which capture the resources that the parties have available, e.g., their communication network. In standard UC, these resources appear only in the real-world—in fact they are formally treated as part of the protocol—whereas GUC [9] allows such resources to be preserved in the ideal world and as such be accessible directly by the environment (instead of their interface being filtered by the protocol.) To avoid confusion with standard UC functionalities, the GUC resources of the above type as often referred to as (global) *setups*. They capture, among others, settings where different protocols might share a common state, and allow to address deniability issues that the original UC framework has [9]. Furthermore, the fact that they do not disappear in the ideal world makes global setups more suitable for capturing universally accessible resources such as deterministic hash functions as discussed in the introduction.

In the following, we describe the real-world resources that are needed in Ouroboros Genesis, along with the ideal world functionality that the protocol implements. Before doing so, we discuss some common conventions that we will use in the descriptions.

**Dynamically available party sets.** A significant extension in the model of computation in our work, is the high granularity in the treatment of the protocol participant's availability. Concretely, already in [3] all functionalities, protocols, and global setups have a dynamic party set. I.e., they all include special instructions allowing parties to register, deregister, and allowing the adversary to learn the current set of registered parties. Additionally, global setups allow any other setup (or functionality) to register and deregister with them, and they also allow other setups to learn their set of registered parties.[6] These registration commands, as outlined

[6] The latter is done by use of a technical modeling trick from [12] (cf. Section A.1.)

in Section A.1 will be part of the code of *all* (hybrid and ideal) functionalities and setups considered in this work. For simplicity, we will not write them explicitly in the pseudo-code of the functionalities.

Having such a flexible and dynamic registration/deregistration schedule, requires special care in the blockchain setting. E.g., in [3] it is observed that parties that have very recently joined the Bitcoin network cannot receive all guarantees of honest parties. Intuitively, the reason is that, due to network delays, these parties, called *desynchronized*, might be temporarily tricked into working on a fake (adversarial) chain. In this work we go one step further towards capturing all availability scenarios, and the corresponding guarantees that can be offered to parties with different availability patterns. We refer to Section 2.2 for more details.

**The adversary.** We assume a central adversary $\mathcal{A}$ who corrupts miners and uses them to attack the protocol. The adversary is *adaptive* meaning that he can add miners to his corrupted set at any point in the protocol execution and can do so depending on his current view of it.

**Assumptions on the environment/adversary as setup-functionality wrappers.** In order to prove statements about cryptographic protocols, one often makes assumptions about what the environment (or the adversary) can or cannot do. For example, to prove resistance against sleepy parties [28], one needs to assume that awake (non-sleepy) honest parties are always in the majority. Such assumptions can be captured by a restricted environment and/or adversary. However, this is against the spirit of a general composition theorem and technically prevents us from applying it in a further construction step (where for example the ledger is used as a hybrid). To circumvent this undesirable property, we follow the paradigm of [3] to capture such assumption by means of a functionality wrapper that wraps the (local setup) functionalities that the protocol accesses and forces the required assumptions on the adversary/environment. In some sense, we shift such assumption or restrictions from the environment into the setup resources. We refer to [3] for a more detailed discussion. Looking ahead, the wrapper used in our security statements is sketched in Section D (This wrapper will only become relevant to interpret Theorem 3 without the need of a restricted environment or adversary).

### 2.1 The Real World Execution

Protocol participants are represented as parties—formally Interactive Turing Machine instances (ITIs)—in a multi-party computation. The main aspects of this computation are as follows:

**Communication.** The parties interact which each other by means of a network of eventual delivery unicast channels [3]—informally, every party $U_p$ has an open incoming-connections interface where he might receive messages on from arbitrary other parties. This captures the natural joining procedure of real-worlds blockchains where new parties find a point of contact and use it to communicate with other parties by means of a gossiping (flooding) protocol. As argued in [3] assuming the honest parties are strongly connected, this network can be used to build the (UC version of the) standard multicast network with eventual delivery assumed in [17, 27, 22]. The abstraction of this network as a (local)[7] UC functionality and its implementation from unicast channels was described in [3]. For completeness, we include this functionality in Section A.2.

For the remainder of this work we will assume parties have access to such a multicast network. This network, denoted as $\mathcal{F}_{\text{N-MC}}^{\Delta}$, has an upper bound $\Delta$ in the delay that the adversary can incur on the delivery of any message; we stress, however, that the protocol is oblivious of $\Delta$ and this bound in only used in the security statement. Hence from the protocol's point of view the network is no better that an eventual delivery network (without a concrete bound).

**Synchrony.** All known PoS-based blockchains, including Ouroboros Genesis, are (partially) synchronous, i.e., they proceed in synchronized rounds with either a known (or an unknown, in the case of partial synchrony) message delay. We model synchronous computation using the synchronous-UC paradigm introduced in [21] and adapted to GUC in [3]. Concretely, the parties are assumed access to a global clock setup, denoted

[7] It is natural to capture network functionalities as local UC functionalities, since networks are often ad-hoc tailored to a specific task.

as $\mathcal{G}_{\text{CLOCK}}$ (see Section A.3.) Each registered party can signal the clock that it is done with the current round, and once all honest registered parties (and functionalities) have done so, the clock advances by one tick. In addition, every party can query the clock to read the (logical) time.

As observed in [3], to obtain UC realization in such a globally synchronized setting, the target ideal functionality needs to keep track of the number of activations that an honest party gets—so that it can enforce in the ideal world the same pace of the clock as in the real world. This can be achieved by describing the protocol so that it has a predictable behavior when it comes to the pattern of activations that it needs before it sends the clock an update command. To capture this, [3] defines a function $\mathsf{predict\text{-}time}_{\Pi}(\vec{\mathcal{I}}_H^T)$ that predicts the time in which the clock is supposed to be according to the given protocol, given as input the timed honest-input sequence. [8] For self-containment, we restate this property formalized in [3] in Definition 7 in Section A.3, where we also prove that Ouroboros Genesis indeed satisfies it.

**Hash functions as global random oracles.** Ouroboros Genesis assumes that parties can query a hash function. As typically in cryptographic proofs the queries to hash function are modeled by assuming access to a random oracle (functionality): Upon receiving a query $(\text{EVAL}, \text{sid}, x)$ from a registered party, if $x$ has not been queried before, a value $y$ is chosen uniformly at random from $\{0,1\}^{\kappa}$ (for security parameter $\kappa$) and returned to the party (and the mapping $(x, y)$ is internally stored). If $x$ has been queried before, the corresponding $y$ is returned.

The random oracle is typically captured as a local UC functionality. As discussed in the introduction, this raises a number of issues, both with respect to how natural this abstraction of a hash function is, and with respect to the induced programmability that comes from this choice. Instead in this work we choose to capture it as a global setup, referred to as GRO and denoted as $\mathcal{G}_{\text{RO}}$ (see Section A.4 for a detailed description.) The fact that our security proof can be carried out in this model serves as an indication of the augmented composability that PoSs bring to the blockchain ecosystem. As mentioned before, Bitcoin cannot be proved secure in the GRO model.

**The genesis block generation and distribution.** Agreement on the first, so-called *genesis* block, is a necessary condition in all common blockchains for the parties to achieve eventual consensus. In Ouroboros Genesis, this block includes the keys, signatures, and original stake distribution of the parties that are around at the beginning of the protocol. This assumption— i.e., that the genesis block is properly created and distributed to the initial parties, and that it is properly distributed to anyone who joins even later—is captured in [14] by assuming access to a (local) functionality $\mathcal{F}_{\mathsf{INIT}}$. For each stakeholder registered at the beginning of the protocol, $\mathcal{F}_{\mathsf{INIT}}$ records his key in the genesis block; this block is distributed to anyone who requests it in any future round. To simplify the protocol description, we will assume throughout the paper that the first round—i.e., the genesis round—of the protocol occurs when the global time is $\tau = 0$. This is wlog as the actual genesis-round index is written on the genesis block and we assume that all parties are synchronized with the global clock. For completeness we include a description of $\mathcal{F}_{\mathsf{INIT}}$ in Section A.5.

**Hybrids used in the security proof.** Ouroboros Genesis requires only access to the above functionalities and global setups, i.e., $\mathcal{F}_{\text{N-MC}}^{\Delta}, \mathcal{F}_{\mathsf{INIT}}, \mathcal{G}_{\text{CLOCK}}$, and $\mathcal{G}_{\text{RO}}$. However, for a clearer protocol description it is convenient to assume hybrid access to two more functionalities, one that abstracts verifiable random functions (VRF), denoted as $\mathcal{F}_{\mathsf{VRF}}$, and another one that abstracts key-evolving signature schemes (KES), denoted as $\mathcal{F}_{\mathsf{KES}}$. They first appeared in [14] and naturally allow for a cleaner and more modular protocol description and proof. For completeness we include their description in Section A.6 and refer to [14] for concrete realizations.

### 2.2 The Ideal World Execution

We next turn to the functionalities available in the ideal-world. Recall that in this world, the parties execute the so-called dummy protocol. Since the clock and the random oracle are modeled as global setups, i.e.,

[8] The timed honest-input sequence looks like $\vec{\mathcal{I}}_H^T = ((x_1, \text{pid}_1, \tau_1), \ldots, (x_m, \text{pid}_m, \tau_m))$ where $((x_1, \text{pid}_1), \ldots, (x_m, \text{pid}_m))$ are the honest inputs corresponding to an execution (up to a certain point), and for each $i \in [n]$, $\tau_i$ is the time of the global clock when input $x_i$ was handed to $\text{pid}_i$.

$\mathcal{G}_{\text{CLOCK}}$ and $\mathcal{G}_{\text{RO}}$, they are available also in the ideal world. However, the big change in the ideal world, is that the Ouroboros Genesis protocol (and the corresponding network and initialization functionality) are replaced by the ideal functionality that abstracts the protocol's goals. We call this functionality the *(ideal) ledger* and formally specify it in the following.

**The Ledger Functionality.** The ledger that Ouroboros Genesis realizes is almost identical to the abstract ledger that was proved in [3] to be implemented by (the UC adaptation of) Bitcoin. In fact, the abstract ledger proposed in [3] is parameterizable by a collection of four algorithms. The ledger implemented by Ouroboros Genesis is effectively derived by appropriately instantiating these algorithms. This similarity can be seen as a confirmation of the ledger abstraction, and as an affirmation that Ouroboros Genesis meets strong composable security.

Given their common core, in order to describe the Ouroboros Genesis ledger its is helpful to start with a briefly recap of the abstract ledger from [3].

The ledger from [3] maintains a central and unique ledger state denoted by `state`. Each registered party can request to see the state, but is guaranteed to receive a only a sufficiently long prefix of it; the size of each party's view of the state is captured by (monotonically) increasing pointers that define which part of the state each party can read; the adversary has a limited control on these pointers. The dynamics of this can be seen as a sliding window over the sequence of state blocks, with width `windowSize` and starting at the head of the state, and each party's pointer points to a location withing this window. (The adversary can choose the position of the pointers within this sliding window.) As is common in UC, parties advance the ledger when they are instructed to (activated with specific maintain-ledger input by their environment $\mathcal{Z}$.) The ledger uses these queries along with the function $\mathsf{predict\text{-}time}(\cdot)$ to ensure that the ideal world execution advances with the same pace (relatively to the clock) as the protocol does.[9]

Any party can input a transaction to the ledger (upon instructed by $\mathcal{Z}$); upon reception, transactions are validated using a predicate $\mathsf{Validate}$ and, if found valid, are added to a buffer. Each new block of the state consists of transactions which were previously accepted to the buffer. (Note that transaction are treated as abstract objects/input-values.) To give protocols syntactic freedom of how a state block looks like, a vector of transactions, say $\vec{N}_i$ is mapped to the $i$th state block via function $\mathsf{Blockify}(\vec{N}_i)$. $\mathsf{Validate}$ and $\mathsf{Blockify}$ are two of the ledger's parametrization algorithms.

A defining part of the behavior of the ledger is the (parameterizable) procedure which defines when/how to extend `state`. One needs to allow the adversary enough influence, since this is the case in the real protocol, but the ledger should impose certain policies/restrictions regarding such extensions. For example it should require a minimum chain growth rate, a certain chain quality, and liveness of transactions, which are properties studied in [14] for Ouroboros Praos. The procedure $\mathsf{ExtendPolicy}$ is responsible for enforcing such a policy. In nutshell, to enable adversarial influence, $\mathsf{ExtendPolicy}$ takes as an input a proposal from the adversary for extending the state, and can decide to follow this proposal if it satisfies its policy; if it does not, $\mathsf{ExtendPolicy}$ can ignore the proposal (and enforce a default extension). This mechanism is flexible enough to model different kind of scenarios; in particular, as we show in this work, it enables to capture the composable guarantees of proof-of-stake as well.

SETTING THE LEDGER FUNCTIONALITY PARAMETERS. To specify the ledger achieved by Ouroboros Genesis, we need to instantiate the relevant parameters/procedures from above. $\mathsf{Blockify}$, $\mathsf{Validate}$, and $\mathsf{predict\text{-}time}$ are chosen to mimic the input/output format restrictions of the protocol; concretely, $\mathsf{Blockify} := \mathsf{blockify}_{\mathsf{OG}}$, $\mathsf{predict\text{-}time} := \mathsf{predict\text{-}time}_{\mathsf{OG}}$ (defined in Lemma 3), and

$$\mathsf{Validate}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) := \mathsf{ValidTx}_{\mathsf{OG}}(\mathtt{tx}, \mathtt{state}),$$

where $\mathsf{blockify}_{\mathsf{OG}}$, $\mathsf{predict\text{-}time}_{\mathsf{OG}}$, and $\mathsf{ValidTx}_{\mathsf{OG}}$ are identical to what the real protocol uses, whose description appears in Section 3. As in [3], $\mathsf{blockify}_{\mathsf{OG}}$ and $\mathsf{ValidTx}_{\mathsf{OG}}$ must not disqualify each other[10] (see [3, Definition 2]). This is easily ensured and also the case for Ouroboros Genesis.

[9] Recall that the clock waits (also) for the ledger to check-in to advance its time/round index.

[10] If they do, only empty state blocks would emerge

The procedure ExtendPolicy policy is trickier, but it again follows the same principles as in [3]. It enforces the following properties: First, all blocks of `state` are semantically valid. Furthermore, it ensures the following properties:

1. The state grows at a minimal rate of blocks over a time interval. This is formalized by specifying a value $\mathtt{maxTime}_{\mathtt{window}}$ in which at least `windowSize` blocks have to be inserted into the ledger state.
2. A substantial fraction of blocks added to the state are declared as being what we call "honestly generated" and formally have to fulfill higher standards than other blocks.[11] This fraction of good blocks is enforced by requiring a limit $\mathtt{advBlcks}_{\mathtt{window}}$ of adversarial blocks (i.e., contributed blocks that do not need to employ higher standards) in each window of `windowSize` state blocks.

Note that honestly generated blocks are crucial to ensure a liveness guarantee for transactions. The liveness guarantee captures that if a transaction is old enough and still valid, then it is guaranteed to be inserted into the state. This guarantee can be enhanced by using digital signatures in a modular next step, i.e., within a ledger-hybrid protocol. We refer to [3] for details. A detailed specification of the Ouroboros Genesis ExtendPolicy can be found in Section A.8.

Party sets maintained by the ledger. The analysis (and ledger) of [3], and also in this work, separates the honest parties into two different basic categories called *synchronized* and *desynchronized*. A party is considered synchronized if it has been continuously connected to all its resources for a sufficiently long interval and has maintained connectivity to these resources, except maybe the GRO, until the current time. Formally, here, "sufficiently long" refers to `Delay`-many rounds, where `Delay` is a parameter of the ledger that depends on the network delay (we defer a formal definition of (de)synchronized parties to Section 4.4); as a consequence, the chains held by synchronized parties satisfy common prefix property as defined in [17]. Honest parties that are registered to the ledger but are not synchronized are called desynchronized. Because we cannot guarantee that these parties' view is consistent with the rest of the honest network, the ledger assigns them less guarantees. As soon as the interval of `Delay` rounds from registration passes (during which the party needs to have continuous access to the GRO and the clock), these parties become synchronized and obtain the best guarantees the ledger can assign. This party set corresponds to what we call alert parties in this work (and will be outlined in more detail in the next section). We give a detailed overview of all relevant party sets, and how they are reflected in the ideal-world, in the next section.

### 2.3 On Modeling Dynamic Availability

As already discussed in the introduction, in this work we aim (and achieve) the highest granularity in the guarantees that honest parties receive, with respect to their availability status. To define the important classes, we first assign the following attributes to define an honest party's status at a given point in the execution: a party is considered *offline* if it is not registered with the network and otherwise it is considered *online*. A party is *time-aware* if it is registered with the clock. And finally, we say that a party is *operational* if it is registered with the second global setup, namely the random oracle (and otherwise considered as stalled). While this gives us a clean technical definition of being operational or stalled, it assigns a rather negative meaning to the term *stalled*. One could alternatively introduce an explicit protocol feature to allow a higher-level application to choose to stall the operation for a while (but still being connected to the blockchain network for example); we plan to consider such a change in a future revision. For a concise overview, we refer to Figure 1. Based on the four basic attributes mentioned, we can define the types or categories of parties discussed in the introduction. The basic types, such as online/offline parties or operational/stalled parties, simply refer to the subset of honest parties possessing a certain attribute at a given point in the execution.

We further define two more complex party types. First, we have the *alert* parties that can be considered the core set of honest parties establishing the desired properties of the blockchain and are those honest parties that have a synchronized state and are connected to all resources needed to execute the protocol.

[11] In reality, these are the blocks contributed by alert parties.

**Fig. 1. Classification of honest parties.** Based on access to resources (random oracle $\mathcal{G}_{\mathrm{RO}}$, clock $\mathcal{G}_{\textsc{clock}}$, network $\mathcal{F}_{\text{N-MC}}$) and presence in their current non-offline status for more than `Delay` rounds (synchronized or desynchronized).

| | **Basic types of *honest* parties** | |
|---|---|---|
| **Resource** | **Resource unavailable** | **Resource available** |
| random oracle $\mathcal{G}_{\mathrm{RO}}$ | *stalled* | *operational* |
| clock $\mathcal{G}_{\textsc{clock}}$ | *time-unaware* | *time-aware* |
| network $\mathcal{F}_{\text{N-MC}}$ | *offline* | *online* |
| synchronized state | *desynchronized* | *synchronized* |

**Derived party types.**

$$\begin{aligned} \textit{alert} &:\Leftrightarrow \textit{operational} \wedge \textit{time-aware} \wedge \textit{online} \wedge \textit{synchronized} \\ \textit{active} &:\Leftrightarrow (\textit{operational} \wedge \textit{time-aware} \wedge \textit{online}) \vee \textit{adversarial} \vee \textit{time-unaware} \end{aligned}$$

Note that while *alert* parties are honest, the set of *active* parties also contains all adversarial parties.

---

Next, in view of the security statements of later sections, we also define in Figure 1 the derived type of *potentially active* parties (or *active* for short). This class contains any (honest or corrupted) party that can potentially propose a block in its current status. Formally, it includes all honest parties with access to all resources (be they synchronized or not), all corrupted (i.e., adversarial) parties, and moreover any party that is time-unaware (independently of the other attributes). In the case of time-unaware parties, note that those parties are in particular not capable of evolving their signing keys reliably and hence it cannot be excluded that if they later get corrupted, they might retroactively perform protocol operations in a malicious way.

The definition of a party type is extended from a single point in an execution to a single round in an execution, where a round, say $t$, consists of all system configurations where the value of the clock is $t$. We say a party is alert in round $t$, if it is alert at any point in the execution when the clock's time is $t$. We say a party is potentially active in round $t$ if it is considered active at some point in the execution when the clock's time is $t$. We will recall the above classes and their respective role in our security arguments in Section 4 and Appendix E.

Ideal-world correspondence. As described in the previous section, as in [3], the ledger keeps an updated track of registered parties with all global setups so it can know which category each party belongs in. As described above, the ledger can thereby distinguish between desynchronized parties (with reduced security guarantees), stalled parties (that obtain no output), alert parties (full security guarantees), and offline parties (which are ignored). We note in passing that, although not included in [3], an analogous level of granularity is an interesting extension to the Bitcoin analysis too. In fact, as an exercise the reader can be convinced that Bitcoin does also implement the ledger with respect to such a fine-grained, dynamic-availability model.[12]

A minor deviation: Fitting the functionality to the PoS setting. There is one minor point where the PoS ledger needs to deviate from the Bitcoin one. Concretely, in Bitcoin the contents of the genesis block are irrelevant (i.e., the ledger can simply have this block hardwired.) However, in PoS it is inherent that the initial stake (or tokens) is distributed in a trustworthy manner. This is reflected in the need for initialization, where the parties associated to this setup need to register in the very first round. To make sure that the ledger execution is indistinguishable from Ouroboros Genesis, we equip the ledger with an additional parameter, the initial stakeholders set and corresponding stake distribution $\mathcal{S}_{\text{initStake}} := \{(U_1, s_1), \ldots, (U_n, s_1)\}$. If some honest stakeholder abstains from registering in the first round, the ledger stops execution. In this work, this parameter is defined to be identical to the stake-distribution parameter of $\mathcal{F}_{\mathsf{INIT}}$.

[12] Recall that in [3], a party is never stalled. If it is not offline, and hence contributes to the overall hashing power, it either belongs to the synchronized or to the de-synchronized set (and de-synchronized parties increase adversarial power).

Given its strong similarities with the abstract ledger from [3], the complete and formal specification of the concrete ledger that Ouroboros Genesis realizes can be found in Section A.7.

## 3 Ouroboros Genesis as a UC-Protocol

In this section we provide a detailed description of our protocol Ouroboros-Genesis as a synchronous (G)UC protocol. The protocol has a similar structure as Ouroboros Praos [14], but differs considerably in the novel chain selection rule, which allows parties to join at any point without the need of external checkpointing. As already discussed, the protocol only assumes access to the network functionalities and global setups, i.e., $\mathcal{F}_{\text{N-MC}}^{\Delta}, \mathcal{F}_{\mathsf{INIT}}, \mathcal{G}_{\text{CLOCK}}$, and $\mathcal{G}_{\text{RO}}$. However, for clarity we describe the protocols as having access to two additional functionalities $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$; as mentioned in the Section 2.1, these latter two functionalities can be implemented using the former.

The section is organized as follows: First we discuss how the hybrids are used and provide a high level description of the protocol. Then we proceed to the detailed protocol specification.

**Protocol overview.** The protocol Ouroboros-Genesis assumes as hybrids a network $\mathcal{F}_{\text{N-MC}}^{\Delta}$, a verifiable random function $\mathcal{F}_{\mathsf{VRF}}$, a key-evolving signature scheme $\mathcal{F}_{\mathsf{KES}}$, a global random oracle $\mathcal{G}_{\text{RO}}$, and a global clock $\mathcal{G}_{\text{CLOCK}}$.

The protocol execution proceeds in disjoint, consecutive time intervals called *slots*. Importantly, time is divided in such a way that all parties know when a new slot starts—in our specification, every slot is one round, hence the parties can compute the current slot by comparing the round, i.e., clock value, recorded on the genesis block with the current round. Without loss of generality we will assume that the protocols starts when the global time is $\tau = 0$; in this case the current slot index will always be $\tau$.

In each slot $\mathtt{sl}$, the parties execute a so-called *staking procedure* to extend the blockchain. At a high level, the staking procedure consists of the following steps: First, the parties execute an implicit lottery to elect a *slot leader* from a distribution which, roughly, is biased by the stake distribution—the more stake a party has in the system, the more likely he is to be elected slot leader.

In any given slot, the elected slot leaders are in charge of extending the blockchain. Concretely, slot leaders are allowed to propose an updated blockchain. To this direction, the slot leader creates and signs a block for the current slot. Each such block contains transactions that may move stake among stakeholders. The slot leader then broadcasts the new chain extended by its block to its peers via $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$. We remark that as in [14], in order to achieve adaptive security the blocks are signed using a key-evolving signature scheme $\mathcal{F}_{\mathsf{KES}}$ instead of a standard signature, and honest parties are mandated to update their private key in each slot.

A chain proposed by any party might be adopted only if it satisfies the following two conditions: (1) it is valid according to a well defined validation procedure, and (2) the block corresponding to each slot is signed by a corresponding certified slot leader.

To ensure the second property we need the implicit slot-leader lottery to provide its winners (slot leaders) with a certificate/proof of slot-leadership. For this reason, we implement the slot-leader election as follows: Each party $U_p$ checks whether or not it is a slot leader, by locally evaluating a verifiable random function (VRF, [15], modeled by $\mathcal{F}_{\mathsf{VRF}}$) using the secret key associated with its stake, and providing as inputs to the VRF both the slot index $\mathtt{sl}$ and the so-called epoch randomness $\eta$ (we will discuss shortly where this randomness comes from). If the VRF output $y$ is below a certain threshold $T_p$—which depends on $U_p$'s stake—then $U_p$ is an eligible slot leader; furthermore, he can use the verifiability of the VRF to generate a proof $\pi$ of the function's output, thereby certifying his own eligibility to act as a slot leader. In particular, in addition to transactions, each new block broadcast by a slot leader also contains the VRF output $y$ and a proof $\pi$ of its validity to certify the party's eligibility to act as a slot leader.

Using the output of a VRF to identify the slot leaders as above not only allows for certifying the winner, but it also ensures that slot leaders are chosen from the appropriate distribution. In a nutshell, this is achieved as follows: Multiple slots are collected into *epochs*, each of which contains $R \in \mathbb{N}$ slots.[13] The idea

[13] Unlike [14], where R is fixed, in this work we treat $R$ as a protocol parameter, which will be bounded appropriately by our security statements.

of having epochs is that it allows to use stake reference points that are old enough to be stable—with high probability—and are therefore appropriate to be used in a universally verifiable proof. Concretely, during an epoch $\mathtt{ep}$, the stake distribution $\mathbb{S}_{\mathtt{ep}}$ that is used for deriving the threshold $T_p^{\mathtt{ep}}$ used for the slot-leader election corresponds to the distribution recorded in the ledger up to the last block of epoch $\mathtt{ep}-2$. Additionally, the *epoch randomness* $\eta_{\mathtt{ep}}$ for sampling slot leaders in epoch $\mathtt{ep}$ is derived as a hash of additional VRF-values $y_\rho$ that were included (together with their respective VRF-proofs $\pi_\rho$) into blocks from the first two thirds of epoch $\mathtt{ep}-1$ for this purpose by the respective slot leaders. (To unify block structure, our protocol includes these values into *all* blocks, but this would not be necessary in practice.) The values $\mathbb{S}_{\mathtt{ep}}$ and $\eta_{\mathtt{ep}}$ are updated at the beginning of each epoch.

A delicate point of the above staking procedure is that there will inevitably be some slots with zero or several slot leaders. This means that the parties might receive valid chains from several certified slot leaders. To determine which of these chains to adopt as the new state of the blockchain, each party collects all valid broadcast chains and applies a chain selection rule $\mathsf{maxvalid\text{-}bg}$. In fact, the power of the protocol $\mathsf{Ouroboros\text{-}Genesis}$ and its superiority over all existing PoS-based blockchains stems from this new chain-selection rule which we discuss in detail below.

We next turn to the formal specification of the protocol $\mathsf{Ouroboros\text{-}Genesis}$. The protocol describes the code that each party $U_p$ executes. Recall that in UC parties can be dynamically created by the environment; upon its creation a party is assigned a session ID, sid, and connects to all global setups, to the adversary, and to all functionalities with which it shares the same session ID sid. Then the party becomes idle (releases the activation) and waits for the environment's input or for a message by a party with which it has been connected. (Using a standard UC convention, we assume that newly created parties do not register to any functionality or setup unless they are explicitly instructed to, by receiving a special input from their environment. Thus the party generation process is decoupled from the protocol itself.)

To make the protocol description modular, we describe different components as subprotocols and include in their header the parameters they need to be aware of. All protocols described here are $\{\mathcal{G}_{\mathrm{CLOCK}}, \mathcal{G}_{\mathrm{RO}}, \mathcal{F}^{\Delta}_{\mathrm{N\text{-}MC}}, \mathcal{F}_{\mathsf{INIT}}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}}\}$-hybrid protocols, i.e., have access to all these functionalities (and protocol participants share the same session ID with all local functionalities in this set.)

### 3.1 The Formal Protocol Description

We start with some notation. We use $x \prec y$ to indicate that the string $x$ is a prefix of the string $y$. Consider an arbitrary partitioning of the time axis into subsequent, non-overlapping, equally long intervals called *slots*. For the purpose of this section, a *block* is an arbitrary piece of data that contains an identification of a time slot to which it belongs. A blockchain (or *chain*, for short) is a sequence of blocks with increasing time slots, starting with a special *genesis block* and with each subsequent block containing a hash of the previous one. A more concrete description of blocks and chains created by the Ouroboros Genesis protocol will be given in Section 3.

We denote the length of a chain $\mathcal{C}$ (i.e., the number of its blocks) by $\mathsf{len}(\mathcal{C})$. For a chain $\mathcal{C}$ and an interval of slots $I \triangleq [\mathtt{sl}_i, \mathtt{sl}_j]$, we denote by $\mathcal{C}[I] = \mathcal{C}[\mathtt{sl}_i : \mathtt{sl}_j]$ the sequence of blocks in $\mathcal{C}$ such that their slot numbers fall into the interval $I$. We replace the brackets in this notation with parentheses to denote intervals that do not include endpoints; e.g., $(\mathtt{sl}_i, \mathtt{sl}_j] = \{\mathtt{sl}_i + 1, \ldots, \mathtt{sl}_j\}$. Finally, we denote by $\#_{i:j}(\mathcal{C}) \triangleq \#_I(\mathcal{C}) \triangleq |\mathcal{C}[I]|$ the number of blocks in $\mathcal{C}[I]$.

Before giving the formal specification we introduce some necessary terminology and notation. Each party U stores a local blockchain $\mathcal{C}_{\mathsf{loc}}^{U_p}$—$U_p$'s local view of the blockchain.[14] Such a local blockchain is a sequence of blocks $B_i$ ($i > 0$) where each $B \in \mathcal{C}_{\mathsf{loc}}$ has the following format: $B = (h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$. The first block $B_0$ is special and is referred to as the *genesis block* $\mathbf{G}$. In each following block $B_i, i > 0$, $h$ is a hash of the previous block, $\mathtt{st}$ is the encoded data of this block, and $\mathtt{sl}$ is the slot number this block belongs to. The value $crt = (U_p, y, \pi)$ certifies that the block was indeed proposed by an eligible slot leader $U_p$ for slot $\mathtt{sl}$ by providing the output $y$ of $U_p$'s VRF evaluation for this slot, along with the corresponding VRF proof $\pi$.

[14] For brevity, wherever clear from the context we omit the party ID from the local chain notation, i.e., write $\mathcal{C}_{\mathsf{loc}}$ instead of $\mathcal{C}_{\mathsf{loc}}^{U}$.

Additionally, $\rho = (y_\rho, \pi_\rho)$ is an independent VRF output—along with its proof—that is also inserted into the block by $U_p$ and is later used to derive the future epoch randomness. Finally, $\sigma$ is the signature by $U_p$ on the entire block (using a key-evolving signature scheme).

If $\mathcal{C}_{\mathsf{loc}} = B_0||\cdots||B_\ell$ is a (local) chain, we define its associated *encoded state* $\vec{\mathtt{st}}$ as the sequence $\mathtt{st}_0||\ldots||\mathtt{st}_\ell$, where each $\mathtt{st}_i$—referred to as the $i$th *state block* of the state—is the encoded data stored in block $B_i$. (The genesis data is defined to be $\mathtt{st}_0 := \varepsilon$.) The *exported state* is then a specific prefix $\vec{\mathtt{st}}^{\lceil k}$ of this state (we define this expression to be $\varepsilon$ if $k$ is larger than the size of the chain). The exact format of the state blocks depends on the actual implementation and is enforced by use of the function $\mathsf{blockify}_{\mathsf{OG}}$. Concretely, each state block $\mathtt{st}$ is formed by applying this predicate on a vector $N$ of transactions to derive an appropriately formatted version of the block. This parameterization allows flexibility in the way the exported state is formatted.

To enable dynamic availability every party stores in a variable $t_{\text{on}}$ (initially set to 1) the time/slot it was last online (and not stalled). It also store in a variable $t_{\text{work}}$ (initially set to 0) the last time when the staking procedure run to completion. Every protocol machine also stores the current (local) state $\vec{\mathtt{st}}$ encoded in the chain $\mathcal{C}_{\mathsf{loc}}$ and the local buffer `buffer` (corresponding to the transactions seen so far on the network and not added on the blockchain); $\vec{\mathtt{st}}, \mathcal{C}_{\mathsf{loc}}$ and `buffer` are all initially empty.

For brevity, whenever in the protocol we say that a party *uses the clock to update,* $\tau$, $\mathtt{ep}$, *and* $\mathtt{sl}$ we mean the following step:

- Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$; receive the current time $\tau$ and update $\mathtt{ep} := \lceil \tau/R \rceil$ and slot index $\mathtt{sl} = \tau$, accordingly.[15]

**Handling interrupts in a UC protocol.** A protocol command might consists of a sequence of operations. In UC, certain operations, such as sending a message to another party, outputting a message to the environment, or the inability to conclude a task because a resource is unavailable, result into the protocol machine having to loose its activation. Thus, one needs a mechanism for ensuring that a party that looses the activation in the middle of such a multi-step command is able to resume and complete this command. Such a mechanism is implicitly described in [21]. This mechanism can be made explicit by introducing an anchor $a$ that stores a pointer to the current operation; the protocol associates each anchor with such a multiple command and an input $I$, so that when such an input is received it directly jumps to the stored anchor, executes the next operation(s) and updates (increases) the anchor before releasing the activation. We refer to execution in such a manner as *I-interruptible.*

For clarity we include an example of an interruptible execution. Assume that the protocol mandates that upon receiving input $I$, the party should run a command that consists of $m$ steps Step 1, Step $2, \ldots,$ Step $m$, but some of these steps might result in the executing party releasing its activation. Running this command in an *I-interruptible* manner means executing the following code: Upon receiving input $I$ if $a < m$ go to Step $a$ and increase $a = a + 1$ before executing the first operation that releases the activation; otherwise go to Step 1 and set $a = 2$ before executing any operation that releases the activation.

The Ouroboros Genesis protocol is described in detail in Figure 2. Aside of the core operation of the protocol, the description already includes a block of commands (in the bottom of the description) which specify what parties do when they receive external queries to their global setups, such as queries to the global random oracle. Note that since the ideal-world (dummy) parties would forward such queries to their setups, the protocol needs to do basically this, except possibly for some book-keeping actions in case they are not fully operational (such as making sure that at least the signing key evolves before moving to the next round).

### 3.2 Registration and Deregistration

The first thing a party needs to do in order to have any role in the protocol is register with its resources. Registration (and deregistration) is dictated to the (honest) parties by the environment. This captures the

[15] Recall that we assume for simplicity that the protocol starts when $\tau = 0$ and that $R$ is a protocol parameter defining the duration of an epoch (in rounds).

**Protocol** $\mathsf{Ouroboros\text{-}Genesis}_k(U_p, \mathrm{sid}; \mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}, \mathcal{F}^{\Delta}_{\text{N-MC}})$

**Global Variables:**
- Read-only (parameters): $R$, $k$, $f$, $s$
- Read-write: $v_p^{\text{vrf}}$, $v_p^{\text{kes}}$, $\tau$, $\mathtt{ep}$, $\mathtt{sl}$, $\mathcal{C}_{\mathsf{loc}}$, $T_p^{\mathsf{ep}}$, $\mathtt{isInit}$, $t_{\text{on}}$, $t_{\text{work}}$, $\mathtt{buffer}$

**Registration/Deregistration *(cf. Section 3.2):***
- Upon receiving input $(\text{REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol $\mathsf{Registration\text{-}Genesis}(U_p, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\text{DE-REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol $\mathsf{Deregistration\text{-}Genesis}(U_p, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\text{IS-REGISTERED}, \mathrm{sid})$ return $(\text{REGISTER}, \mathrm{sid}, 1)$ if the local registry $\mathtt{Reg}$ indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}_{\text{LEDGER}}$ (and did not de-register since then). Otherwise, return $(\text{REGISTER}, \mathrm{sid}, 0)$.

**Interacting with the Ledger *(cf. Section 3.3):***
Upon receiving a ledger-specific input $I \in \{(\text{SUBMIT}, \ldots), (\text{READ}, \ldots), (\text{MAINTAIN-LEDGER}, \ldots)\}$ verify first that all resources are available. **If** not all resources are available, **then** ignore the input; **else** (i.e., the party is operational and time-aware) execute one of the following steps depending on the input $I$:

- **If** $I = (\text{SUBMIT}, \mathrm{sid}, \mathtt{tx})$ **then** set $\mathtt{buffer} \leftarrow \mathtt{buffer}||\mathtt{tx}$, and send $(\text{MULTICAST}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{F}^{\Delta}_{\text{N-MC}}$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \mathrm{sid}, \mathrm{minerID})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, U_p, \mathrm{sid}, k, s, R, f)$; if $\mathsf{LedgerMaintenance}$ halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\text{READ}, \mathrm{sid})$ **then** invoke protocol $\mathsf{ReadState}(k, \mathcal{C}_{\mathsf{loc}}, U_p, \mathrm{sid}, R, f)$.

**Handling external (protocol-unrelated) calls to the clock and the RO:**
- Upon receiving $(\text{CLOCK-READ}, \mathrm{sid}_C)$ forward it to $\mathcal{G}_{\text{CLOCK}}$ and output $\mathcal{G}_{\text{CLOCK}}$'s response.
- Upon receiving $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$, record that a clock-update was received in the current round. If this instance is currently time-aware but otherwise stalled or offline, then evolve the KES signing key by sending $(\mathsf{USign}, \mathrm{sid}, U_p, 0, \tau)$ to $\mathcal{F}_{\mathsf{KES}}$, where $\tau$ is the current time, and forward $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Furthermore, consider any active interruptible execution as completed.
- Upon receiving $(\text{EVAL}, \mathrm{sid}_{RO}, x)$ forward the query to $\mathcal{G}_{\text{RO}}$ and output $\mathcal{G}_{\text{RO}}$'s response.

**Fig. 2.** The Ouroboros Genesis Protocol

fact that resource availability is not something controlled by the protocol itself. For example, a crash of the timing or hashing process of the party's computer is captured by the environment instructing the party to deregister from the clock or the GRO, respectively. To capture our high-resolution (dynamic) availability, the environment is allowed to register and deregister parties from any of the resources at will.

In the following we describe the protocol that the parties execute upon receiving a registration/deregistration request. For clarity, we assume that every party keeps a local registry, denoted by $\mathtt{Reg}$, that includes a registration-flag for each of the functionalities (local and global) the party is connected to; whenever the party registers or deregisters with some functionality/setup the corresponding flag is updated accordingly. The protocols for registration and deregistration are described in the following. Since such commands are addressed to setups or to the ledger, they are only effecting in the real-world protocol if they are addressed to one of the functionalities/setups that are present, i.e., to some $\mathcal{G} \in \{\mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}, \mathcal{G}_{\text{LEDGER}}\}$. Any registration input with session ID different than that of those three functionalities will be ignored by the protocol. Without loss of generality, we do not write the session IDs of global setups and refer to them simply with their name.

### 3.2.1 Registration

The registration with any of the global setups $\mathcal{G}_{\text{RO}}$ and $\mathcal{G}_{\text{CLOCK}}$ is straightforward. However, registering with the ledger is a little more complicated. Upon receiving a ledger-registration query from the environment, the party first checks that it is registered with the global functionalities $\mathcal{G}_{\text{RO}}$ and $\mathcal{G}_{\text{CLOCK}}$. If not, then it ignores the input (and is still considered offline). Otherwise, it registers with each functionality—excluding the already registered-to global setup functionalities $\mathcal{G}_{\text{RO}}$ and $\mathcal{G}_{\text{CLOCK}}$. Moreover, once a party registers with its network it also stores the current time in variable $t_{\text{on}}$. (Recall that $t_{\text{on}}$ stores the last time the party was online, i.e., connected to all its resources.)

Note that the registration to and from the global functionalities has to stay under the control of the environment. Only once this procedure is completed, the party becomes operational and otherwise is considered de-registered and does not answer any ledger-specific queries (i.e., it is offline). The activation after any (de)registration goes back to the environment. The registration process is detailed in Figure 3.

**Protocol** Registration-Genesis$(U_p, \text{sid}, \texttt{Reg}, \mathcal{G})$

**if** $\mathcal{G} \in \{\mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}\}$ **then** send $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
**end if**
**if** $\mathcal{G} = \mathcal{G}_{\text{LEDGER}}$ **then**
    **if** the party is not registered with $\mathcal{G}_{\text{CLOCK}}$ or $\mathcal{G}_{\text{RO}}$ or is already registered with all setup functionalities **then** ignore this input
    **else**
        **for** each $\mathcal{F} \in \{\mathcal{F}_{\mathsf{INIT}}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}}\}$ **do**
            Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}$, set its registration status as registered with $\mathcal{F}$, but do not output the received values.
        **end for**
        Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and receive and store the current time $\tau$.
        Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}^{\Delta}_{\text{N-MC}}$ and set $t_{\text{on}} \leftarrow \tau$.
        Output $(\text{REGISTER}, \text{sid}, U_p)$ once completing the registration with all the above resources $\mathcal{F}$.
    **end if**
**end if**

**Fig. 3.** The registration process.

### 3.2.2 De-registration

The deregistration process is described in Figure 4. De-registering from global setups is analogous to above, while for simplicity, we model de-registering from the "ledger" as disconnecting from the network and hence becoming an offline party.

## 3.3 Interacting with the Ledger

At the core of the Ouroboros Genesis protocol is the process that allows parties to maintain the ledger. There are three types of processes that are triggered by three different commands provided that the party is already registered to all its local and global functionalities—if this in not the case, the corresponding command is ignored.[16]

[16] Recall that our ledger functionality ensures that a parties input is considered—not ignored—only if this party is registered with all its global inputs (see Appendix A.7 for details.)

**Protocol** Deregistration-Genesis($U_p$, sid, $\mathtt{Reg}$, $\mathcal{G}$)

1: **if** $\mathcal{G} \in \{\mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}\}$ **then** Send (DE-REGISTER, sid) to $\mathcal{G}$, set registration status as de-registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
**end if**
2: **if** $\mathcal{G} = \mathcal{G}_{\text{LEDGER}}$ **then**
Send (DE-REGISTER, sid) to $\mathcal{F}_{\text{N-MC}}^{\Delta}$, set its registration status as de-registered with $\mathcal{F}_{\text{N-MC}}^{\Delta}$ and output (DE-REGISTER, sid, $U_p$).
**end if**

**Fig. 4.** The deregistration process.

- The command (SUBMIT, sid, $\mathtt{tx}$) is used for sending a new transaction to the ledger (to be included in one of the upcoming blocks). It results in the party storing the submitted transaction in its local transaction buffer and multicasting it to the network so that other parties also add it to their buffers.
- The command (READ, sid) is used for the environment to ask for a read of the current ledger state. It results in the party outputting a prefix $\vec{\mathtt{st}}^{\lceil k}$ of the state $\vec{\mathtt{st}}$ extracted from its most recently updated (local) blockchain. As we argue any such output will be a prefix of any output given by any other party (this will follow from the common-prefix property).
- The command (MAINTAIN-LEDGER, sid, minerID) triggers the main ledger update and maintenance procedure which is the most involved part. A party receiving this command first fetches from its network all information relevant for the current round, then it uses the received information to update its local info—i.e., asks the clock for the current time $\tau$, updates its epoch counter $\mathtt{ep}$, its slot counter $\mathtt{sl}$, and its (local view of) stake distribution parameters, accordingly; and finally it invokes the staking procedure unless it has already done so in the current round. If this is the first time that the party processes a (MAINTAIN-LEDGER, sid, minerID) message then before doing anything else, the party invokes an initialization protocol to receive the initial information it needs to start executing the protocol—in particular the genesis block. Furthermore, in order accommodate stalled parties, if the party is registered with the network but not with all other setups, this stalled party remembers the time it was stalled and returns the activation back to the environment. Also, since a stalled party remembers the last time it was online—thereby also the time it became stalled—in variable $t_{\text{on}}$, once such a party gets reconnected—i.e., re-registers with the ledger in the ideal world (resp. with the network, the VRF and the KES in the real world)—then upon its next activation to maintain the ledger, the party fetches all messages it has missed by comparing the current time $\tau$ to $t_{\text{on}}$ and querying the network the corresponding number of times. Details of this procedure are given in Section 3.3.2.

The relevant sub-processes involved in the handling of a MAINTAIN-LEDGER query are detailed in the following Sections 3.3.1 to 3.3.4. After introducing each of these basic ingredients, we conclude with a technical overview of the main ledger maintenance protocol LedgerMaintenance in Figure 12 and a detail specification of the protocol ReadState for answering requests to read the ledger's state (see Figure 13.)

### 3.3.1 Party Initialization

A party that has been registered with all its resources and setups becomes operational by invoking the initialization protocol Initialization-Genesis upon processing its first MAINTAIN-LEDGER command (see Figure 5 for detailed description). As a first step the party receives its keys from $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$. Subsequently, protocol Initialization-Genesis proceeds in one of the following two modes depending on whether or not the current round is the genesis round. Concretely:

- In the *genesis mode,* which is only executed during the genesis round $\tau = 0$, the party interacts with the initialization functionality $\mathcal{F}_{\mathsf{INIT}}$ to claim its stake.

– In the non-genesis mode, i.e., when $\tau > 1$, the protocol Initialization-Genesis queries $\mathcal{F}_{\mathsf{INIT}}$ to receive the genesis block and uses the received stake distribution to determine the initial threshold $T_p^{\mathsf{ep}}$ for each stakeholder $U_p$. Additionally, in order for the party to receive transactions and chains that were circulated over the network prior to this current round, the party multicasts a special message HELLO upon its first maintain-ledger activation (in addition to its normal round messages). Looking ahead, any $U_p$ receiving this message will set a special WELCOME flag to 1 will trigger (at first chance) $U_p$ to multicast his local buffer and chain; receiving these messages will enable the newly joining party to get up to speed. Recall that in order to ensure that the genesis round has been completed (and all initial stakeholders have claimed their stake) before the protocol starts advancing, the functionality $\mathcal{F}_{\mathsf{INIT}}$ throws an exception (halts with an error) if the environment does not allow all stakeholder to claim their stake in the genesis round. If this occurs, the calling protocol (i.e., Ouroboros Genesis) also halts (cf. Figure 2).

Independent of the round, the protocol concludes with the party setting `isInit` $\leftarrow$ true (to make sure that it is never re-initialized) and $t_{\mathrm{on}} \leftarrow \tau$ to remember the last time it became online—which in this case is also the first one.

**Protocol** Initialization-Genesis$(U_p, \mathrm{sid}, R)$

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

```
Send (KeyGen, sid, U_p) to F_VRF and F_KES; receiving (VerificationKey, sid, v_p^vrf) and (VerificationKey, sid, v_p^kes),
respectively.
// The following branch is executed on the first maintenance query after registration of this instance at
time 0.
if τ = 0 then
    Send (ver_keys, sid, U_p, v_p^vrf, v_p^kes) to F_INIT to claim stake from the genesis block.
    Invoke FinishRound(U_p) // Resume below on next maintain-ledger activation.
    Invoke UpdateTime(U_p) to update τ, ep, and sl.
    while τ = 0 do
        Call UpdateTime(U_p) to update τ, ep, and sl. and give up the activation (set anchor here)
    end while
end if
// The following executed if this is a non-genesis round
if τ > 0 then
    if F_INIT signals an error then
        Halt the execution.
    end if
    Send (genblock_req, sid, U_p) to F_INIT.
```

Receive from $\mathcal{F}_{\mathsf{INIT}}$ the response $(\mathsf{genblock}, sid, \mathbf{G} = (\mathbb{S}_1, \eta_1))$, where

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

```
    Set C_loc ← (G).
    Set T_p^ep ← 2^{ℓ_VRF} φ_f(α_p^ep) as the threshold for stakeholder U_p for epoch ep, where α_p^ep is the relative stake of
    stakeholder U_p in S_ep and ℓ_VRF denotes the output length of F_VRF.
    Send (HELLO, sid, U_p, v_p^vrf, v_p^kes) to F_N-MC^new.
end if
Set isInit ← true, t_on ← τ, and t_work ← 0.
```

GLOBAL VARIABLES: The protocol stores the list of variables $v_p^{\mathrm{vrf}}, v_p^{\mathrm{kes}}, \tau, \mathtt{ep}, \mathtt{sl}, \mathcal{C}_{\mathsf{loc}}, T_p^{\mathsf{ep}}$, `isInit`, $t_{\mathrm{on}}$ to make each of them accessible by all protocol parts.

**Fig. 5.** The initialization protocol of Ouroboros Genesis (run only the first time a party joins).

### 3.3.2 Fetching Information from the Network

The first thing that an already initialized (and fully online) party does is to attempt to read its incoming messages. Recall that in our network setting, a party accesses its network interface by sending a FETCH command to its network. A network latency of, say, $\Delta$ rounds, in the delivery of any given messages is then captured by the network withholding this message until $\Delta$ FETCH commands are issued (cf. [21]). In order to ensure that parties which have been stalled (but were not taken offline) can catch up with the messages sent to them while they where stalled, we use the following mechanism. The party first gets the current time $\tau$ from the clock, and then sets a counter $\mathsf{fetchcount}$ to $\tau - t_{\text{on}}$. (Since $t_{\text{on}}$ stores the last round that the party was online, $\mathsf{fetchcount}$ will be the number of rounds this party was stalled.) Subsequently the party issues $\mathsf{fetchcount}$ FETCH-queries to its network. Recall that a party that was offline and becomes online is considered de-synchronized for (at least) as many rounds as it needs for that party to receive all the relevant information and for the chain-selection rule to bootstrap it[17] —by detecting a chain that is guaranteed to originate from an honest and synchronized party. This party does not get to retrospectively receive messages sent to it while it was offline, which is is reflected in our protocol by the fact that this party will execute the network-registration procedure from scratch and will therefore set $t_{\text{on}} = \tau$.

There are three types of messages that are exchanged through the network, namely: blockchains—e.g,. when a slot leader creates a new block; regular messages, also referred to as transactions—which are broadcasted to the network when received by the environment; and HELLO-messages, as described above, sent by newly joining parties. To simplify the exposition, in our description we make the convention that each of these three types of messages is multicasted by its own network. Concretely, we will assume a network used for disseminating transactions, denoted as $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$, a network used for circulating HELLO message, denoted as $\mathcal{F}^{\mathsf{new}}_{\text{N-MC}}$, and a network used for disseminating other information (in particular new blockchains) as $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$. We stress that this distinction of networks is only for sake of clarity, as these three networks can be simulated over the original multicast network $\mathcal{F}_{\text{N-MC}}$ by appending a special identifier indicating the type of the exchanged message.

The protocol $\mathsf{FetchInformation}$ performing the above operations can be found in Figure 6.

**Protocol** $\mathsf{FetchInformation}(U_p, \text{sid})$

1: Define $\mathsf{fetchcount} := \tau - t_{\text{on}}$ // Difference of actual time and last time online when fetched
   // Fetching on $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$.
2: Send $\mathsf{fetchcount}$ fetch-queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$; denote the $i$th response from $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ by $(\text{FETCH}, \text{sid}, b_i)$.
3: Extract chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from $b_1 \ldots b_{\mathsf{fetchcount}}$.
   // Fetching on $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
4: Send $\mathsf{fetchcount}$ fetch-queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$; denote the $i$th response from $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ by $(\text{FETCH}, \text{sid}, b_i)$.
5: Extract received transactions $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ from $b_1 \ldots b_{\mathsf{fetchcount}}$.
   // Fetching on $\mathcal{F}^{\mathsf{new}}_{\text{N-MC}}$.
6: Send $\mathsf{fetchcount}$ fetch-queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}^{\mathsf{new}}_{\text{N-MC}}$.
7: **if** a message $(\text{HELLO}, \text{sid}, \cdot)$ was received **then**
      set WELCOME $= 1$
8: **else**
      set WELCOME $= 0$
   **end if**

OUTPUT: The protocol outputs $(\mathcal{C}_1, \ldots, \mathcal{C}_k)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$, and WELCOME to its caller (but not to $\mathcal{Z}$) if $\mathsf{fetchcount} > 0$ and otherwise provide an empty output.

**Fig. 6.** Fetching new information circulated through the multicast network.

[17] We give concrete bounds on the time it needs to become synchronized in Section 4.

### 3.3.3 The Staking Procedure

The next part of the ledger-maintenance protocol is the staking procedure which is used for the slot leader to compute and send the next block.

Recall that a party $U_p$ is an eligible slot leader for a particular slot $\texttt{sl}$ in an epoch $\texttt{ep}$ if its VRF-output (for an input dependent on $\texttt{sl}$) is smaller than a threshold value $T_p^{\mathsf{ep}}$. We next discuss how this threshold is computed for the party's current (local) blockchain, where we use the following notation: $\ell_{\mathsf{VRF}}$ denotes the VRF output length in bits. The (local) stake distribution $\mathbb{S}_{\mathsf{ep}}$ at epoch $\texttt{ep}$ corresponding to the (local) blockchain $\mathcal{C}_{\mathsf{loc}}$ is a mapping from a party (identified by its public keys) to its stake and can be derived solely based on encoded transactions in $\mathcal{C}_{\mathsf{loc}}$ (and the genesis block).[18] The relative stake of $U_p$ in the stake distribution $\mathbb{S}_{\mathsf{ep}}$, denoted as $\alpha_p^{\mathsf{ep}} \in [0,1]$, is the fraction of stake that is associated with this party (more precisely, its public key) in $\mathbb{S}_{\mathsf{ep}}$ out of all stake. The mapping $\phi_f(\cdot)$ is defined as

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

and is parametrized by a quantity $f \in (0,1]$ called the *active slots coefficient* [14], which is an important parameter of the protocol $\mathsf{Ouroboros\text{-}Genesis}$ (cf. Section 3.3.3).

Given the above, the threshold $T_p^{\mathsf{ep}}$ is determined as

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

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 $\phi$ has an important property called "independent aggregation" [14]:

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

In particular, when leadership is determined according to $\phi_f$, 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. Therefore, we can conclude that under arbitrary stake distribution, a particular slot has *some* slot leader with probability $f$, giving the active slots coefficient its intuitive meaning.

The technical description of the staking procedure appears in Figure 7. It starts by two calls evaluating the VRF in two different points, using constants $\texttt{NONCE}$ and $\texttt{TEST}$ to provide domain separation, and receiving $(y_\rho, \pi_\rho)$ and $(y, \pi)$, 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 as described in Section 3.1. The information includes the proof of leadership $(y, \pi)$, additional VRF-output $(y_\rho, \pi_\rho)$ that influences the epoch-randomness for the next epoch, and the block signature $\sigma$ produced[19] 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 $B$).

**Transaction Validity.** Blockchain ledgers typically put restrictions on transactions that can be added to a block. For example, Bitcoin only allows transactions that are properly signed and are spending an unspent coin. Although this is not directly related to the consistency guarantees, similarly to [3], our ledger also has such a transaction filter in place (this makes it suitable for applications like cryptocurrencies). This filter is

[18] The exact encoding is not of primary relevance as long as it gives rise to the aforementioned mapping to stake distributions (which therefore also binds public keys to identities). A possible, straightforward instantiation of an encoding is explained in [14].

[19] Note that signing is a local operation and reflected in this work by requiring $\mathcal{F}_{\mathsf{KES}}$ to be responsive as explained in Section A.6.

implemented by means of a predicate $\mathsf{ValidTx}_{\mathsf{OG}}$. To decide which transactions can be included in the state of a new block, the party checks for each transaction contained in its buffer whether it is valid, according to $\mathsf{ValidTx}_{\mathsf{OG}}$, with respect to the current state of the chain. Note that to allow for full generality we leave $\mathsf{ValidTx}_{\mathsf{OG}}$ as a protocol/ledger parameter (the same for both); this will allow to use the same protocol and ledger for different definitions of transaction validity.

The transaction validity predicate $\mathsf{ValidTx}_{\mathsf{OG}}$ induces a natural transaction validity on blockchain-states. This is captured by the predicate $\mathsf{isvalidstate}(\vec{\mathtt{st}})$ that decides whether a state consists of valid transactions according to $\mathsf{ValidTx}_{\mathsf{OG}}$. The predicate simply checks that each transaction $\mathtt{tx}$ of any state-block $\mathtt{st}_i$ included in the state $\vec{\mathtt{st}} = \mathtt{st}_0||\dots||\mathtt{st}_\ell$ includes transactions that are valid with respect to the state $\mathtt{st}_0||\dots||\mathtt{st}_{i-1}||\mathtt{st}_i^{-\mathtt{tx}}$, where $\mathtt{st}_i^{-\mathtt{tx}}$ is the $i$-th state block $\mathtt{st}_i$ with $\mathtt{tx}$ removed.

*Remark 1 (Building a Cryptocurrency Ledger).* Consistently with the cryptographic literature on blockchains, we use the term *transaction* to refer to input values $\mathtt{tx}$ given to the ledger protocol (and the ledger functionality). It is important to recall that in order to achieve the standard ledger functionality of this work, where *weak* transaction liveness is enforced, transactions need not be signed (cf. [17, 3]). [20] Using composition, a protection to amplify the liveness of transactions can be applied as a next modular step, on top of our ledger functionality. We note in passing that such an amplification has been achieved assuming a signature scheme combined with an explicit encoding of transactions to contain the source and destination addresses of the involved parties that relate to their public keys and/or identities; an honest protocol participant would consequently only sign its transactions but no others, and signature verification would be part of the validity check $\mathsf{ValidTx}_{\mathsf{OG}}$. We refer to [3] for details on how to build a UC cryptocurrency ledger on top of a generic transaction ledger using the composability guarantees of the UC framework.

### 3.3.4 Chain Selection

The most novel component of our protocol is the way in which a party decides which chain to adopt given a set of alternatives it (repeatedly) receives over the network. The chain selection protocol is invoked once a party has collected all chains he can in the current round—denote the set of all these chains by $\mathcal{N} = \{\mathcal{C}_1, \dots, \mathcal{C}_M\}$—and is trying to decide whether to keep his current local chain $\mathcal{C}_{\mathsf{loc}}$, or adopt one of the newly received chains in $\mathcal{N}$. As we prove, the power of the new rule lies in the fact that it allows a desynchronized or even a newly joining party—whose $\mathcal{C}_{\mathsf{loc}}$ is empty—to eventually converge to a good chain. We refer to this process as *bootstrapping from genesis,* and denote the new chain selection algorithm as $\mathsf{maxvalid\text{-}bg}$.

The chain selection process proceeds in three steps: First the party $U_p$ uses the clock to make sure the time-relevant parameters, i.e., $\tau, \mathtt{ep}$, and $\mathtt{sl}$, are up-to-date, and updates its local state accordingly (see below). Second, $U_p$ filters all the received chains, one-by-one, to keep only the ones that satisfy a syntactic validity property. Informally, those are chains whose signatures are consistent with the genesis block, and their block-contents are consistent with the keys recorded in KES, the VRF, and the global random oracle. The filtering of any given chain $\mathcal{C}$ is done by an invocation of protocol $\mathsf{IsValidChain}$ described below. Finally, the party applies our new chain selection rule $\mathsf{maxvalid\text{-}bg}$ on the filtered list of chains to (possibly) update its local chain. The above three steps are detailed in the following.

**Step 1: Updating the local state and time variables.** Every time a party fetches new information from the network, it needs to refresh its local view, and in particular to update the current epoch counter $\mathtt{ep}$ using the current clock time, as well as its view of the state parameters: the current epoch stake distribution $\mathbb{S}_{\mathsf{ep}}$, the relative stake $\alpha_p^{\mathsf{ep}}$, and epoch randomness $\eta_{\mathsf{ep}}$, and the staking threshold $T_p^{\mathsf{ep}}$. This is achieved by the protocols $\mathsf{UpdateStakeDist}$ (see Figure 8) and (the very simple) $\mathsf{UpdateTime}$ in Figure 9. The algorithm used to update the stake parameters, in particular the threshold $T_p^{\mathsf{ep}}$ was discussed in Section 3.3.3.

[20] More technically speaking, whether transactions are signed or not is completely orthogonal to the security proof in this paper. The reason is that the main honest-stake-majority condition refers to the stake-distribution and hence is a property of the basic content of the blockchain (and the corruption state of the miners) and therefore under the control of the environment providing the contents via inputs to the protocol.

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

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

// Determine leader status
Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y_\rho, \pi_\rho)$.
Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$.
**if** $y < T_p^{\mathsf{ep}}$ **then**
    // Generate a new block
    Set $\mathtt{buffer}' \leftarrow \mathtt{buffer}$, $\vec{N} \leftarrow \mathtt{tx}_{U_p}^{\text{base-tx}}$, and $\mathtt{st} \leftarrow \mathsf{blockify}_{\mathsf{OG}}(\vec{N})$
    **repeat**
        Parse $\mathtt{buffer}'$ as sequence $(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
        **for** $i = 1$ to $n$ **do**
            **if** $\mathsf{ValidTx}_{\mathsf{OG}}(\mathtt{tx}_i, \vec{\mathtt{st}}||\mathtt{st}) = 1$ **then**
                $\vec{N} \leftarrow \vec{N}||\mathtt{tx}_i$
                Remove $\mathtt{tx}$ from $\mathtt{buffer}'$
                Set $\mathtt{st} \leftarrow \mathsf{blockify}_{\mathsf{OG}}(\vec{N})$
            **end if**
        **end for**
    **until** $\vec{N}$ does not increase anymore
    Set $crt = (U_p, y, \pi)$, $\rho = (y_\rho, \pi_\rho)$ and $h \leftarrow H(\mathtt{head}(\mathcal{C}_{\mathsf{loc}}))$.
    Send $(\mathsf{USign}, \mathrm{sid}, U_p, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$; denote the response from $\mathcal{F}_{\mathsf{KES}}$ by $(\mathsf{Signature}, \mathrm{sid}, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, \sigma)$.
    Set $B \leftarrow (h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$ and update $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,\|\, B$.
    // Multicast the extended chain and wait.
    Send $(\textsc{multicast}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and proceed from here upon next activation of this procedure.
**else**
    Evolve the KES signing key by sending $(\mathsf{USign}, \mathrm{sid}, U_p, 0, \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$ and set the anchor at end of procedure to resume on next maintenance activation.
**end if**

**Fig. 7.** The Ouroboros Genesis staking procedure.

**Protocol** $\mathsf{UpdateStakeDist}(k, U_p, R, f)$

1: Set $\mathbb{S}_{\mathsf{ep}}$ to be the stakeholder distribution at the end of epoch $\mathtt{ep} - 2$ in $\mathcal{C}_{\mathsf{loc}}$ in case $\mathtt{ep} \geq 2$ (and keep the initial stake distribution in case $\mathtt{ep} < 2$).
2: Set $\alpha_p^{\mathsf{ep}}$ to be the relative stake of $U_p$ in $\mathbb{S}_{\mathsf{ep}}$ and $T_p^{\mathsf{ep}} \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha_p^{\mathsf{ep}})$.
3: Set $\eta_{\mathsf{ep}} \leftarrow H(\eta_{\mathsf{ep}-1} \,\|\, \mathtt{ep} \,\|\, v)$ where $v$ is the concatenation of the VRF outputs $y_\rho$ from all blocks in $\mathcal{C}_{\mathsf{loc}}$ from the first $2R/3$ slots of epoch $\mathtt{ep} - 1$.

Output: The protocol outputs $\tau, \mathtt{ep}, \mathtt{sl}, \mathbb{S}_{\mathsf{ep}}, \alpha_p^{\mathsf{ep}}, T_p^{\mathsf{ep}}$, and $\eta_{\mathsf{ep}}$ to its caller (but not to $\mathcal{Z}$).

**Fig. 8.** The protocol for updating the local stake distribution parameters.

**Step 2: Filtering out invalid chains.** The protocol IsValidChain which filters out invalid chains is the same as the corresponding protocol from [14]. For completeness we include it in Appendix B (see Figure 15).

**Step 3: The new chain selection rule.** The chain selection rule maxvalid from [14] (which, to avoid confusion, we hereafter refer to as maxvalid-mc for "moving checkpoint", cf. Section 4) prefers longer chains, unless the new chain $\mathcal{C}_i$ forks more than $k$ blocks relative to the currently held chain $\mathcal{C}_{\mathsf{max}}$ (in which case the new chain would be discarded). This so-called *moving checkpointing* is crucial for the security proof in [14]; indeed, maxvalid-mc only guarantees satisfactory blockchain properties when coupled with a checkpointing functionality that provides newly joining, or re-joining, parties with a recent trusted chain. In particular, such checkpointing provides resilience against so-called "long-range attacks" (see [18] for a detailed discussion).

**Protocol** $\mathsf{UpdateTime}(U_p, R)$

Update $\tau$ to the current time stamp of the clock (by sending Send $(\textsc{clock-read}, \mathrm{sid}_C)$ to $\mathcal{G}_{\textsc{clock}}$ and obtaining its answer).
Set $\mathtt{ep} \leftarrow \lceil \tau/R \rceil$, and $\mathtt{sl} \leftarrow \tau$.

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

**Fig. 9.** The protocol for updating the time variables.

Our new chain selection rule, formally specified as algorithm $\mathsf{maxvalid\text{-}bg}(\cdot)$ (see Figure 10), surgically adapts $\mathsf{maxvalid\text{-}mc}$ by adding an additional condition (Condition B). When satisfied, the new condition can lead to a party adopting a new chain $\mathcal{C}_i$ even if this chain did fork more than $k$ blocks relative to the currently held chain $\mathcal{C}_{\mathsf{max}}$. Specifically, the new chain would be preferred if it grows more quickly in the $s$ slots following the slot associated with the last block common to both $\mathcal{C}_i$ and $\mathcal{C}_{\mathsf{max}}$ (here $s$ is a parameter of the rule that we discuss in full detail in the proof). Roughly, this "local chain growth"—appearing just after the chains diverge—serves as an indication of the amount of participation in that interval. The intuition behind this criterion is that in a time interval shortly after the two chains diverge, they still agree on the leadership attribution for the upcoming slots, and out of the eligible slot leaders, the (honest) majority has been mostly working on the chain that ended up stabilizing.

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

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

**Fig. 10.** The new chain selection rule.

Thus the new rule substitutes a "global" longest chain rule with a "local" longest chain rule that prefers chains that demonstrate more participation after forking from the currently held chain $\mathcal{C}_{\mathsf{max}}$. As proven in Section 4, this additional condition allows an honest party that joins the network at an arbitrary point in time to bootstrap based only on the genesis block (obtained from $\mathcal{F}_{\mathsf{INIT}}$) and the chains it observes by listening to the network for a sufficiently long period of time. In prior work, a newly spawned party had to be assumed to be bootstrapped by obtaining an honest chain from an external, and fully trusted, mechanism (or, alternatively, be given a list of trustworthy nodes from which to request an honest chain); our solution does not rely on any such assumption. We refer to this process/assumption as *checkpointing*; provably avoiding this process by means of an updated chain selection rule is one of the major contributions of our work.

The protocol executed by the parties to select a new chain, denoted as SelectChain, can be found in Figure 11.

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

// Step 1: Filter out invalid chains

1: Initialize $\mathcal{N}_{\text{valid}} \leftarrow \emptyset$

2: **for** $i = 1 \ldots M$ **do**

Invoke Protocol IsValidChain$(U_p, \mathrm{sid}, k, \mathcal{C}_i, f, R)$; if it returns `true` then update $\mathcal{N}_{\text{valid}} \leftarrow \mathcal{N}_{\text{valid}} \cup \mathcal{C}_i$

**end for**

// Step 2: Applying the chain selection rule.

3: Execute Algorithm maxvalid-bg$(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}_{\text{valid}}, k, s, f)$ and receive its output $\mathcal{C}_{\mathsf{max}}$.

4: Replace $\mathcal{C}_{\mathsf{loc}}$ by $\mathcal{C}_{\mathsf{max}}$

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

**Fig. 11.** The protocol for parties to adopt a (new) chain.

We conclude this section by referring to Figure 12 for the technical overview of the main ledger maintenance protocol LedgerMaintenance which makes use of the previously introduced sub-processes.

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

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

1: **if** `isInit` is false **then** invoke Initialization-Genesis$(U_p, \mathrm{sid}, R)$; if Initialization-Genesis halts then halt (this will abort the execution).

**end if**

2: // From here the variables $v_p^{\text{vrf}}, v_p^{\text{kes}}, \tau, \mathtt{ep}, \mathtt{sl}, \mathcal{C}_{\mathsf{loc}}, T_p^{\mathsf{ep}}$, `isInit`, $t_{\text{on}}$,$t_{\text{work}}$ can be used to read from as they are guaranteed to be initialized.

3: Invoke FetchInformation$(U_p, \mathrm{sid})$ to receive the newest messages for this round; denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$, and record the flag WELCOME if not yet recorded for this round.

4: Set $\mathtt{buffer} \leftarrow \mathtt{buffer} || (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and define $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$

5: Invoke UpdateTime$(U_p)$ to update $\tau, \mathtt{ep}$ and $\mathtt{sl}$.

6: Set $t_{\text{on}} \leftarrow \tau$

7: Invoke SelectChain$(U_p, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$ to replace $\mathcal{C}_{\mathsf{loc}}$.

8: **if** $t_{\text{work}} < \tau$ **then**

9: Invoke UpdateStakeDist$(k, U_p, R, f)$ to update $\mathbb{S}_{\mathsf{ep}}, \alpha_p^{\mathsf{ep}}, T_p^{\mathsf{ep}}$, and $\eta_{\mathsf{ep}}$.

10: Call StakingProcedure$(U_p, \mathrm{sid}, k\mathtt{ep}, \mathtt{sl}, \mathtt{buffer}, \mathcal{C}_{\mathsf{loc}})$ // Eexecution stays $(\textsc{maintain-ledger}, \mathrm{sid}, \mathrm{minerID})$-interruptible.

11: Set $t_{\text{work}} \leftarrow \tau$ // React on a joining party

12: If $\textsc{welcome} = 1$ in this round, then send $(\textsc{multicast}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and $(\textsc{multicast}, \mathrm{sid}, \mathtt{buffer})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$. Otherwise, give up activation and resume below upon next maintenance activation.

**end if**

13: Invoke FinishRound$(U_p)$

**Fig. 12.** The main ledger maintenance protocol.

### 3.3.5 Reading the State

The last command related to the interaction with the ledger is the read command $(\textsc{read}, \mathrm{sid})$ that is used to read the current contents of the state. Note that in the ideal world, the result of issuing such a command is for the ledger to output a (long enough prefix) of the current state of the ledger. Analogously, in the real world, the result is for the party receiving it to execute protocol ReadState which works as follows: the party, first, gets up to speed with time, and updates its local blockchain using the blockchains that have been sent to it,[21] and then it computes and outputs the prefix of its local chain (chopping of $k$ blocks.) The protocol ReadState is detailed in Figure 13.

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

1: If `isInit` is false output the empty state $(\textsc{read}, \mathrm{sid}, \varepsilon)$ (to $\mathcal{Z}$). Otherwise, do the following:
2: Invoke FetchInformation$(U_p, \mathrm{sid})$ to receive the newest messages for this round; denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$, and read the flag WELCOME.
3: Set $\mathtt{buffer} \leftarrow \mathtt{buffer}||(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$; $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$; Record if $\textsc{welcome} = 1$ (for later usage).
4: Invoke UpdateTime$(U_p)$ to update $\tau, \mathtt{ep}$ and $\mathtt{sl}$.
5: Set $t_{\mathrm{on}} \leftarrow \tau$
6: Invoke Protocol SelectChain$(U_p, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$.
7: Extract the state $\vec{\mathtt{st}}$ from the current local chain $\mathcal{C}_{\mathsf{loc}}$.
8: Output $(\textsc{read}, \mathrm{sid}, \vec{\mathtt{st}}^{\lceil k})$ (to $\mathcal{Z}$). // $\vec{\mathtt{st}}^{\lceil k}$ denotes the prefix of $\vec{\mathtt{st}}$ with the last $k$ state blocks chopped off

**Fig. 13.** The protocol for parties to adopt a (new) chain.

# 4 Security Analysis

## 4.1 Blockchain Security Properties

We first define the standard security properties of blockchain protocols: *common prefix*, *chain growth* and *chain quality*. While the security guarantees we prove in this paper are formulated in the UC setting, these standalone properties will turn out to be useful tools for our analysis.

**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 $\mathtt{sl}_1 \leq \mathtt{sl}_2$ 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 $\tau \in (0, 1]$ and $s \in \mathbb{N}$.** 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 $\mathtt{sl}_1$ is at least $s$ slots prior to $\mathtt{sl}_2$. Then $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \geq \tau \cdot s$. We call $\tau$ 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 in this portion is at least $\mu$. We call $\mu$ the chain quality coefficient.

Note that previous work identified and studied a stronger version of chain growth (denoted below as CG2), which controls the relative growth of chains held by potentially distinct honest parties.

**(Strong) Chain Growth (CG2); with parameters $\tau \in (0, 1]$ and $s \in \mathbb{N}$.** Consider the chains $\mathcal{C}_1, \mathcal{C}_2$ possessed by two alert parties at the onset of two slots $\mathtt{sl}_1, \mathtt{sl}_2$ with $\mathtt{sl}_1$ at least $s$ slots prior to $\mathtt{sl}_2$. Then it holds that $\mathsf{len}(\mathcal{C}_2) - \mathsf{len}(\mathcal{C}_1) \geq \tau \cdot s$. We call $\tau$ the speed coefficient.

We remark that the notion of chain growth CG2 follows from CP and CG (with some appropriate decay in parameters). However, it appears that CG is a preferable formulation in our setting, as it can be established with stronger parameters than CG2 and more naturally dovetails with several aspects of the security proofs.

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

[21] Observe that a stalled party that returns to the alert status will fetch all messages sent to it while it was stalled.

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

As a side remark, the CG (resp. CQ) property follows from $\exists$CQ and an additional property called *honest-bounded chain growth* HCG (resp. *honest-bounded chain quality*, HCQ). We define HCG and HCQ and establish these relationships in Appendix E.5.

Note that typically these security properties for blockchain protocols are formulated so that they grant the above-described guarantees to all *honest* parties. However, in our more fine-grained modeling of parties' availability, a natural choice is to analyze these properties for the *alert* parties only. In particular, the properties CQ and $\exists$CQ guarantee sufficient presence of blocks generated by alert parties in the chain, i.e., generated by parties that were fully up-to-date with the state of the protocol and capable of contributing to it. In the following treatment, we often talk about *honestly-generated blocks*, but this always refers to blocks generated by such alert parties. Similarly, *adversarial blocks* don't necessarily have to come from corrupted parties, rather from any parties that are not fully alert.

### 4.2 Security of Ouroboros Genesis with maxvalid-mc

The original Ouroboros Praos protocol given in [14] differs from Ouroboros Genesis in a single point: it employs a different chain selection rule, which we call maxvalid-mc here and outline below. The difference in maxvalid-mc compared to maxvalid-bg is that if the considered chain $\mathcal{C}_i$ forks from the current chain $\mathcal{C}_{\mathsf{loc}}$ more than $k$ blocks in the past, it is immediately discarded, without evaluating Condition B as in maxvalid-bg. This can be seen as a "moving checkpoint" $k$ blocks behind the current tip of the chain, which is what the suffix "-mc" stands for. To preserve clarity, we will use Ouroboros-Praos to refer to the protocol that is identical to the one given in Section 3 except that is uses maxvalid-mc instead of maxvalid-bg as its chain-selection rule.

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

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

Our first goal is to establish that the useful properties of common prefix, chain growth, and chain quality are achieved by Ouroboros-Praos, when executed in a slightly restricted environment. Namely, we start by assuming that all parties participate in the protocol run from the beginning and never get deregistered from the network $\mathcal{F}_{\text{N-MC}}$ (i.e., honest parties are always *online*); we refer to this setting as the *setting with static $\mathcal{F}_{\text{N-MC}}$-registration*. We will drop this assumption later.

The desired statement for this limited environment is given in Theorem 1, the rest of Section 4.2 will be dedicated to sketching its proof, which is fully spelled out in Appendix E. First, we need to define some relevant quantities.

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

Looking ahead, we remark that even though we give the general definition above, our protocol will have the property that for all party types and all time slots, $\mathcal{S}^-(\mathcal{P}_{\mathsf{type}}[t]) = \mathcal{S}^+(\mathcal{P}_{\mathsf{type}}[t])$ 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).** *At any time slot $t$ during the execution, we let:*

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

It is instructive to see that the potentially active stake ratio allows to conclude the ratio of stake belonging to parties that cannot participate in slot $t$.

Note that in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration, all honest parties are online from the beginning, and therefore also synchronized. The set of active parties hence consists only of alert, adversarial and time-unaware parties. In the general case it also contains honest parties that are online but desynchronized, we will discuss these in detail in Section 4.4.

**Theorem 1.** *Consider the execution of Ouroboros-Praos with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration. Let $f$ be the active-slot coefficient, let $\Delta$ be the upper bound on the network delay and let $Q$ be an upper bound on the total number of queries issued to $\mathcal{G}_{\text{RO}}$. Let $\alpha, \beta \in [0,1]$ denote a lower bound on the alert ratio and participating ratio throughout the whole execution, respectively. Let $R$ and $L$ denote the epoch length and the total lifetime of the system (in slots). If for some $\epsilon \in (0,1)$ we have*

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

*and $R \geq 144\Delta/\epsilon\beta f$ then Ouroboros-Praos achieves the following guarantees:*

- ***Common prefix.*** *The probability that Ouroboros-Praos violates the common prefix property with parameter $k$ is no more than*
$$\epsilon_{\mathsf{CP}}(k) \triangleq \frac{19L}{\epsilon^4}\exp(\Delta - \epsilon^4 k/18) + \epsilon_{\mathsf{lift}}\ ;$$
- ***Chain growth.*** *The probability that Ouroboros-Praos violates the chain growth property with parameters $s \geq 48\Delta/(\epsilon\beta f)$ and $\tau_{\mathsf{CG}} = \beta f/16$ is no more than*
$$\epsilon_{\mathsf{CG}}(\tau_{\mathsf{CG}}, s) \triangleq \frac{sL^2}{2}\exp\left(-(\epsilon\beta f)^2 s/256\right) + \epsilon_{\mathsf{lift}}\ ;$$
- ***Existential chain quality.*** *The probability that Ouroboros-Praos violates the existential chain quality property with parameter $s \geq 12\Delta/(\epsilon\beta f)$ is no more than*
$$\epsilon_{\exists\mathsf{CQ}}(s) \triangleq (s+1)L^2\exp\left(-(\epsilon\beta f)^2 s/64\right) + \epsilon_{\mathsf{lift}}\ ;$$
- ***Chain quality.*** *The probability that Ouroboros-Praos violates the chain quality property with parameters $k \geq 48\Delta/(\epsilon\beta f)$ and $\mu = \epsilon\beta f/16$ is no more than*
$$\epsilon_{\mathsf{CQ}}(\mu, k) \triangleq \frac{kL^2}{2}\exp\left(-(\epsilon\beta f)^2 k/256\right) + \epsilon_{\mathsf{lift}}\ ;$$

*where $\epsilon_{\mathsf{lift}}$ is a shorthand for the quantity*

$$\epsilon_{\mathsf{lift}} \triangleq QL \cdot \left[R^3 \cdot \exp\left(-\frac{(\epsilon\beta f)^2 R}{768}\right) + \frac{38R}{\epsilon^4}\cdot\exp\left(\Delta - \frac{\epsilon^4\beta f R}{864}\right)\right]\ .$$

*Proof (sketch).* The proof is inspired by the proof of property-based security of Ouroboros Praos given in [14]; however, a major extension of the techniques is necessary. To appreciate the need for this extension, let us first recall in very broad terms how the proof in [14] proceeds:

1. First, the above security properties (or slight variations of them, cf. Section 4.1) are proven for a single epoch. For this, the dynamics of the protocol execution is abstracted into combinatorial objects called *forks*, while the slot leader selection (assuming static corruption) is captured by sampling a so-called *characteristic string*.
2. A recursive rule is given that identifies whether a characteristic string allows for "dangerous" forks, and a probabilistic analysis shows that under static corruption, leader schedules corresponding to such characteristic strings are extremely rare.
3. Given the rarity of such undesirable characteristic strings, the $\mathsf{CP}$, $\mathsf{CG}$, and $\mathsf{CQ}$ properties are established for a single epoch and a static-corruption adversary.
4. The analysis is generalized to fully adaptive corruption by showing a static-corruption adversary that dominates any adaptive one.
5. The analysis is extended to an arbitrary number of epochs by analyzing the subprotocol for generating new randomness to be used in the following epoch to sample the leader schedule.

The main improvement of Theorem 1 over the analysis in [14] is that it captures stalled and time-unaware parties (and making honest parties stalled or time-unaware is a fully adaptive decision of the environment). This is done in two different ways: broadly speaking, while stalled parties don't pose a threat as long as the majority of active parties is honest, time-unaware parties are problematic (as they cannot evolve their keys) and have to be counted as adversarial until they regain their time-awareness.

Unfortunately, the adaptive nature of the above environment's control makes it impossible to start with a static analysis of the slot-leader selection as done above in steps 1–3. Moreover, the argument in step 4 completely breaks down as the static adversary given in [14] no longer dominates any possible adaptive combination of corruption and stalling. Therefore, our proof needs to revisit the steps 1–4 and replace the analysis of a sequence of binomially distributed random variables (representing the characteristic string) by considering inter-slot dependence right from the beginning. This is done via a martingale framework that is an important contribution of this paper and might prove useful also outside of the analysis of the Ouroboros protocols. We give all the details of our approach in Appendix E, where we also describe the parts of the framework from [14] that are necessary to follow our proof. □

### 4.3 Adopting the New maxvalid-bg Rule

**Theorem 2.** *Consider the protocol Ouroboros-Genesis using maxvalid-bg as described in Section 3, executed in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration, under the same assumptions as in Theorem 1. If the maxvalid-bg parameters, $k$ and $s$, satisfy*

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

*then the guarantees given in Theorem 1 for common prefix, chain growth, chain quality, and existential chain quality are still valid except for an additional error probability*

$$\exp\left(\ln L - \Omega(k)\right) + \epsilon_{\mathsf{CG}}(\beta f/16, k/(4f)) + \epsilon_{\exists\mathsf{CQ}}(k/(4f)) + \epsilon_{\mathsf{CP}}(k\beta/64)\,. \tag{5}$$

*Proof.* We show that when replacing maxvalid-mc with maxvalid-bg, the overall execution of the protocol remains the same except with negligible probability. To see this, consider a run of the protocol with maxvalid-mc, and let $\mathtt{sl}_b$ denote the first slot when any honest party discards a received candidate chain $\mathcal{C}_{\mathsf{cand}}$ (longer than $\mathcal{C}_{\mathsf{loc}}$) because it forks from its $\mathcal{C}_{\mathsf{loc}}$ by more than $k$ blocks, as described by maxvalid-mc. Until $\mathtt{sl}_b$, the whole execution would proceed identically if parties were using maxvalid-bg instead, as in both cases they would always prefer the longer of the compared chains using Condition A.

Consider now the decision that a party running maxvalid-bg would make regarding this chain $\mathcal{C}_{\mathsf{cand}}$ in the slot $\mathtt{sl}_b$. We will argue that it will also favor $\mathcal{C}_{\mathsf{loc}}$ with overwhelming probability. This will then imply the full statement, as the reasoning can be applied inductively to each of the slots where $\mathsf{maxvalid\text{-}mc}$ discards a longer chain, throughout the whole execution.

Let $\mathtt{sl}_a$ be the slot associated with the last common block of $\mathcal{C}_{\mathsf{loc}}$ and $\mathcal{C}_{\mathsf{cand}}$. Recall that by the design of the protocol (independently of the underlying $\mathsf{maxvalid}$ rule), for every slot $\mathtt{sl}_i$ there is an event $E_i$ such that: (i.) $\Pr[E_i] = 1 - f$; (ii.) the events $E_1, E_2, \ldots$ are independent; (iii.) if $E_i$ occurs, then no valid block can be created for the slot $\mathtt{sl}_i$. Therefore, using a Chernoff bound (cf. Appendix F) and a union bound over the running time $L$ of the system, we can also lower-bound the number of slots between $\mathtt{sl}_a$ and $\mathtt{sl}_b$ as $a - b \geq k/(2f)$, except with error probability $\exp(\ln L - \Omega(k))$. For the remainder of the proof, we will assume that the execution satisfies this property (that is, $\mathtt{sl}_b - \mathtt{sl}_a > k/(2f)$ for all pairs of slots bounding $k$ blocks on an honestly held chain) and, further, that:

(CP) there is no $k\beta/64$-CP violation;
($\exists$CQ) there is no $s$-$\exists$CQ violation; and
(CG) there is no $(\beta f/16, s)$-CG violation.

As indicated in the statement of the theorem, $s$ is fixed to be $k/(4f)$. Observe that the error probabilities associated with these events are then precisely those appearing in (5).

By the definition of $\mathsf{maxvalid\text{-}bg}$, the chain $\mathcal{C}_{\mathsf{cand}}$ can only be adopted in favor of $\mathcal{C}_{\mathsf{loc}}$ if

$$\left|\mathcal{C}_{\mathsf{cand}}[0 : \mathtt{sl}_a + s]\right| > \left|\mathcal{C}_{\mathsf{loc}}[0 : \mathtt{sl}_a + s]\right| . \tag{6}$$

We will show that under the assumptions described above, this is not possible. For convenience, we consider two disjoint, consecutive subintervals of $(\mathtt{sl}_a, \mathtt{sl}_b]$:

$$I_{\text{growth}} = (\mathtt{sl}_a, \mathtt{sl}_a + s] \qquad \text{and} \qquad I_{\text{stabilize}} = (\mathtt{sl}_a + s, \mathtt{sl}_a + 2s] . \tag{7}$$

Note that by the choice of $s$, both $I_{\text{growth}}$ and $I_{\text{stabilize}}$ are indeed subintervals of $(\mathtt{sl}_a, \mathtt{sl}_b]$. Moreover, since $2s \leq R/3$, the chains $\mathcal{C}_{\mathsf{loc}}$ and $\mathcal{C}_{\mathsf{cand}}$ use the same stake distribution and randomness to determine slot leaders for the interval $I_{\text{growth}} \cup I_{\text{stabilize}}$.

First, we observe that $\mathcal{C}_{\mathsf{loc}}$ exhibits significant growth over the interval $I_{\text{growth}}$: specifically, by the chain growth property established in Theorem 1 and the assumption $s = k/(4f) \geq 48\Delta/(\epsilon\beta f)$, we have

$$\left|\mathcal{C}_{\mathsf{loc}}[I_{\text{growth}}]\right| \geq s\beta f/16 = k\beta/64 .$$

Similarly, observe that $\mathcal{C}_{\mathsf{loc}}$ possesses at least one honestly-generated block over the interval $I_{\text{stabilize}}$: specifically, by the existential chain quality property established in Theorem 1 and the assumption $s = k/(4f) \geq 24\Delta/(\epsilon\beta f)$, there must exists a slot $\mathtt{sl}^* \in I_{\text{stabilize}}$ for which $\mathcal{C}_{\mathsf{loc}}[\mathtt{sl}^*]$ was honestly generated.

To complete the argument, we observe that the assertion (6) would yield a violation of common prefix. To argue this, we take advantage of the notions of characteristic strings, forks, (viable) tines and divergence, defined in Appendix E.1.

Specifically, consider the characteristic string $W$ and the fork $F \vdash_\Delta W$ associated with this execution of the protocol. (The fork $F$ reflects all valid chains adopted by honest players during the execution—the local chains that result from application of the $\mathsf{maxvalid}$ rule.) Let $t_{\text{loc}}$ denote the tine associated with the chain $\mathcal{C}_{\mathsf{loc}}[0 : \mathtt{sl}^* - 1]$ and $t_{\text{cand}}$ denote the tine associated with the chain $\mathcal{C}_{\mathsf{cand}}[0 : \mathtt{sl}_a + s]$. The tine $t_{\text{loc}}$ is viable, as the honest leader associated with $\mathtt{sl}^*$ chose $\mathcal{C}_{\mathsf{loc}}$ to extend. To construct a viable tine from $t_{\text{cand}}$, we extend it using the adversarial slots associated with the portion of $t_{\text{loc}}$ in $I_{\text{stabilize}}$. Specifically, recalling that $\mathtt{sl}^*$ is associated with the first honestly generated block of $\mathcal{C}_{\mathsf{loc}}$ in $I_{\text{stabilize}}$, any blocks of $\mathcal{C}_{\mathsf{loc}}$ associated with slots in the interval $(\mathtt{sl}_a + s, \mathtt{sl}^*)$ are associated with adversarial slots of $W$, and we may use these adversarial slots to extend $t_{\text{cand}}$: Let $\widehat{t}_{\text{cand}}$ denote the extension of the tine $t_{\text{cand}}$ formed by adding an adversarial node for each slot in $(\mathtt{sl}_a + s, \mathtt{sl}^*)$ associated with a block of $t_{\text{loc}}$. Note, also, that $\widehat{t}_{\text{cand}}$ is viable, as $\mathtt{length}(\widehat{t}_{\text{cand}}) > \mathtt{length}(t_{\text{loc}})$. (Note that $|\mathcal{C}_{\mathsf{loc}}[0 : \mathtt{sl}_a]| < |\mathcal{C}_{\mathsf{cand}}[0 : \mathtt{sl}_a]|$ by assumption, and the tines $t_{\text{loc}}$ and $t_{\text{cand}}$ have the same number of blocks in the region $(\mathtt{sl}_a + s, \mathtt{sl}^*)$.) Thus these two tines form a divergence-violation (that is, a CP-violation) with parameter $\left|\mathcal{C}_{\mathsf{loc}}[\mathtt{sl}_a + 1, \mathtt{sl}_a + s]\right| \geq s\beta f/16 = k\beta/64$ (by the chain growth guarantee above). $\square$

### 4.4 Newly Joining Parties

In this section we prove that the guarantees on common prefix, chain growth and (existential) chain quality obtained for Ouroboros-Genesis in Section 4.3 remain valid also when new parties join the protocol later during its execution.

To capture this, we proceed as follows. For any new party $U$ that joins the protocol later during its execution (say at slot $\mathtt{sl}_{\mathsf{join}}$), we consider a "virtual" party $\tilde{U}$ that holds no stake, but was participating in the protocol since the beginning and was alert all the time. Moreover, we assume that starting from $\mathtt{sl}_{\mathsf{join}}$, $\tilde{U}$ is receiving the same messages (in the same slots) as $U$. Clearly, the run of the protocol up to $\mathtt{sl}_{\mathsf{join}}$ would look the same with and without $\tilde{U}$, as $\tilde{U}$ would never be elected a slot leader, and would not affect $\alpha$ or $\beta$. Therefore, the execution of the protocol up to the point when the first party $U$ tries to join is covered by the statements proven in Section 4.3 (even when also considering the participation of $\tilde{U}$).

**Definition 3 (Adopting and discarding chains).** *We say that an honest party* adopts *a chain $\mathcal{C}$ when an execution of the procedure maxvalid-bg by this party returns $\mathcal{C}$. An honest party* discards *a chain $\mathcal{C}$ when an execution of the procedure maxvalid-bg by this party takes $\mathcal{C}$ as one of its inputs, but does not output $\mathcal{C}$.*

**Definition 4 (Virtual executions and virtual parties).** *We say that an honest party $U$ is* joining the protocol execution *at slot $\mathtt{sl}_{\mathsf{join}}$ if $\mathtt{sl}_{\mathsf{join}}$ is the slot in which $U$ becomes operational, time-aware and online for the first time. For a party $U$ joining the execution $\mathcal{E}$ of the protocol Ouroboros-Genesis at slot $\mathtt{sl}_{\mathsf{join}}$, consider an execution $\mathcal{E}'$ that only differs from $\mathcal{E}$ by one additional party $\tilde{U}$ being present from the beginning, registering 0 stake, remaining alert throughout the execution, and receiving the same messages as U from $\mathtt{sl}_{\mathsf{join}}$ on. We call $\mathcal{E}'$ (resp. $\tilde{U}$) the* virtual execution *(resp. the* virtual party*) for $U$.*

**Definition 5 (Synchronizing chains).** *We call (a message containing) a chain $\mathcal{C}_{\mathsf{sync}}$* synchronizing for *$U$, if this is the first chain that its virtual party $\tilde{U}$ adopts after slot $\mathtt{sl}_{\mathsf{join}}$.*

**Definition 6 (Synchronized parties).** *A party $U$ is called* synchronized *at time $t$ (cf. Fig. 1) with respect to synchronization parameter $t_{\mathsf{sync}} \geq 0$, if either $U$ has been online and time-aware since the beginning; or the last time it registered to the network $\mathcal{F}_{\text{N-MC}}$ was at time $t_0$ and there is a slot $t_1 \in \{t_0, \ldots, t - t_{\mathsf{sync}}\}$ such that the following conditions are satisfied:*

*(i) $U$ has been online and time-aware since $t_1$;*
*(ii) $U$ was operational in slots $t_1, \ldots, t_1 + t_{\mathsf{sync}} - 1$.*

*Otherwise, the party is called* desynchronized.

The above definition leaves flexibility when a party is considered synchronized, or more importantly, for how long after joining it is considered de-synchronized. Note that during the time a party is de-synchronized, the bad impact on the protocol run is that it could potentially extend chains that are purely adversarial. Thus, a lower value of $t_{\mathsf{sync}}$ is generally preferable to get a stronger statement, but could come at the cost of higher network traffic. Before we instantiate the parameter $t_{\mathsf{sync}}$ of Definition 6 for the specific case of Ouroboros Genesis, which is done in Lemma 2, we first need to examine the joining process in general.

In the following, whenever we refer to the set of synchronized/desynchronized parties, we implicitly refer to a generic synchronization parameter $t_{\mathsf{sync}}$.

The heart of our argument for newly joining parties is captured in the following lemma.

**Lemma 1.** *In the same setting as Theorem 2 but with dynamic $\mathcal{F}_{\text{N-MC}}$-registrations, any newly joining party that remains operational, time-aware and online until it receives its synchronizing chain, will adopt it except with probability (5).*

*Proof.* We assume that none of the bad events considered in the proof of Theorem 2 occurs. Let $U$ be a new party joining the protocol at slot $\mathtt{sl}_{\mathsf{join}}$. Moreover, let $U$ be the first such party in this execution, the argument can then inductively be applied to other parties joining later.

Consider the virtual execution $\mathcal{E}'$ for $U$, let $\tilde{U}$ be its corresponding virtual party, let $\mathcal{C}_{\mathsf{sync}}$ be its synchronizing chain, and let $\mathtt{sl}_{\mathsf{sync}}$ be the slot in which $U$ and $\tilde{U}$ receive $\mathcal{C}_{\mathsf{sync}}$. For the sake of contradiction, assume that $U$ does not adopt $\mathcal{C}_{\mathsf{sync}}$, and let $\mathcal{C}_1$ denote the chain that $U$ is holding as its local chain $\mathcal{C}_{\mathsf{loc}}$ when running maxvalid-bg in slot $\mathtt{sl}_{\mathsf{sync}}$. Additionally, let $\mathtt{sl}_{j_1}$ denote the slot that contains the last common block of $\mathcal{C}_1$ and $\mathcal{C}_{\mathsf{sync}}$. Finally, let $\mathcal{C}_2$ denote the chain that $\tilde{U}$ is holding as its local chain $\mathcal{C}_{\mathsf{loc}}$ when running maxvalid-bg in slot $\mathtt{sl}_{\mathsf{sync}}$. As $\mathcal{C}_{\mathsf{sync}}$ is the first chain $\tilde{U}$ adopts after $\mathtt{sl}_{\mathsf{join}}$, we know that $\mathcal{C}_2$ was adopted by $\tilde{U}$ before $\mathtt{sl}_{\mathsf{join}}$. Let $\mathtt{sl}_{j_2}$ denote the slot that contains the last common block of $\mathcal{C}_2$ and $\mathcal{C}_{\mathsf{sync}}$.

We have to analyze two possible cases here, depending on which condition in the procedure maxvalid-bg was used by $U$ to discard $\mathcal{C}_{\mathsf{sync}}$.

- $U$ **discards $\mathcal{C}_{\mathsf{sync}}$ using Condition A.** Since Condition A was invoked, this means that $\#_{j_1:\mathsf{sync}}(\mathcal{C}_1) \leq k$, and since $\mathcal{C}_{\mathsf{sync}}$ was discarded, we have $|\mathcal{C}_1| \geq |\mathcal{C}_{\mathsf{sync}}|$.
  However, since $\tilde{U}$ adopted $\mathcal{C}_{\mathsf{sync}}$, we argue that $|\mathcal{C}_{\mathsf{sync}}| > |\mathcal{C}_2|$. This holds because $\tilde{U}$ always adopts a new chain using Condition A, which is a direct consequence of the arguments given in the proof of Theorem 2. Hence, we can derive $|\mathcal{C}_1| > |\mathcal{C}_2|$. To obtain a contradiction with the fact that $\tilde{U}$ did not adopt $\mathcal{C}_1$ to replace $\mathcal{C}_2$, we only need to show that when it received $\mathcal{C}_1$ it used Condition A to make its adoption decision, i.e., that $\mathcal{C}_1$ does not fork more than $k$ blocks back from $\mathcal{C}_2$.
  This can be shown by case analysis. We need to consider two subcases:
  **Case $j_1 \leq j_2$:** This means that $\mathcal{C}_2$ forks from $\mathcal{C}_{\mathsf{sync}}$ not earlier than $\mathcal{C}_1$ does and hence $\mathcal{C}_1$ forks from $\mathcal{C}_2$ in slot $\mathtt{sl}_{j_1}$. Since we know that $\#_{j_1:\mathsf{sync}}(\mathcal{C}_1) \leq k$ and $|\mathcal{C}_1| \geq |\mathcal{C}_{\mathsf{sync}}| > |\mathcal{C}_2|$, we can easily conclude $\#_{j_1:\mathsf{sync}}(\mathcal{C}_2) \leq k$ in this case.
  **Case $j_1 > j_2$:** Here $\mathcal{C}_2$ forks from $\mathcal{C}_{\mathsf{sync}}$ earlier than $\mathcal{C}_1$, and hence $\mathcal{C}_1$ forks from $\mathcal{C}_2$ in slot $\mathtt{sl}_{j_2}$. The desired inequality $\#_{j_2:\mathsf{sync}}(\mathcal{C}_2) \leq k$ in this case follows from the common prefix property.
- $U$ **discards $\mathcal{C}_{\mathsf{sync}}$ using Condition B.** The contradiction in this case is obtained by using exactly the same argument as in the proof of Theorem 2 to show that if $U$ invokes Condition B on $\mathcal{C}_{\mathsf{sync}}$, it must actually adopt it.
  Namely, observe that we have $\#_{j_1:\mathsf{sync}}(\mathcal{C}_1) > k$ and hence with overwhelming probability $\mathsf{sync} - j_1 > k/(2f)$. For intervals $I_{\text{growth}}, I_{\text{stabilize}}$ defined as in (7) for $a := j_1$ and $b := \mathsf{sync}$, on $\mathcal{C}_{\mathsf{sync}}$ we again have a guarantee of sufficient chain growth in $I_{\text{growth}}$ and at least one honest block in $I_{\text{stabilize}}$. Hence, by the same argument, Condition B in maxvalid-bg will favor $\mathcal{C}_{\mathsf{sync}}$, otherwise a violation of common prefix would occur. □

Based on Lemma 1, what remains is to upper-bound the time interval that a newly joining party has to be considered desynchronized, i.e., the time it takes until it obtains its synchronizing chain (which it will adopt). We present some alternatives beyond the default mechanism.

**Lemma 2.** *Consider the same setting as Lemma 1 and let $\Delta$ be the network delay. Consider an honest party $U_p$ in slot $\mathtt{sl}$, which newly joined the protocol execution (and hence being registered to the network) at some slot $\mathtt{sl}_{\mathsf{join}} \leq \mathtt{sl}$. If party $U_p$ is considered synchronized in slot $\mathtt{sl}$ according to Definition 6 with parameter $t_{\mathsf{sync}} \geq 2\Delta$, then it has also received its synchronizing chain.*

*Furthermore, if alert parties multicast their local state every (constant) $T$ rounds, we obtain the statement for $t_{\mathsf{sync}} \geq T + \Delta$ even without any active request by the newly joining party.*

*Proof.* Both cases follow from observing when the alert party $\tilde{U}$ would receive a synchronizing chain in the respective case. Clearly, for the first (and default) case this is no more than the round-trip time $2\Delta$ after the actual new party joins the network, as any other alert party multicasts its local state by $\mathtt{sl}_{\mathsf{join}} + \Delta$ (and in case of any later state update, it will multicast such a newer state by definition of the protocol). The second part follows similarly by observing that the above argument still holds, but where other alert parties multicast their local state by $\mathtt{sl}_{\mathsf{join}} + T$. □

Hence, a party is considered synchronized in an execution of the protocol Ouroboros Genesis, if it satisfies Definition 6 with respect to parameter $t_{\mathsf{sync}} = 2\Delta$.

*Remark 2 (Self-synchronization).* Note that the protocol Ouroboros-Genesis is self-synchronizing in the sense that even without any active request, the newly joining party will receive its synchronizing chain by slot $\mathtt{sl}_{\mathsf{join}} + t_{\mathsf{sync}}$ except with error probability $\epsilon_{\mathsf{CG2}}(t_{\mathsf{sync}})$ of the event that $\tilde{U}$ does not adopt a new chain during a period of $t_{\mathsf{sync}}$, which directly contradicts the CG2 security property for the respective parameters. A bound on CG2-violation (and hence also $\epsilon_{\mathsf{CG2}}(t_{\mathsf{sync}})$) could be established as described in Section 4.1, however it would lead to longer synchronization times. We therefore do not pursue this option further, and instead choose to consider the default synchronization process as presented in Section 3.

The analysis of the synchronization process that was outlined above applies also to resynchronization of parties that have already participated in the protocol, acquired some stake, and then got deregistered from $\mathcal{F}_{\text{N-MC}}$ and hence became offline. The only difference is that, since the joining party does not know which of the messages it receives is actually its synchronizing message containing $\mathcal{C}_{\mathsf{sync}}$, it starts participating in the protocol immediately after rejoining. Hence, before it receives $\mathcal{C}_{\mathsf{sync}}$ its participation is to some extent controlled by the adversary and hence its stake has to be counted towards the adversarial stake even though the party is not formally corrupted. This is already captured in the general form of Definition 2, and hence we have established the following corollary.

**Corollary 1.** *Consider the protocol Ouroboros-Genesis as described in Section 3, executed in an environment with dynamic $\mathcal{F}_{\text{N-MC}}$-registrations and deregistrations. Then, under the assumptions of Theorem 2, the guarantees it gives for common prefix, chain growth, and chain quality are valid also in this general setting.*

### 4.5 Composable Guarantees

In this section, we conclude the analysis by showing how the property-focused statement of Corollary 1 can be turned into a universally composable security statement. This concludes the UC-analysis of Ouroboros-Genesis. The statement is conditioned again on the honest majority assumption introduced above. As explained in [3] for fully composable statements, it is desirable not to restrict the environment, but rather model these restrictions as part of the setup. In [3], they put forth a general methodology to model such restrictions as wrapper functionalities that control the interaction between an adversary and the assumed setup functionality to enforce the restrictions. For completeness, we provide the corresponding wrapper in Section A.

To prove composable security, the properties proven above for the real-world UC-execution play a crucial role in realizing the ledger $\mathcal{G}_{\text{LEDGER}}$ functionality (implementing a certain policy): first, the common-prefix property ensures that the ledger can maintain a unique ledger-state (a chain of state-blocks). Second, the chain quality ensures that the ledger can enforce a fraction of honestly generated blocks. Third, chain growth ensures that the ledger functionality can enforce its state to grow. The remaining arguments are given in the proof below. We now state the composable version of Corollary 1 (again for the default $t_{\mathsf{sync}} = 2\Delta$ case) as a theorem:

**Theorem 3.** *Let $k$ be the common-prefix parameter and let $R$ be the epoch-length parameter (restricted as in Theorem 2), let $\Delta$ be the network delay, let $\tau_{\mathsf{CG}}$ and $\mu$ be the speed and chain-quality coefficients, respectively (both defined as in Theorem 1), and let $\alpha$ and $\beta$ refer to the respective bounds on the participation ratios (as in Theorem 1). Let $\mathcal{G}_{\text{LEDGER}}$ be the ledger functionality defined in Section 2.2 and instantiate its parameters by*

$$\mathtt{windowSize} = k \quad \text{and} \quad \mathtt{Delay} = 2\Delta$$
$$\mathtt{maxTime}_{\mathtt{window}} \geq \frac{\mathtt{windowSize}}{\tau_{\mathsf{CG}}} \quad \text{and} \quad \mathtt{advBlcks}_{\mathtt{window}} \geq (1-\mu)\mathtt{windowSize}.$$

*The protocol Ouroboros-Genesis (with access to its specified hybrids) securely UC-realizes $\mathcal{G}_{\text{LEDGER}}$ under the assumptions required by Theorem 1 (which are formally enforceable by a real-world wrapper functionality $\mathcal{W}^{PoS}_{\text{OG}}(\cdot)$ as given in Section D). In addition, the corresponding simulation is perfect except with negligible probability in the parameter $k$ when setting $R \geq \omega(\log k)$.*

*Proof.* We prove the theorem by proving the security with respect to the EUC version of UC (which stands for externalized UC) and we treat the clock and the random oracle as the only two shared functionalities. Our proof then implies naturally standard UC security (where all hybrids are local, i.e., not shared) and, by the equivalence shown in [9], full GUC security. Secure realization is proven by providing a simulator $\mathcal{S}_{\text{ledg}}$ in the ideal world (with access to the ledger, global clock and random oracle) such that the protocol execution is indistinguishable from the ideal-world execution with the ledger functionality and the simulator. The simulator $\mathcal{S}_{\text{ledg}}$ is given in detail in Section C. The simulator basically runs internally an entire protocol execution and emulates this real-world view in a black-box way towards the real world adversary $\mathcal{A}$. This simulation can be done perfectly, as nothing restricts the simulator in evaluating, in each round what the corresponding party does in the protocol upon a maintain command (including aborts of protocols due to key collisions in $\mathcal{F}_{\mathsf{INIT}}$ for example). Also, the simulator can extract the ledger state from the emulated blockchains (procedure ExtendLedgerState), and the views of honest parties on this state (procedure AdjustView). The only events that prevent a successful simulation are therefore when the ledger functionality does not allow the simulator to specify the state and the view appropriately. Simulating a ledger state fails, if the simulator encounters a violation of the common prefix property (in this case the simulation aborts as seen in the code of $\mathcal{S}_{\text{ledg}}$ when flag BAD-CP is triggered). Similarly, if the state grows too slowly, the simulator aborts (flag BAD-CG), or the state contains too few honestly generated blocks (flag BAD-CQ). This events, however, hold except with negligible probability in the parameter $k$ which follows exactly as proven in the previous sections (under the given assumptions). Considering that the analysis conditions no collisions among random oracle outputs, the corresponding total error probability of Theorem 2 can be invoked here and yields an upper bound of $\exp(-\Omega(\kappa)) + \exp(\ln \mathsf{poly}(\kappa) - \Omega(k)) + \exp(\ln \mathsf{poly}(\kappa) - \Omega(R))$, where $\mathsf{poly}(\kappa)$ denotes the polynomial upper bound on the runtime of $\mathcal{Z}$ measured with respect to the security parameter $\kappa$. Note that in particular, the parameters $L$ and $Q$ of the security bound can simply be upper bounded by this polynomial.

The remaining technical properties are straightforward to verify: first, pointers of alert parties are monotonically increasing, since the chains adopted by alert parties are monotonically increasing in size (recall from 6.2 that the new maxvalid-bg applied by alert parties essentially implements the longest-chain-rule but does not need checkpointing). The pointers of alert parties can also not be too far apart, i.e., the slackness is upper bounded by $\texttt{windowSize} = k$ (meaning they fall within a window of size windowSize), as otherwise the common-prefix property is violated in that execution (if the prefix of the chain known to any honest party was further away than $k$ blocks from the prefix of the actual longest chain, this would yield a fork and violate common-prefix). Second, the synchronization time does not take more than Delay time as given in the theorem statements, as this is exactly the time until the a newly joining party will have received a synchronizing chain and all honest transactions that were sent out (and still are valid) before this party joined the network (note that the round-trip time is just $2\Delta$). Hence, the overall bound is exactly the time it takes to receive a synchronizing chain as by Lemma 2.

Overall, this means that except with negligible probability, the simulator will not abort and does never violate the ledger's policy (as specified by (ExtendPolicy) or the additional restrictions on pointers into the unique ledger state. □

## 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] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.

[3] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.

[4] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014.

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

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

[7] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016.

[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] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001.

[11] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 597–608. ACM Press, November 2014.

[12] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global PKI. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 265–296. Springer, Heidelberg, March 2016.

[13] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `https://eprint.iacr.org/2016/919`.

[14] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.

[15] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, Heidelberg, January 2005.

[16] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. Random oracles with(out) programmability. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 303–320. Springer, Heidelberg, December 2010.

[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] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. Cryptology ePrint Archive, Report 2018/248, 2018. `https://eprint.iacr.org/2018/248`.

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

[20] Martin Hirt and Vassilis Zikas. Adaptively secure broadcast. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 466–485. Springer, Heidelberg, May / June 2010.

[21] 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.

[22] 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.

[23] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 30–41. ACM Press, November 2014.

[24] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 418–429. ACM Press, October 2016.

[25] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

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

[27] 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.

[28] 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.

[29] Alexander Russell, Cristopher Moore, Aggelos Kiayias, and Saad Quader. Forkable strings are rare. Cryptology ePrint Archive, Report 2017/241, 2017. `https://eprint.iacr.org/2017/241`.

# A The Model (Cont'd)

This appendix includes complementary material to Section 2.

## A.1 Functionalities With Dynamic Party Sets

All our functionalities and global setups handle a dynamic party set. The employed mechanism works as follows: such functionalities include the instructions that allow honest parties to join or leave the set $\mathcal{P}$ of players that the functionality interacts with, and inform the adversary about the current set of registered parties:[22] Unless otherwise specified, the term party refers to an active protocol machine and, as usual in UC, includes the session information. The standard mechanism is as follows (and applies to all functionalities unless otherwise indicated by the functionality specification).

- Upon receiving $(\textsc{register}, \mathrm{sid})$ from some party $U_p$ (or from $\mathcal{A}$ on behalf of a corrupted $U_p$), set $\mathcal{P} = \mathcal{P} \cup \{U_p\}$. Return $(\textsc{register}, \mathrm{sid}, U_p)$ to the caller.
- Upon receiving $(\textsc{de-register}, \mathrm{sid})$ from some party $U_p \in \mathcal{P}$, the functionality sets $\mathcal{P} := \mathcal{P} \setminus \{U_p\}$ and returns $(\textsc{de-register}, \mathrm{sid}, U_p)$ to the caller.
- Upon receiving $(\textsc{is-registered}, \mathrm{sid})$ from some party $U_p$, return $(\textsc{register}, \mathrm{sid}, b)$ to the caller, where the bit $b$ is 1 if and only if $U_p \in \mathcal{P}$.
- Upon receiving $(\textsc{get-registered}, \mathrm{sid})$ from $\mathcal{A}$, the functionality returns $(\textsc{get-registered}, \mathrm{sid}, \mathcal{P})$ to $\mathcal{A}$.

For simplicity in the description of the functionalities, for a party $U_p \in \mathcal{P}$ we will use $U_p$ to refer to this party's ID. In addition to the above registration instructions, global setups, i.e., shared functionalities that are available both in the real and in the ideal world and allow parties connected to them to share state [9], allow also UC functionalities to register with them. We note in passing that although we allow no communication between functionalities, we will allow functionalities to communicate with global setups along the lines of [12, Section 2].

Concretely, global setups include, in addition to the above party registration instructions, two registration/de-registration instructions for functionalities:

- Upon receiving $(\textsc{register}, \mathrm{sid}_C)$ from a functionality $\mathcal{F}$ (in session sid), set $F := F \cup \{(\mathcal{F}, \mathrm{sid})\}$.
- Upon receiving $(\textsc{de-register}, \mathrm{sid}_C)$ from a functionality $\mathcal{F}$ (in session sid, set $F := F \setminus \{(\mathcal{F}, \mathrm{sid})\}$.
- Upon receiving $(\textsc{get-registered-f}, \mathrm{sid}_C)$ from $\mathcal{A}$, return $(\textsc{get-registered-f}, \mathrm{sid}_C, F)$ to $\mathcal{A}$.

## A.2 The Communication Network

We specify the multicast network with bounded delay in the following. The network is modeled as a local functionality. However, we conjecture that it is straightforward to make it global since the simulator has to simulate all the messages on the network. Since we do not consider properties such as network congestion, we choose not to model it as a global functionality for simplicity. As it is sometimes useful to distinguish (the same kind of network) according to the values sent over the network, we use the notation $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc},\Delta}$ and $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx},\Delta}$ to distinguish chain and transaction multicast in the protocol. However, since both networks can be realized from a single network we often just refer to $\mathcal{F}_{\text{N-MC}}^{\Delta}$ for simplicity.

[22] Note that making the set of parties dynamic means that the adversary needs to be informed about which parties are currently in the computation so that he can chose how many (and which) parties to corrupt.

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

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

– **Honest sender multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from some $U_p \in \mathcal{P}$, where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set, choose $n$ new unique message-IDs $\text{mid}_1, \ldots, \text{mid}_n$, initialize $2n$ new variables $D_{\text{mid}_1} := D^{MAX}_{\text{mid}_1} \ldots := D_{\text{mid}_n} := D^{MAX}_{\text{mid}_n} := 1$, set $\vec{M} := \vec{M}||(m, \text{mid}_1, D_{\text{mid}_1}, U_1)|| \ldots ||(m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, and send $(\text{MULTICAST}, \text{sid}, m, U_p, (U_1, \text{mid}_1), \ldots, (U_n, \text{mid}_n))$ to the adversary.
– **Adversarial sender (partial) multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, (m_{i_1}, U_{i_1}), \ldots, (m_{i_\ell}, U_{i_\ell})$ from the adversary with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}$, choose $\ell$ new unique message-IDs $\text{mid}_{i_1}, \ldots, \text{mid}_{i_\ell}$, initialize $\ell$ new variables $D_{\text{mid}_{i_1}} := D^{MAX}_{\text{mid}_{i_1}} := \ldots := D_{\text{mid}_{i_\ell}} := D^{MAX}_{\text{mid}_{i_\ell}} := 1$, set
$\vec{M} := \vec{M}||(m_{i_1}, \text{mid}_{i_1}, D_{\text{mid}_{i_1}}, U_{i_1})|| \ldots ||(m_{i_\ell}, \text{mid}_{i_\ell}, D_{\text{mid}_{i_\ell}}, U_{i_\ell})$, and send
$(\text{MULTICAST}, \text{sid}, (m_{i_1}, U_{i_1}, \text{mid}_{i_1}), \ldots, (m_{i_\ell}, U_{i_\ell}, \text{mid}_{i_\ell})$ to the adversary.
– **Honest party fetching.** Upon receiving $(\text{FETCH}, \text{sid})$ from $U_p \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of $U_p$ if $U_p$ is corrupted):
  1. For all tuples $(m, \text{mid}, D_{\text{mid}}, U_p) \in \vec{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\vec{M}_0^{U_p}$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \text{mid}, D_{\text{mid}}, U_p)$ with $D_{\text{mid}} = 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}_0^{U_p}$ from $\vec{M}$, and send $\vec{M}_0^{U_p}$ to $U_p$.
– **Adding adversarial delays.** Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\text{mid}_{i_j}}, \text{mid}_{i_j})$:
  If $D^{MAX}_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}} \leq \Delta$ and mid is a message-ID registered in the current $\vec{M}$, set $D_{\text{mid}_{i_j}} := D_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$ and set $D^{MAX}_{\text{mid}_{i_j}} := D^{MAX}_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$; otherwise, ignore this pair.
– **Adversarially reordering messages.** Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from the adversary, if mid and mid′ are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \text{mid}, D_{\text{mid}}, \cdot)$ and $(m, \text{mid}', D_{\text{mid}'}, \cdot)$ in $\vec{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.

### A.3 Modeling Synchrony

As in [3], the basic functionality to capture a round-based protocol is the clock-functionality described below. In this functionality, each registered party can update the clock and once all honest parties have done so, the clock advances by one tick. In addition, every party can query the clock to read the (logical) time. A shared (global) clock allows each party to synchronize each session it participates in.

An important property thereby is that for an ideal-world functionality to be UC implementable by a synchronous protocol, it needs to keep track of the number of activations that an honest party gets—such that the advancement of the ideal process is identical to advancement of the real world process. This requires that the protocol itself, when described as a UC interactive Turing-machine instance, has a predictable behavior (per session) when it comes to the pattern of activations that it needs before it sends the clock an update command. This is captured by defining a predictor $\mathsf{predict\text{-}time}_{\Pi}(\vec{\mathcal{I}}_H^T)$ of the time, given as input the timed honest-input sequence.[23] We restate this property formalized in [3] here for completeness in Definition 7.

**Definition 7.** *A $\mathcal{G}_{\text{CLOCK}}$-hybrid protocol $\Pi$ has a* predictable synchronization pattern *iff there exist an algorithm $\mathsf{predict\text{-}time}_{\Pi}(\cdot)$ such that for any possible execution of $\Pi$ in a session sid (i.e., for any adversary and environment, and any choice of random coins) the following holds: If $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$ is the corresponding timed honest-input sequence for this execution, then for any $i \in [m-1]$ :*

$$\mathsf{predict\text{-}time}_{\Pi}((x_1, id_1, \tau_1), \ldots, (x_i, id_i, \tau_i)) = \tau_{i+1},$$

*where $\tau_{i+1}$ refers to the clock time of this session.*

[23] The timed honest-input sequence looks like $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$ where $((x_1, id_1), \ldots, (x_m, id_m))$ are the honest inputs corresponding to an execution (up to a certain point), and for each $i \in [n]$, $\tau_i$ is the time of the global clock when input $x_i$ was handed to ITI (or party) $id_i$ in the session sid (recall that in UC this means that $id = (\text{pid}, \text{sid})$ for some bitstring pid).

Having such a predictor is beneficial in modeling synchronous protocols in UC, as the theorems and the proofs only depend on this function but not on the *exact* number of activations of a party in each round. For example, if an additional computation step requires one activation more, then the only thing that changes is the concrete specification of the function $\mathsf{predict\text{-}time}_{\Pi}$ but the theorems stay the same.

**Functionality** Functionality $\mathcal{G}_{\text{CLOCK}}$

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

For each session sid the clock maintains a variable $\tau_{\text{sid}}$. For each identity $U_p := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable $d_{U_p}$. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})}$ (all integer variables are initially 0).

*Synchronization:*

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $U_p \in \mathcal{P}$ set $d_{U_p} := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, U_p)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return $(\text{CLOCK-READ}, \text{sid}, \tau_{\text{sid}})$ to the requestor (where sid is the sid of the calling instance).

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

We next show that the protocol has a predictable synchronization pattern according to Definition 7.

**Lemma 3.** *The protocol Ouroboros-Genesis satisfies Definition 7.*

*Proof.* We will show that there is an easy and efficient algorithm $\mathsf{predict\text{-}time}_{\text{OG}}(\cdot)$ that, given any possible execution of the protocol in some session sid (for any adversary, environment, and choice of random coins), we have that if $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$ is the corresponding timed honest-inputs sequence for this execution, then for any $i \in [m-1]$ :

$$\mathsf{predict\text{-}time}_{\Pi}((x_1, id_1, \tau_1), \ldots, (x_i, id_i, \tau_i)) = \tau_{i+1}.$$

The basic mechanism to predict the clock time is an inductive process. The first advancement of the clock from $\tau = 0$ to $\tau = 1$ is after all parties $U_p \in \mathcal{S}_{\text{initStake}}$ have received a registration query from the environment and if all additionally registered, uncorrupted parties have sent a clock-update message to the clock. The advancement from $\tau$ to $\tau + 1$ follows by observing that each honest miner that is registered with all global functionalities needs one activation query MAINTAIN-LEDGER followed by an clock-update request from the environment to send his clock-update message (other honest miners do not send such a request). Once every honest party registered with the clock has sent its clock-update message, the clock advances. □

### A.4 The Global Random Oracle Setup

**Functionality** $\mathcal{G}_{\text{RO}}$

The functionality is parameterized by a security parameter $\kappa$. It maintains a set of registered parties $\mathcal{P}$ (initially set to $\emptyset$) and a (dynamically updatable) function table $\mathcal{T}$ (initially $\mathcal{T} = \emptyset$). For simplicity we write $T[x] = \perp$ to denote the fact that no pair of the form $(x, \cdot)$ is in $\mathcal{T}$.

- Upon receiving $(\text{EVAL}, \text{sid}_{RO}, x)$ from some party $U_p \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $U_p$), do the following:

1. If $H[x] = \bot$ sample a value $y$ uniformly at random from $\{0,1\}^\kappa$, set $H[x] \leftarrow y$ and add $(x, T[x])$ to $\mathcal{T}$.
2. Return $(\text{EVAL}, \text{sid}_{RO}, x, H[x])$ to the requester.

### A.5 The Genesis Block Distribution

The functionality $\mathcal{F}_{\mathsf{INIT}}$ describe below was introduces in [14] to formalize the procedure of genesis block creation and distribution.

**Functionality $\mathcal{F}_{\mathsf{INIT}}$**

The functionality $\mathcal{F}_{\mathsf{INIT}}$ is parameterized by the set $U_1, \dots, U_n$ of initial stakeholders $n$ and their respective stakes $s_1, \dots, s_n$. It maintains the set of registered parties $\mathcal{P}$.

- Upon receiving any message from a party, the functionality first sends $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ to the clock to receive the current round. Subsequently:
  - If this is the first (genesis) round and the message is request from some initial stakeholder $U_i$ of the form $(\mathsf{ver\_keys}, sid, U_i, v_i^{\text{vrf}}, v_i^{\text{kes}})$, then $\mathcal{F}_{\mathsf{INIT}}$ stores the verification keys tuple $(U_i, v_i^{\text{vrf}}, v_i^{\text{kes}})$ and acknowledges its receipt. If some of the registered public keys are equal, it outputs an error and halts. Otherwise, it samples and stores a random value $\eta_1 \xleftarrow{\$} \{0,1\}^\lambda$ and constructs a genesis block $(\mathbb{S}_1, \eta_1)$, where $\mathbb{S}_1 = \big((U_1, v_1^{\text{vrf}}, v_1^{\text{kes}}, s_1), \dots, (U_n, v_n^{\text{vrf}}, v_n^{\text{kes}}, s_n)\big)$.
  - If this is not the first round, then do the following
    - ∗ If any of the $n$ initial stakeholders has not send a request of the above form, i.e., a $(\mathsf{ver\_keys}, sid, U_i, v_i^{\text{vrf}}, v_i^{\text{kes}})$-message, to $\mathcal{F}_{\mathsf{INIT}}$ in the genesis round then $\mathcal{F}_{\mathsf{INIT}}$ outputs an error and halts.
    - ∗ Otherwise, if the currently received input is a request of the form $(\mathsf{genblock\_req}, sid, U_i)$ from any (initial or not) stakeholder $U$, $\mathcal{F}_{\mathsf{INIT}}$ sends $(\mathsf{genblock}, sid, (\mathbb{S}_1, \eta_1))$ to the requester.

### A.6 Additional Functionalities/Hybrids Used in the Security Proof

The security of Ouroboros Praos and Genesis is proven in a hybrid world with access to a multicast-network with upper bound on the message delay (unknown to the protocol), a global random oracle, a functionality that idealizes verifiable random functions (VRF), a functionality that idealizes key-evolving signature schemes (KES), and a setup functionality that distributes the initial tokens for proof-of-stake blockchains. The network, clock, RO, and initialization (genesis block), are assumed resources (see Section 2). On the other hand the VRF and KES functionalities are only hybrids used in the proof and are shown to be UC-realizable in [14] by concrete constructions. Therefore, hence they are only employed for simplicity in the proof (the overall security once instantiated by the constructions follows from the UC composition theorem). For completeness we include their definition below.

**Verifiable Random Functions.** The following functionality $\mathcal{F}_{\mathsf{VRF}}$ capturing a verifiable random function was introduced in [14].

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

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

- **Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, sid)$ from a stakeholder $U_i$, hand $(\mathsf{KeyGen}, sid, U_i)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, sid, U_i, v)$ from the adversary, if $U_i$ is honest, verify that $v$ is unique, record the pair $(U_i, v)$ and return $(\mathsf{VerificationKey}, sid, v)$ to $U_i$. Initialize the table $T(v, \cdot)$ to empty.
- **Malicious Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, sid, v)$ from $\mathcal{S}$, verify that $v$ has not being recorded before; in this case initialize table $T(v, \cdot)$ to empty and record the pair $(\mathcal{S}, v)$.

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

  Output $(\mathsf{Verified}, sid, m, y, \pi, f)$ to $P$.

**Key-Evolving Signatures.** Ouroboros Praos and Genesis also make use of a key-evolving signature scheme for signing blocks. In our treatment, we reflect the fact that signing a message is a local operation performed by a slot-leader and invoke the proposed formalism of Camenisch et al. [7] and declare signing request as *restricting*. This means that although activated to provide a signature, the adversary has to provide the answer, i.e. the signature string, to this request immediately (no other output to another protocol machine is allowed) and return the activation token back to the functionality $\mathcal{F}_{\mathsf{KES}}$. In this sense, the adversary and the environment are called *responsive* on signing queries.

The following formalization of key-evolving signatures was given in [14] and we adopt it slightly using the notation from [7, Section 6] to indicate that signing request have to be answered immediately: we indicate the request by prefixing the query with the keyword Respond (and hence this query is considered to be restricting).

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

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

- **Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from a stakeholder $U_S$ , send $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, sid, U_S, v)$ from the adversary, send $(\mathsf{VerificationKey}, sid, v)$ to $U_S$, record the triple $(sid, U_S, v)$ and set counter $\mathsf{k}_{\mathsf{ctr}} = 1$.
- **Sign and Update.** Upon receiving a message $(\mathsf{USign}, sid, U_S, m, j)$ from $U_S$, verify that $(sid, U_S, v)$ is recorded for some $sid$ and that $\mathsf{k}_{\mathsf{ctr}} \leq j \leq T$. If not, then ignore the request. Else, set $\mathsf{k}_{\mathsf{ctr}} = j + 1$ and send $(\mathsf{Respond}, (\mathsf{Sign}, sid, U_S, m, j))$ to the adversary. Upon receiving $(\mathsf{Signature}, sid, U_S, m, j, \sigma)$ from the adversary, verify that no entry $(m, j, \sigma, v, 0)$ is recorded. If it is, then output an error message to $U_S$ and halt. Else, send $(\mathsf{Signature}, sid, m, j, \sigma)$ to $U_S$, and record the entry $(m, j, \sigma, v, 1)$.
- **Signature Verification.** Upon receiving a message $(\mathsf{Verify}, sid, m, j, \sigma, v')$ from some stakeholder $U_i$ do:
  1. If $v' = v$ and the entry $(m, j, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key $v'$ is the registered one and $\sigma$ is a legitimately generated signature for $m$, then the verification succeeds.)

2. Else, if $v' = v$, the signer is not corrupted, and no entry $(m, j, \sigma', v, 1)$ for any $\sigma'$ is recorded, then set $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. (This condition guarantees unforgeability: If $v'$ is the registered one, the signer is not corrupted, and never signed $m$, then the verification fails.)
3. Else, if there is an entry $(m, j, \sigma, v', f')$ recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
4. Else, if $j < \mathsf{k_{ctr}}$, let $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. Otherwise, if $j = \mathsf{k_{ctr}}$, hand $(\mathsf{Verify}, sid, m, j, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, j, \phi)$ from the adversary let $f = \phi$ and record the entry $(m, j, \sigma, v', \phi)$. (This condition guarantees that the adversary is only able to forge signatures under keys belonging to corrupted parties for time periods corresponding to the current or future slots.)

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

Following the treatment of [7], the above functionality is realized by the same construction as in [14] because the simulator in that proof is responsive upon signing requests. We refer to [7] for more details.

### A.7 The Ouroboros Genesis Ledger

We next provide the complete description of the ledger functionality that, as we prove, is implemented by Ouroboros Genesis.

**Functionality $\mathcal{G}_{\text{LEDGER}}$**

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

**Party Management:** The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (as discussed below). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a (currently unregistered) honest party is registered at the ledger, *if it is registered with the clock and the global RO already,* then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$.

**Handling initial stakeholders:** If during round $\tau = 0$, the ledger did not received a registration from each initial stakeholder, i.e., $U_p \in \mathcal{S}_{\text{initStake}}$, the functionality halts.

---

**Upon receiving any input** $I$ from any party or from the adversary, send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and upon receiving response $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ set $\tau_L := \tau$ and do the following if $\tau > 0$ (otherwise, ignore input):

1. Updating synchronized/desynchronized party set:
   (a) Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously) to the ledger, the clock, and the GRO since time $\tau' < \tau_L - \mathtt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.
   (b) For any synchronized party $U_p \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if $U_p$ is not registered to the clock, then consider it desynchronized, i.e., set $\mathcal{P}_{DS} \cup \{U_p\}$.
2. If $I$ was received from an honest party $U_p \in \mathcal{P}$:
   (a) Set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T || (I, U_p, \tau_L)$;

(b) Compute $\vec{N} = (\vec{N}_1, \dots, \vec{N}_\ell) :=$ ExtendPolicy$(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$ and if $\vec{N} \neq \varepsilon$ set $\texttt{state} := \texttt{state}||\mathsf{Blockify}(\vec{N}_1)||\dots||\mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\texttt{state}} := \vec{\tau}_{\texttt{state}}||\tau_L^\ell$, where $\tau_L^\ell = \tau_L||\dots,||\tau_L$.

(c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$ then delete BTX from buffer. Also, reset $\texttt{NxtBC} := \varepsilon$.

(d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ or $\texttt{pt}_j < |\texttt{state}_j|$, then set $\texttt{pt}_k := |\texttt{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

3. If the calling party $U_p$ is *stalled or time-unaware* (according to the defined party classification), then no further actions are taken. Otherwise, depending on the above input $I$ and its sender's ID, $\mathcal{G}_{\text{LEDGER}}$ executes the corresponding code from the following list:

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

- *Reading the state:*
  If $I = (\text{READ}, \text{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\texttt{state}_p := \texttt{state}|_{\min\{\texttt{pt}_p, |\texttt{state}|\}}$ and return $(\text{READ}, \text{sid}, \texttt{state}_p)$ to the requester. If the requester is $\mathcal{A}$ then send $(\texttt{state}, \texttt{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

- *Maintaining the ledger state:*
  If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $U_p \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Else send $I$ to $\mathcal{A}$.

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

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

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

### A.8 Formal Specification of **ExtendPolicy** for the PoS Ledger

The detailed ExtendPolicy for Ouroboros is given below.

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

**function** $\text{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$
  // First, create a default honest client block as alternative:
  $\vec{N}_{\texttt{df}} \leftarrow \text{DefaultExtension}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$ // Extension if adversary violates policy.
  Let $\tau_L$ be current ledger time (computed from $\vec{\mathcal{I}}_H^T$)

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

```
    end for
    // Final checks (if policy is violated, it is enforced by the ledger):
    // Must not proceed too slow or with missing transaction.
    if τ_low > 0 and τ_L − τ_low > maxTime_window then // A sequence of blocks cannot take too much time.
        return N⃗_df
    else if τ_low = 0 and τ_L − τ_low > 2 · maxTime_window then // Bootstrapping cannot take too much time.
        return N⃗_df
    else if oldValidTxMissing then // If not all old enough, valid transactions have been included.
        return N⃗_df
    end if
    return N⃗
end function
```

**Algorithm for Default State Extension**

**function** $\textsc{DefaultExtension}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$

*We assume call-by-value and hence the function has no side effects.*

*The function returns a policy-compliant extension of the ledger state.*

Set $\vec{N}_{\texttt{df}} \leftarrow \texttt{tx}^{\text{base-tx}}_{\text{minerID}}$ of an honest miner
Sort `buffer` according to time stamps.
Let $\vec{\texttt{tx}} = (\texttt{tx}_1, \ldots, \texttt{tx}_\ell)$ be the transactions in `buffer`
Set $\texttt{st} \leftarrow \mathsf{blockify}_{\mathcal{B}}(\vec{N}_{\texttt{df}})$
**repeat**
  Let $\vec{\texttt{tx}} = (\texttt{tx}_1, \ldots, \texttt{tx}_\ell)$ be the current list of (remaining) transactions
  **for** $i = 1$ to $\ell$ **do**
    **if** $\mathsf{ValidTx}_{\mathcal{B}}(\texttt{tx}_i, \texttt{state}||\texttt{st}) = 1$ **then**
      $\vec{N}_{\texttt{df}} \leftarrow \vec{N}_{\texttt{df}}||\texttt{tx}_i$
      Remove $\texttt{tx}_i$ from $\vec{\texttt{tx}}$
      Set $\texttt{st} \leftarrow \mathsf{blockify}_{\mathcal{B}}(\vec{N}_{\texttt{df}})$
    **end if**
  **end for**
**until** $\vec{N}_{\texttt{df}}$ does not increase anymore
// Let $\tau_{\texttt{low}}$ be the time of the block which is $\texttt{windowSize} - 1$ blocks behind the head of the state.
**if** $|\texttt{state}| + 1 \geq \texttt{windowSize}$ **then**
  Set $\tau_{\texttt{low}} \leftarrow \vec{\tau}_{\texttt{state}}[|\texttt{state}| - \texttt{windowSize} + 2]$
**else**
  Set $\tau_{\texttt{low}} \leftarrow 1$ // First epoch starts at time 1 (time 0 is initialization time).
**end if**
$c \leftarrow 1$
**while** $\tau_L - \tau_{\texttt{low}} > \texttt{maxTime}_{\texttt{window}}$ **do**
  Set $\vec{N}_c \leftarrow \texttt{tx}^{\text{base-tx}}_{\text{minerID}}$ of an honest miner
  $\vec{N}_{\texttt{df}} \leftarrow \vec{N}_{\texttt{df}}||\vec{N}_c$
  $c \leftarrow c + 1$
  // Update $\tau_{\texttt{low}}$ to the time of the state block which is $\texttt{windowSize} - c$ blocks behind the head.
  **if** $|\texttt{state}| + c \geq \texttt{windowSize}$ **then**
    Set $\tau_{\texttt{low}} \leftarrow \vec{\tau}_{\texttt{state}}[|\texttt{state}| - \texttt{windowSize} + c + 1]$
  **else**
    Set $\tau_{\texttt{low}} \leftarrow 1$
  **end if**
**end while**
**return** $\vec{N}_{\texttt{df}}$
**end function**

**Fig. 14.** Function to compute a policiy-compliant default ledger-state extension.

# B Ouroboros Genesis as a UC-Protocol (Cont'd)

This appendix includes protocols that have been excluded from the body.

**Chain validation.** The chain validation procedure is given in Figure 15.

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

**if** $\mathcal{C}$ contains future blocks, empty epochs, starts with a block other than **G**, or encodes an invalid state with $\mathsf{isvalidstate}(\vec{\mathtt{st}}) = 0$ **then**
    **return** false
**end if**
**for** each epoch $\mathtt{ep}$ **do**
    // Derive stake distribution and randomness for this epoch from $\mathcal{C}$
    Set $\mathbb{S}_{\mathtt{ep}}^{\mathcal{C}}$ to be the stakeholder distribution at the end of epoch $\mathtt{ep} - 2$ in $\mathcal{C}$.
    Set $\alpha_{p'}^{\mathtt{ep},\mathcal{C}}$ to be the relative stake of any party $U_{p'}$ in $\mathbb{S}_{\mathtt{ep}}^{\mathcal{C}}$ and $T_{p'}^{\mathtt{ep},\mathcal{C}} \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha_{p'}^{\mathtt{ep},\mathcal{C}})$.
    Set $\eta_{\mathtt{ep}}^{\mathcal{C}} \leftarrow H(\eta_{\mathtt{ep}-1}^{\mathcal{C}} \,\|\, \mathtt{ep} \,\|\, v)$ where $v$ is the concatenation of the VRF outputs $y_\rho$ from all blocks in $\mathcal{C}$ from the first two-thirds of slots of epoch $\mathtt{ep} - 1$, and $\eta_1^{\mathcal{C}} \triangleq \eta_1$ from **G**.
    **for** each block $B$ in $\mathcal{C}$ from epoch $\mathtt{ep}$ **do**
        Parse $B$ as $(h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$.
        // Check hash. H(.) is shorthand for a RO-evaluation.
        Set $\mathsf{badhash} \leftarrow (h \neq H(B^{-1}))$, where $B^{-1}$ is the last block in $\mathcal{C}$ before $B$ (possibly the genesis block).
        // Check VRF values
        Parse $crt$ as $(U_{p'}, y, \pi)$ for some $p'$.
        Send $(\mathsf{Verify}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST}, y, \pi, v_{p'}^{\mathrm{vrf}})$ to $\mathcal{F}_{\mathsf{VRF}}$,
            denote its response by $(\mathsf{Verified}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST}, y, \pi, b_1)$.
        Send $(\mathsf{Verify}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, v_{p'}^{\mathrm{vrf}})$ to $\mathcal{F}_{\mathsf{VRF}}$,
            denote its response by $(\mathsf{Verified}, sid, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, b_2)$,
        Set $\mathsf{badvrf} \leftarrow \left(b_1 = 0 \lor b_2 = 0 \lor y \geq T_{U_{p'}}^{\mathtt{ep},\mathcal{C}}\right)$.
        // Check signature
        Send $(\mathsf{Verify}, sid, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, \sigma, v_{p'}^{\mathrm{kes}})$ to $\mathcal{F}_{\mathsf{KES}}$,
            denote its response by $(\mathsf{Verified}, sid, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, b_3)$.
        Set $\mathsf{badsig} \leftarrow (b_3 = 0)$.
        **if** $(\mathsf{badhash} \lor \mathsf{badvrf} \lor \mathsf{badsig})$ **then**
            **return** false
        **end if**
    **end for**
**end for**
**return** true

**Fig. 15.** The chain validation (filtering) protocol.

**The round finish procedure.** Once a party is done its actions in a round it has to advance the synchronous computation by sending the indication to $\mathcal{G}_{\text{CLOCK}}$. Since the functionality is shared and and an update-request by the environment might not be well aligned with the round actions, the protocol simply remembers that such an update has been received. The procedure FinishRound enforces that the protocol only sends the clock-update once (1) the round operations are concluded and (2) the environment has given the command to advance the round.[24]

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

**Protocol** FinishRound($U_p$)

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

**Fig. 16.** The UC-specific round conclusion of a synchronous protocol.

## C The Simulator

Below we present the simulator used in the proof that the UC implementation of Ouroboros Genesis securely realizes the ledger functionality $\mathcal{G}_{\text{LEDGER}}$. The simulator shares the basic structure with the simulator provided in [3] and differs in several low-level details.

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

**Overview:**

- The simulator internally emulates all local UC functionalities by running the code (and keeping the state) of $\mathcal{F}_{\mathsf{VRF}}$, $\mathcal{F}_{\mathsf{KES}}$, $\mathcal{F}_{\mathsf{INIT}}$, $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$, and $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
- The simulator mimics the execution of Ouroboros-Genesis for each honest party $U_p$ (including their state and the interaction with the hybrids).
- The simulator emulates a view towards the adversary $\mathcal{A}$ in a black-box way, i.e., by internally running adversary $\mathcal{A}$ and simulating his interaction with the protocol (and hybrids) as detailed below for each hybrid. To simplify the description, we assume $\mathcal{A}$ does not violate the requirements by the wrapper $\mathcal{W}^{PoS}_{\text{OG}}(\cdot)$ as this would imply no interaction between $\mathcal{S}_{\text{ledg}}$ (i.e., the emulated hybrids) and $\mathcal{A}$.
- For global functionalities, the simulator simply relays the messages sent from $\mathcal{A}$ to the global functionalities (and returns the generated replies). Recall that the ideal world consists of the dummy parties, the ledger functionality, the clock, and the global random oracle.

**Party sets:**

- As defined in the main body of this paper, honest parties are categorized. We denote $\mathcal{S}_{\text{alert}}$ the alert parties (synchronized and executing the protocol) and use $\mathcal{S}_{\text{syncStalled}}$ shorthand for parties that are synchronized (and hence time aware and online) but stalled. Finally, we denote by $\mathcal{P}_{DS}$ all honest but de-synchronized parties (both operational or stalled).
- For each registered honest party, the simulator maintains the local state containing in particular the local chain $\mathcal{C}^{(U_p)}_{\mathsf{loc}}$, the time $t_{\text{on}}$ it remembers when last being online. For each party $U_p$ and clock time $\tau$, the simulator stores a flag $\mathsf{update}_{U_p,\tau}$ (initially $\mathsf{false}$) to remember whether this party has updated its state already in this round. Note that an registered party is registered with all its local hybrids.
- Upon any activation, the simulator will query the current party set from the ledger, the clock, and the random oracle to evaluate in which category an honest party belongs to. If a new honest party is registered to the ledger, it internally runs the initialization procedure of Ouroboros-Genesis.
- We assume that the simulator queries upon any activation for the sequence $\vec{\mathcal{I}}^T_H$, and the current time $\tau$ from the clock. We note that the simulator is capable of determining $\mathsf{predict\text{-}time}(\cdot)$ of $\mathcal{G}_{\text{LEDGER}}$.

**Messages from the Clock:**

- Upon receiving (CLOCK-UPDATE, $\mathrm{sid}_C, U_p$) from $\mathcal{G}_{\text{CLOCK}}$, if $U_p$ is an honest registered party, then remember that this party has received such a clock update (and the environment gets an activation). Otherwise, send (CLOCK-UPDATE, $\mathrm{sid}_C, U_p$) to $\mathcal{A}$.

**Messages from the Ledger:**

– Upon receiving (SUBMIT, $\texttt{BTX}$) from $\mathcal{G}_{\text{LEDGER}}$ where $\texttt{BTX} := (\texttt{tx}, \text{txid}, \tau, U_p)$ forward (MULTICAST, sid, $\texttt{tx}$) to the simulated network $\mathcal{F}_{\text{N-MC}}$ in the name of $U_p$. Output the answer of $\mathcal{F}_{\text{N-MC}}$ to the adversary.
– Upon receiving (MAINTAIN-LEDGER, sid, minerID) from $\mathcal{G}_{\text{LEDGER}}$, extract from $\vec{\mathcal{I}}_H^T$ the party $U_p$ that issued this query. If $U_p$ has already completed its round-task, then ignore this request. Otherwise, execute SIMULATESTAKING$(U_p, \tau)$.

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

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

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

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

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

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

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

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

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

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

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

**procedure** $\textsc{SimulateStaking}(U_p, \tau)$
Simulate the core staking procedure of party $U_p$ as in the protocol in round $\tau$. This includes running procedures FetchInformation and UpdateStakeDist of party $U_p$ (using the emulated network).
**if** $\mathsf{update}_{U_p,\tau}$ **then**
Send $(\textsc{clock-update}, \mathrm{sid}_C, U_p)$ to $\mathcal{A}$ if $\mathcal{S}_{\text{ledg}}$ has received such an input in round $\tau$
**else**
Execute the StakingProcedure and set $\mathsf{update}_{U_p,\tau} \leftarrow \mathsf{true}$
- Includes sending messages to the emulated network $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$.
Before the activation goes to $\mathcal{A}$, execute $\textsc{ExtendLedgerState}(\tau)$.
**end if**
Remember that party $U_p$ has completed for this round $\tau$.
**end procedure**

**procedure** $\textsc{ExtendLedgerState}(\tau)$
**for** each synchronized party $U_p \in \mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ of round $\tau$ **do**
Let $\mathcal{C}^{(U_p)}_{\mathsf{loc}}$ be the party's currently stored local chain.
Determine the number of rounds $\rho^{(U_p)}$ this party legs behind $\tau$, i.e., $\rho^{(U_p)} = \tau - t^{(U_p)}_{\text{on}}$.
Let $\mathcal{C}^{(U_p)}_1, \ldots, \mathcal{C}^{(U_p)}_k$ be the chains contained in the receiver buffer $\vec{M}^{(U_p)}$ of $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ with delay at most $\rho^{(U_p)}$.
Evaluate $\mathcal{C}_{U_p} \leftarrow \mathsf{maxvalid\text{-}bg}(\mathcal{C}^{(U_p)}_{\mathsf{loc}}, \mathcal{C}^{(U_p)}_1, \ldots, \mathcal{C}^{(U_p)}_k)$ and let this chain's encoded state be $\vec{\mathtt{st}}_{U_p}$.
**end for**
Let $\vec{\mathtt{st}}$ be the longest state among all such states $\vec{\mathtt{st}}_{U_p}$, $U_p \in \mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ from above.
Compare $\vec{\mathtt{st}}^{\lceil k}$ with the current state $\mathtt{state}$ of the ledger
**if** $|\mathtt{state}| > |\vec{\mathtt{st}}^{\lceil k}|$ **then** // Only pointers need adjustments
Execute $\textsc{AdjustView}(\mathtt{state})$
**end if**
**if** $\mathtt{state}$ is not a prefix of $\vec{\mathtt{st}}^{\lceil k}$ **then** // Simulation fails
**Abort** simulation: consistency violation among synchronized parties. // Event $\mathsf{BAD\text{-}CP}_k$
**end if**
Define the difference $\mathsf{diff}$ to be the block sequence s.t. $\mathtt{state}||\mathsf{diff} = \vec{\mathtt{st}}^{\lceil k}$.
Parse $\mathsf{diff} := \mathsf{diff}_1 || \ldots || \mathsf{diff}_n$.
**for** $j = 1$ to $n$ **do**
Map each transaction $\mathtt{tx}$ in this block to its unique transaction ID txid. If a transaction does not yet have a txid, then submit it to the ledger first and receive the corresponding txid from $\mathcal{G}_{\textsc{ledger}}$
Let $\mathsf{list}_j = (\mathrm{txid}_{j,1}, \ldots, \mathrm{txid}_{j,\ell_j})$ be the corresponding list for this block $\mathsf{diff}_j$
**if** coinbase $\mathrm{txid}_{j,1}$ specifies a party honest at block creation time **then**
$\mathrm{hFlag}_j \leftarrow 1$
**else**
$\mathrm{hFlag}_j \leftarrow 0$
**end if**
Output $(\textsc{next-block}, \mathrm{hFlag}_j, \mathsf{list}_j)$ to $\mathcal{G}_{\textsc{ledger}}$ (receiving $(\textsc{next-block}, ok)$ as an immediate answer)
**end for**

**if** Fraction of blocks with hFlag $= 0$ in the recent $k$ blocks $> 1 - \mu$ **then**
  **Abort** simulation: chain quality violation. // Event $\mathsf{BAD\text{-}CQ}_{\mu,k}$
**else if** State increases less than $k$ blocks during the last $\frac{k}{\tau_{\mathsf{CG}}}$ rounds **then**
  **Abort** simulation: chain growth violation. // Event $\mathsf{BAD\text{-}CG}_{\tau_{\mathsf{CG}},k/\tau_{\mathsf{CG}}}$
**end if**
// If no bad event occurs, we can adjust pointers into this new state.
Execute AdjustView($\mathtt{state}||\mathsf{diff}$)
**end procedure**

**procedure** AdjustView($\mathtt{state}, \tau$)
// Adjust the view of synchronized parties.
$\mathsf{pointers} \leftarrow \varepsilon$
**for** $U_p \in \mathcal{P}$ of round $\tau$ **do**
  Let $\mathcal{C}_{\mathsf{loc}}^{(U_p)}$ be the party's currently stored local chain.
  Determine the number of rounds $\rho^{(U_p)}$ this party legs behind $\tau$, i.e., $\rho^{(U_p)} = \tau - t_{\text{on}}^{(U_p)}$.
  Let $\mathcal{C}_1^{(U_p)}, \ldots, \mathcal{C}_k^{(U_p)}$ be the chains contained in the receiver buffer $\vec{M}^{(U_p)}$ of $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ with delay at most $\rho^{(U_p)}$.
  Evaluate $\mathcal{C}_{U_p} \leftarrow \mathsf{maxvalid\text{-}bg}(\mathcal{C}_{\mathsf{loc}}^{(U_p)}, \mathcal{C}_1^{(U_p)}, \ldots, \mathcal{C}_k^{(U_p)})$ and let this chain's encoded state be $\vec{\mathtt{st}}_{U_p}$.
**end for**
**for** each synchronized party $U_p \in \mathcal{S}_{\text{alert}} \cup \mathcal{S}_{\text{syncStalled}}$ of round $\tau$ **do**
  Determine the pointer $\mathtt{pt}_{U_p}$ s.t. $\vec{\mathtt{st}}_{U_p}^{\lceil k} = \mathsf{state}|_{\mathtt{pt}_{U_p}}$
  **if** such a pointer value does not exist **then**
    **return** // Call on invalid input or event $\mathsf{BAD\text{-}CP}_k$ occurred
  **end if**
  **if** $\mathsf{update}_{U_p,\tau} = \mathsf{false}$ **then** // Party did not start StakingProcedure in $\tau$.
    $\mathsf{pointers} \leftarrow \mathsf{pointers}||(U_p, \mathtt{pt}_{U_p})$
  **end if** // As otherwise, the new state is only fetched in the next round
**end for**
Output (SET-SLACK, pointers) to $\mathcal{G}_{\text{LEDGER}}$
// Now, adjust the view of de-synchronized parties.
$\mathsf{pointers} \leftarrow \varepsilon$
$\mathsf{desyncStates} \leftarrow \varepsilon$
**for** each de-synchronized party $U_p \in \mathcal{P}_{DS}$ **do**
  **if** $\mathsf{update}_{U_p,\tau} = \mathsf{false}$ **then**
    Set the pointer $\mathtt{pt}_{U_p}$ to be $|\vec{\mathtt{st}}_{U_p}^{\lceil k}|$
    $\mathsf{pointers} \leftarrow \mathsf{pointers}||(U_p, \mathtt{pt}_{U_p})$
    $\mathsf{desyncStates} \leftarrow \mathsf{desyncState}||(U_p, \vec{\mathtt{st}}_{U_p}^{\lceil k})$
  **end if** // As otherwise, the new state is only fetched in the next round
  Output (SET-SLACK, pointers) to $\mathcal{G}_{\text{LEDGER}}$
  Output (DESYNC-STATE, desyncStates) to $\mathcal{G}_{\text{LEDGER}}$
**end for**
**end procedure**

# D Proof-of-Stake Assumptions as a UC Wrapper

This section includes complementary material for the main body. We sketch below the wrapper functionality that is applied to the hybrid functionalities used by Ouroboros-Genesis. For details on more background of functionality wrappers we refer to [3]. In a nutshell, the wrapper observes the advancement of the entire system and checks whether the proportional stake of alert parties, of corrupted or de-synchronized parties, and of stalled parties are withing the allowed range specified as required by our main theorems.

**Functionality $\mathcal{W}_{\mathrm{OG}}^{PoS}(\cdot)$**

The wrapper functionality is parameterized by the bounds $\alpha$,$\beta$ on the alert and participating stake ratio (see Definition 2), respectively, the network delay and a value $\varepsilon > 0$ (the parameter that describes the gap between the honest and adversarial stake). The wrapper is assumed to be registered with the global clock $\mathcal{G}_{\text{CLOCK}}$ and is aware of sets of registered parties, and the set of corrupted parties.

*General:*

- Upon receiving any request $I$ from any party $U_p$ or from $\mathcal{A}$ (possibly on behalf of a party $U_p$ which is corrupted) to a wrapped hybrid functionality, record the request $I$ together with its source and the current time.
- The wrapper keeps track of the active parties and their relative share to the stake distribution.

*Restrictions on obtaining VRF proofs:*

- Upon receiving $(\mathsf{EvalProve}, \mathrm{sid}, \cdot)$ to $\mathcal{F}_{\mathsf{VRF}}$ from $\mathcal{A}$ on behalf of a party $U_p$ which is corrupted or registered but de-synchronized do the following:
  1. If the fraction of alert stake relative to all active stake in this round $\tau$ so far does not satisfy the honest majority condition 4 (of Theorems 1 and 2) then ignore the request.
  2. Otherwise, forward the request to $\mathcal{F}_{\mathsf{VRF}}$ and return to $\mathcal{A}$ whatever $\mathcal{G}_{\mathrm{RO}}$ returns.
- Upon receiving $(\mathsf{EvalProve}, \mathrm{sid}, \cdot)$ to $\mathcal{F}_{\mathsf{VRF}}$ from an alert party $U_p$ do the following:
  1. Forward the request to $\mathcal{F}_{\mathsf{VRF}}$ and return to $\mathcal{A}$ whatever $\mathcal{G}_{\mathrm{RO}}$ returns.
  2. If the minimal fraction (in stake) of participation (of alert parties and in total) as demanded by Theorem 1 (and Theorem 2) is reached in round $\tau$, send $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ to release the clock for this round.
- Any other request is relayed to the underlying functionality (and recorded by the wrapper) and the corresponding output is given to the destination specified by the underlying functionality.

# E Proof of Theorem 1

In this appendix we prove Theorem 1. We begin with a detailed treatment of the relevant machinery from [14] for reasoning about blockchain "forks" and the common prefix property in the semi-synchronous setting. Our setting—which provides the adversary adaptive control over availability of the participating parties—appears to require significant further considerations. In particular, the techniques of [14] assume that slot leaders are elected by an *independent* process, so that various relevant events (such as whether a unique party has been assigned to a particular slot) are independent across distinct slots. The stronger adversary in the dynamic availability setting can conspire to correlate such events. Our analysis handles such correlations by modeling the underlying process of leader assignment as a martingale, and constructs a parallel theory to that of [14] that supports these richer distributions. Our exposition is self-contained; however, in some cases where the particular arguments are similar in spirit to the treatment in [14], we only sketch them.

In Sections E.1 and E.2 we briefly lay out the framework of forks, divergence, and $\Delta$-reduction developed by Kiayias et al. [22] and David et al. [14]. With these definitions set down, we proceed in Section E.3 to the new proofs of divergence for the richer distributions induced by an adversary in the setting with dynamic availability. We then describe the exact distribution of the characteristic strings that arises in the real experiment in Section E.4 and combine these results in E.5 to establish common prefix, chain growth, and chain quality for a single epoch. Finally, we lift these results into the multi-epoch setting in Section E.6.

Note that Theorem 1, and hence also this whole section, works in the so-called setting with static $\mathcal{F}_{\text{N-MC}}$-registration as defined in Section 4.2, postulating that all honest parties are registered with the network functionality $\mathcal{F}_{\text{N-MC}}$ from the beginning and never deregister. Therefore, all honest parties are guaranteed to be *online* and *synchronized*. However, not all honest parties are guaranteed to be *alert*, as some of them might still be *time-unaware* or *stalled* (cf. Fig. 1).

We approach these two classes of parties in different ways in the formal treatment below: for *stalled* honest parties that are *time-aware*, we treat the slots where they would be the only slot leaders as empty slots. Looking ahead, this leads to the adaptive control of the environment over the emptiness of slots (via stalling alert parties), which warrants our martingale-based treatment. On the other hand, the honest parties that are *time-unaware* pose a risk for all slots for which they are eligible slot leaders, as even though they are not corrupted at the moment, they are not able to properly evolve their key and hence their later corruption would have detrimental effects for such slots. Therefore, we account for these parties by treating them identically to adversarial parties (namely, including them in the set of *active* parties), even though formally they are not adversarial.

### E.1 Forks and Divergence in the Semi-synchronous Setting

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

**Definition 8 (Characteristic string).** *Let $S = \{\mathtt{sl}_1, \ldots, \mathtt{sl}_R\}$ be a sequence of slots of length $R$; consider an execution (with adversary $\mathcal{A}$ and environment $\mathcal{Z}$) of the protocol. For a slot $\mathtt{sl}_j$, let $\mathrm{P}(j)$ denote the set of active parties assigned to be slot leaders for slot $j$ by the protocol. We define the* characteristic string *$w \in \{0,1,\perp\}^R$ of $S$ to be the random variable so that*

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

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

We emphasize that the characteristic string resulting from an execution is determined by both the nonce (and the effective leader selection process), the adaptive adversary $\mathcal{A}$, and the environment $\mathcal{Z}$ (which, in particular, determines the stake distribution).

*Remark 3.* A reader familiar with the treatment in [14] will notice that Definition 8 syntactically differs from the definition of a characteristic string in [14], by only considering *active* parties, and by assigning the symbol 0 only to slots that have a unique *alert* slot leader, as opposed to a unique *honest* one. This is because the analysis in [14] does not consider stalled parties, and hence there an honest party is always alert and all parties are active. The semantics of the definition is maintained: a slot labeled by 0 in both cases guarantees that there is will be exactly one block created for this slot, and it will be created according to the protocol. Additionally, a slot labeled by $\perp$ guarantees that no party (either honest or adversarial) is in the position to create a block for this slot. This syntactic difference propagates also to some of the following definitions and statements, we will refrain from pointing it out repeatedly.

The notion of a $\Delta$-fork is the analytic tool developed by David et al. [14] to reason about the various blockchains that can be induced by an adversary in the $\Delta$-synchronous setting with a particular characteristic string.

**Definition 9 ($\Delta$-fork).** *Let $w \in \{0,1,\perp\}^k$ and $\Delta$ be a non-negative integer. Let $A = \{i \,|\, w_i \neq \perp\}$ denote the set of potentially active indices, and let $H = \{i \,|\, w_i = 0\}$ denote the set of uniquely alert indices. A* $\Delta$-fork *for the string $w$ is a rooted tree $F = (V,E)$ with a labeling $\ell : V \to \{0\} \cup A$ so that*

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

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

The relationship between executions and $\Delta$-forks is formally described in [14]. Here we only recall the basic intuition: With an execution of Ouroboros Genesis we may associate the collection of all valid blockchains that were adopted by honest players as a result of their application of the maxvalid rule. Observe that any two blockchains held by honest players agree on some common prefix (including, at the very least, the genesis block); on the other hand, aside from this common prefix, the blockchains are entirely disjoint. Thus the union of these blockchains forms a natural "tree of blocks", which is reflected by the notion of fork above. Indeed, the axiom (ii) reflects the fact that blocks in a valid blockchain must be associated with strictly increasing time slots, while axiom (iii) reflects the fact that an honest, alert slot leader emits exactly one block (associated with that slot). The axiom (iv) reflects the fact that an honest party $p$ at time $t$ must have received any blocks produced by honest parties at times prior to $t - \Delta$; thus the depth of any block produced by $p$ must exceed the depths of those blocks produced by these earlier honest parties. Thus, while a fork is clearly an abstraction that neglects some aspects of the execution, it does capture its salient features with respect to common prefix violations; see Remark 4 below.

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

A remark on the intuition behind viability: A viable tine is one which—at least in principle—could have been accepted as the longest chain by an alert party. In particular, if the last block of the chain is associated with slot $t$, the chain must have length at least that of all honest chains produced no later than $t - \Delta$, as these would necessarily be observed by any alert player at time $t + 1$.

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

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

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

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

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

*Remark 4.* Divergence provides an immediate bound on common prefix violations. In particular, any execution of the protocol inducing a characteristic string $w$ produces honest blockchains satisfying the $\mathrm{div}_{\Delta}(w)$-common prefix property.

Given the above, we will now focus on bounding the $\Delta$-divergence of characteristic strings arising from protocol executions.

### E.2 The Reduction Mapping

David et al. [14] provided a method for bounding $\Delta$-divergence by establishing a direct connection between $\Delta$-divergence and divergence in the synchronous setting (when $\Delta = 0$). We will rely on this machinery and here record its basic tools.

**Definition 12 (Synchronous characteristic strings and forks).** *A* synchronous characteristic string *is an element of* $\{0,1\}^*$. *A* synchronous fork *$F$ for a (synchronous) characteristic string $w$ is a 0-fork $F \vdash_0 w$.*

**Definition 13 (Reduction mapping [14]).** *For $\Delta \in \mathbb{N}$, we define the function $\rho_\Delta \colon \{0,1,\perp\}^* \to \{0,1\}^*$ inductively as follows:*

$$\begin{aligned}
\rho_\Delta(\epsilon) &= \epsilon, \\
\rho_\Delta(\perp \| w') &= \rho_\Delta(w'), \\
\rho_\Delta(1 \| w') &= 1 \| \rho_\Delta(w'), \\
\rho_\Delta(0 \| w') &= \begin{cases} 0 \| \rho_\Delta(w') & \text{if } w' \in \perp^{\Delta-1} \| \{0,1,\perp\}^*, \\ 1 \| \rho_\Delta(w') & \text{otherwise.} \end{cases}
\end{aligned} \tag{9}$$

*We call $\rho_\Delta$ the* reduction mapping for delay $\Delta$*. It will be convenient for us to naturally extend the definition of $\rho_\Delta$ to infinite strings over the alphabet $\{0,1,\perp\}$.*

The reduction map provides the basic connection between $\Delta$-divergence and (synchronous) divergence. This is reflected by the lemma below, established by David et al. [14].

**Lemma 4 ([14]).** *Let $w \in \{0,1,\perp\}^*$. Then $\operatorname{div}_\Delta(w) \leq \operatorname{div}_0(\rho_\Delta(w))$.*

We will require also a lemma controlling the behavior of reduction for prefixes of a given string. Here we use the notation $x \prec y$ to indicate the the string $x$ is a prefix of the string $y$.

**Lemma 5 (Implicit in [14]).** *If $w, w' \in \{0,1,\perp\}^*$ and $w \prec w'$, then $\rho_\Delta(w)^{\lceil \Delta} \prec \rho_\Delta(w')$.*

*Proof.* The proof proceeds by induction on the length of $w$. When $|w| \leq \Delta$, observe that $|\rho_\Delta(w)| \leq |w| \leq \Delta$ and hence $\rho_\Delta(w)^{\lceil \Delta} = \epsilon$. Otherwise $|w| > \Delta$ and we may write $w = ax$ for a single symbol $a$ (and a substring $x$). According to the definition, $\rho_\Delta(ax) = \alpha \rho_\Delta(x)$ for some $\alpha \in \{\epsilon, 0, 1\}$ that is determined solely by the first $\Delta$ symbols of $w$; these agree with $w'$. By induction $\rho(x) = \rho(x')$, where $w' = ax'$, which concludes the proof. ☐

### E.3 Reduction and Divergence with Stalled Parties

With these definitions and lemmas behind us, we are prepared to bound divergence (and common prefix) in our setting with stalled parties.

**Definition 14 (The characteristic conditions).** *Consider a family of random variables $W_1, \ldots, W_n$ taking values in $\{0,1,\perp\}$. We say that they satisfy the $(f;\gamma)$-*characteristic conditions *if, for each $k \geq 1$,*

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

*In the expressions above, conditioning on a collection of random variables indicates that the statement is true for any conditioning on the values taken by variables. We may naturally apply the same terminology to infinite sequences of variables taking values in $\{0,1,\perp\}$.*

Specifically, for an adversary constrained to $(1-\epsilon)/2$ stake ratio, the characteristic string $w_1, \ldots, w_R$ induced for an epoch of length $R$ roughly satisfies the $(f;(1+\epsilon)/2)$-characteristic conditions. (We lay out the exact details in the next section.) Our strategy for bounding $\operatorname{div}_\Delta(w)$ will be to analyze the structure of the induced distribution $\rho_\Delta(w)$ (assuming that $w$ satisfies the characteristic conditions) and then directly bound the (synchronous) divergence of the resulting (synchronous) characteristic string.

**The structure of the reduced distribution $\rho_\Delta(w)$.** As mentioned above, we begin by analyzing the structure of the distribution given by $\rho_\Delta(w)$. Specifically, we will show that these random variables are almost *super*-binomial, in the sense that after trimming a short suffix, they satisfy a family of martingale conditions which guarantee that each random variable, conditioned on all prior values, takes the value 0 with probability at least $\gamma(1-f)^{\Delta-1}$. Finally, we appeal to a theorem of Kiayias et al. [22] and Russell et al. [29] to establish that $\rho_\Delta(w)$ is unlikely to have large divergence.

**Definition 15 (The super-binomial martingale conditions).** *Consider a family of random variables $X_1, \ldots, X_n$ taking values in $\{0,1\}^n$. We say that they satisfy the* $\gamma$-super-binomial martingale *conditions (or, simply, the* $\gamma$-martingale *conditions) if*

$$\Pr[X_k = 0 \,|\, X_1, \ldots, X_{k-1}] \geq \gamma\,, \text{ and hence}$$
$$\Pr[X_k = 1 \,|\, X_1, \ldots, X_{k-1}] \leq 1-\gamma\,.$$

*We may naturally apply the same terminology to infinite sequences of variables taking values in* $\{0,1\}$.

It is convenient to explore first the structure of an infinite sequence of these variables, as these do not require any "trimming" in order to provide the martingale conditions.

**Lemma 6 (Structure of the induced distribution without boundary conditions).** *Let $W = W_1, W_2, \ldots$ be an infinite sequence of random variables, each taking values in $\{\perp, 0, 1\}$, which satisfy the $(f;\gamma)$-characteristic conditions and let*

$$X = \rho_\Delta(W)$$

*be the random variables obtained by applying the reduction mapping (for delay $\Delta$) to $W$. Then $X = X_1, \ldots,$ satisfy the $\gamma(1-f)^{(\Delta-1)}$-martingale conditions.*

*Proof.* For each $k \geq 1$ we wish to establish that the random variables $X_1, \ldots, X_k$ satisfy the $\gamma(1-f)^{(\Delta-1)}$-super-binomial martingale conditions. We prove these conditions under further conditioning. Specifically, we say that a finite sequence $w_1, \ldots, w_\ell$, where each $w_i \in \{0, 1, \perp\}$, is a *$t$-sequence* if exactly $t$ of the $w_i$ are elements of $\{0,1\}$. For a $(k-1)$-sequence $w$, let $E_w$ denote the event that $W_i = w_i$ (for each $1 \leq i \leq \ell$) and that $W_{\ell+1} \neq \perp$. Observe that these events $E_w$, taken over all $(k-1)$-sequences $w$ of all possible lengths $\ell$, partition the probability space over which $W_1, W_2, \ldots$ is defined. Furthermore, for any $(k-1)$-sequence $w$, conditioning on $E_w$ determines the random variables $X_1, \ldots, X_{k-1}$; we write $\rho_\Delta(w)$ to denote the unique assignment to $X_1, \ldots, X_{k-1}$ resulting from this $(k-1)$-sequence $w$. It follows that, for any fixed $x_1, \ldots, x_{k-1}$, the events $E_w$ for which $\rho_\Delta(w) = x_1, \ldots, x_{k-1}$ partition the event that the random variables $X_1, \ldots, X_{k-1}$ take the values $x_1, \ldots, x_{k-1}$. Finally, observe that—conditioned on any specific $E_w$—the $(f,\gamma)$-characteristic conditions guarantee that $(W_\ell, W_{\ell+1}, \ldots, W_{\ell+(\Delta-1)}) = (0, \perp, \ldots, \perp)$ with probability at least $\gamma(1-f)^{\Delta-1}$. In this case, $X_k = 0$, and we conclude that

$$\Pr[X_k = 0 \,|\, E_w] \geq \gamma(1-f)^{\Delta-1}\,.$$

It follows that for any fixed values $x_1, \ldots, x_{k-1}$,

$$\Pr[X_k = 0 \,|\, X_i = x_i] \geq \gamma(1-f)^{\Delta-1}\,,$$

as desired. ☐

We record two immediate applications of Azuma's inequality for random variables satisfying the $\gamma$-super-binomial martingale conditions.

**Lemma 7.** *Let $X_1, \ldots, X_n$ satisfy the $\gamma$-super-binomial martingale conditions with $\gamma \geq 1/2$. Then, for any $\delta > 0$,*

$$\Pr[\#_0(X) \leq (1-\delta)\gamma n] \leq \exp\left(-\delta^2 n/2\right) \tag{10}$$

*and*

$$\begin{aligned}\Pr[\#_0(X) - \#_1(X) \leq (1-\delta)(2\gamma-1)n] &\leq \exp\left(-\frac{\delta^2(2\gamma-1)^2 n}{8\gamma^2}\right) \\ &\leq \exp\left(-\delta^2(2\gamma-1)^2 n/8\right)\,,\end{aligned} \tag{11}$$

*where* $\#_0(X) = |\{i \,|\, X_i = 0\}|$ *and* $\#_1(X) = |\{i \,|\, X_i = 1\}|$.

*Proof.* For (10), consider the random variables $H_k = \sum_{i=1}^{k}((1-X_i)-\gamma) = \#_0(X_1,\ldots,X_k) - k\gamma$. Observe that $\mathbb{E}[H_k \,|\, H_1,\ldots H_{k-1}] \geq H_{k-1}$ and that $|H_k - H_{k-1}| \leq \max(\gamma, 1-\gamma) = \gamma$, as $\gamma \geq 1/2$. Applying Azuma's inequality (Theorem 9 in Appendix F) to the variables $H_k$ yields

$$\Pr[H_n \leq -\delta\gamma n] \leq \exp(-\delta^2 n/2)\,,$$

equivalent to (10). As for (11), consider the random variables

$$B_k = 2\sum_{i=1}^{k}(1 - X_i - \gamma) = \big(\#_0(X_1\ldots X_k) - \#_1(X_1\ldots X_k)\big) - k(2\gamma-1)\,.$$

Then $\mathbb{E}[B_k \,|\, B_1,\ldots,B_{k-1}] \geq B_{k-1}$ and $|B_t - B_{t-1}| \leq 2\gamma$ as $\gamma \geq 1/2$; applying Azuma's inequality to the random variables $B_k$ yields (11). ☐

**Lemma 8 (Structure of the induced distribution).** *Let* $W = W_1\cdots W_n$ *be a sequence of random variables, each taking values in* $\{\perp, 0, 1\}$*, which satisfy the* $(f;\gamma)$*-characteristic conditions and let*

$$X = X_1\cdots X_\ell = \rho_\Delta(W_1\cdots W_n)$$

*be the random variables obtained by applying the reduction mapping (for delay* $\Delta$*) to* $W$*. Then there is a sequence of random variables* $Z_1, Z_2, \ldots$*, each taking values in* $\{0,1\}$*, so that*

*(i) the random variables* $Z_1,\ldots,$ *satisfy the* $\gamma(1-f)^{\Delta-1}$*-martingale conditions;*
*(ii)* $X_1,\ldots,X_{\ell-\Delta} = \rho_\Delta(W)^{\lceil\Delta}$ *is a prefix of* $Z_1Z_2\cdots$.

*Under the further condition that* $\Pr[W_i = \perp \,|\, W_1,\ldots,W_{i-1}] \leq (1-a)$, *we also have:*

*(iii) the random variable* $\ell$ *satisfies, for any* $\delta > 0$,

$$\Pr[\ell < (1-\delta)an] \leq \exp\left(-\frac{\delta^2 a^2 n}{2(1-a)^2}\right) \leq \exp\left(-\delta^2 a^2 n/2\right)\,; \tag{12}$$

*(iv) finally, if* $\gamma(1-f)^{\Delta-1} \geq (1+\epsilon)/2$ *for some* $\epsilon \geq 0$ *then*

$$\Pr\left[\#_0(X) < \frac{(1+\epsilon)an}{4} - \Delta\right] \leq \exp\left(-\frac{a^2 n}{32}\right) + \exp\left(-\frac{an}{64}\right) \leq 2\exp\left(-\frac{a^2 n}{64}\right) \tag{13}$$

*and*

$$\Pr\left[\#_0(X) - \#_1(X) < \frac{\epsilon a n}{4} - 2\Delta\right] \leq \exp\left(-\frac{a^2 n}{8}\right) + n\exp\left(-\frac{\epsilon^2 a n}{64}\right) \leq (n+1)\exp\left(-\frac{\epsilon^2 a^2 n}{64}\right)\,. \tag{14}$$

*Proof.* Treat the random variables $W = W_1\cdots W_n$ as the first $n$ symbols of an infinite sequence $W_1W_2\cdots$ of random variables satisfying the $(f,\epsilon)$-characteristic conditions. It is clear that such an infinite sequence of variables exists, as the random variables appearing in the extension $W_{n+1},\ldots$ can be taken to be i.i.d. with a coordinatewise distribution that satisfies the $(f;\gamma)$-characteristic conditions with equality. Then define

$$Z_1Z_2\cdots \triangleq \rho_\Delta(W_1W_2\cdots)\,,$$

we wish to show that these variables satisfy the statement of the theorem.

In light of Lemma 6, the random variables $Z_1, Z_2, \ldots$ satisfy the $\gamma(1-f)^{\Delta-1}$-martingale conditions as needed for (i). By Lemma 5,

$$\rho_\Delta(W_1 \ldots W_n)^{\lceil \Delta} \prec \rho_\Delta(W_1 W_2 \ldots) = Z_1 Z_2 \ldots ,$$

proving (ii).

The bound (12) on $\ell$ follows by considering the random variables

$$A_i \triangleq \begin{cases} 0 & \text{if } W_i = \perp, \\ 1 & \text{if } W_i \neq \perp, \end{cases}$$

so that $\ell = \sum_{i=1}^{n} A_i$. Then $\Pr[A_i = 1 \mid A_1, \ldots, A_{i-1}] \geq a$ and applying Azuma's inequality (Theorem 9) to the random variables $B_t \triangleq \sum_{i=1}^{t}(A_i - a)$ yields the result.

With this length bound established, we note that $\ell \leq (3/4)an$ with probability no more than $\exp(-a^2 n/32)$ and, in light of (ii), when $\ell \geq (3/4)an$ we must have $\#_0(X) \geq \#_0(Z_1, \ldots, Z_{3an/4}) - \Delta$. Applying the bound of (10) to the $Z_i$ with $\delta = 1/4$, we conclude that the probability that

$$\#_0(Z_1, \ldots, Z_{3an/4}) \leq \frac{(1+\epsilon)an}{4} \leq \frac{1+\epsilon}{2} \cdot \frac{3an}{4} \cdot \frac{3}{4}$$

is no more than $\exp(-(3/4)an/32) \leq \exp(-an/64)$; taking the union bound over these two bad events yields (13).

Finally, consider (14). As above, we note that $\ell \leq an/2$ with probability no more than $\exp(-a^2 n/8)$. Note that

$$\#_0(X) - \#_1(X) \geq \#_0(\widehat{Z}) - \#_1(\widehat{Z}) - 2\Delta ,$$

where $\widehat{Z} \triangleq Z_1 \ldots Z_\ell$. Observe, however, that the probability that *any* prefix $Z^{(t)} = Z_1 \ldots Z_t$, where $an/2 \leq t \leq n$, has $\#_0(Z^{(t)}) - \#_1(Z^{(t)}) \leq (2[(1+\epsilon)/2] - 1)an/4 = \epsilon an/4$ is no more than

$$n \cdot \exp(-\epsilon^2 an/64)$$

by (11). (This follows by taking the union bound over each of the individual $n - an/2 \leq n$ bad events.) Finally, taking the union bound over these two bad events yields (14). □

**Divergence and forkability of $\rho_\Delta(w)$.** We record a theorem of Russell et al. [29] which bounds the probability that random variables satisfying the $(1+\epsilon)/2$-martingale conditions are forkable.

**Theorem 4 (implicit in [29]).** *Let $X_1, \ldots, X_n$ be random variables taking values in $\{0,1\}$ that satisfy the $(1+\epsilon)/2$-martingale conditions. Then*

$$\Pr[X_1 \cdots X_n \textit{ is forkable}] \leq \exp\left(-\frac{2\epsilon^4 n}{1+35\epsilon}\right) \leq \exp\left(-\frac{\epsilon^4 n}{18}\right) .$$

*Note that the constant $1/18$ is quite loose when $\epsilon$ is small; in particular, the bound is $\exp(-2\epsilon^4(1-O(\epsilon))n)$.*

In fact, the original presentation [29] stated the result for binomially distributed variables, but the proof appearing there proceeds via a martingale analysis which can be immediately adapted to our setting where the $X_i$ are themselves a super-binomial martingale (e.g., satisfy the $(1+\epsilon)/2$-martingale conditions).

We record, also, the fundamental relationship between forkable strings and divergence, established by Kiayias et al. [22].

**Theorem 5 ([22]).** *Let $w \in \{0,1\}^*$. Then there is forkable substring $\check{w}$ of $w$ with $|\check{w}| \geq \mathrm{div}_0(w)$.*

Finally, we combine these results to control $\mathrm{div}_\Delta(W)$ for a string $W$ satisfying the $(f;\gamma)$-characteristic conditions.

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

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

*Proof.* Defining $X = \rho_\Delta(W)$, we have

$$\mathrm{div}_\Delta(W) \overset{(a)}{\leq} \mathrm{div}_0(X) \overset{(b)}{\leq} \mathrm{div}_0(X^{\lceil \Delta}) + \Delta \overset{(c)}{\leq} \mathrm{div}(Z_1 \ldots Z_R) + \Delta\,, \tag{15}$$

where $Z_1, Z_2, \ldots$ are the random variables satisfying the $\gamma(1-f)^{\Delta-1}$-martingale conditions promised by Lemma 8. Above, inequality $(a)$ follows from Lemma 4, inequality $(b)$ from the fact that divergence satisfies the growth bound

$$\mathrm{div}_0(xy) \leq \mathrm{div}_0(x) + |y|\,, \tag{16}$$

and inequality $(c)$ follows from Lemma 8(ii) and (16). By Theorem 5, when $\mathrm{div}_0(Z) \geq k$ there is a forkable substring of $Z$ of length at least $k$; then summing the bounds provided by Theorem 4 over all lengths at least $k$ we find that the probability of such a substring beginning at a particular fixed index is no more than

$$\begin{aligned}
\sum_{t=k}^{\infty} \exp\left(-\frac{\epsilon^4 t}{18}\right) &= \exp\left(-\frac{\epsilon^4 k}{18}\right) \sum_{t=0}^{\infty} \exp\left(-\frac{\epsilon^4 t}{18}\right) = \exp\left(-\frac{\epsilon^4 k}{18}\right)\left(\frac{1}{1-\exp(-\epsilon^4/18)}\right) \\
&\leq \exp\left(-\frac{\epsilon^4 k}{18}\right)\left(\frac{1}{1-(1-\epsilon^4/18+(\epsilon^4/18)^2/2)}\right) \\
&\leq \exp\left(-\frac{\epsilon^4 k}{18}\right)\left(\frac{18}{\epsilon^4}\right)\left(\frac{1}{1-(\epsilon^4/36))}\right) \\
&\leq \exp\left(-\frac{\epsilon^4 k}{18}\right)\left(\frac{18}{\epsilon^4}\right)\left(\frac{36}{35}\right) \\
&\leq \frac{19}{\epsilon^4} \exp\left(-\frac{\epsilon^4 k}{18}\right)\,.
\end{aligned}$$

As there are no more than $R$ indices where such a forkable string could begin, we conclude that

$$\Pr[\mathrm{div}_0(Z_1 \ldots Z_R) \geq k] \leq \frac{19R}{\epsilon^4} \exp\left(-\frac{\epsilon^4 k}{18}\right)\,.$$

Combining this with (15), the statement of the theorem follows immediately. ◻

### E.4 Distribution of Characteristic Strings in a Single Epoch

We now consider an execution of Ouroboros-Praos over a single epoch consisting of $R$ slots in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration (as in Theorem 1). We assume that the randomness used for slot leader selection throughout this epoch is perfect (i.e., unbiased by the adversary) and known to all participating stakeholders (just as in the first epoch, where it is a part of the genesis block $\mathbf{G}$ provided by $\mathcal{F}_{\mathsf{INIT}}$). In what follows, we refer to this as the *single-epoch* setting.

Recall that within a single epoch, the stake distribution used for electing slot leaders is fixed. Nonetheless, there are still several adaptive aspects of the experiment: the adversary is allowed to adaptively corrupt stakeholders (so the amount of corrupted stake may adaptively increase during an epoch); and the environment can adaptively make parties stalled or time-unaware by deregistering them either from $\mathcal{G}_{\text{RO}}$ or $\mathcal{G}_{\text{CLOCK}}$ (and of course, register them back).

As determined by the Ouroboros-Praos protocol, a party with relative stake $\alpha \in [0, 1]$ becomes a slot leader for a given slot with probability

$$\phi_f(\alpha) = 1 - (1-f)^\alpha\,.$$

We recall the motivation (from [14]) for this non-linear stake scaling convention for leader selection: the function $\phi_f$ satisfies the "independent aggregation" property:

$$1 - \phi\left(\sum_i \alpha_i\right) = \prod_i (1 - \phi(\alpha_i))\,. \tag{17}$$

In particular, when leadership is determined according to $\phi_f$, 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. In particular, consider a party $U$ with relative stake $\alpha$ who contrives to split its stake among two virtual subordinate parties with stakes $\alpha_1$ and $\alpha_2$ (so that $\alpha_1 + \alpha_2 = \alpha$). Then the probability that one of these virtual parties is elected for a particular slot is $1-(1-\phi(\alpha_1))(1-\phi(\alpha_2))$, as these events are independent. Property (17) guarantees that this is identical to $\phi(\alpha)$. *Thus this selection rule is invariant under arbitrary reapportionment of a party's stake among virtual parties.* We record some further elementary properties of this convention.

**Proposition 1.** *The function $\phi_f(\alpha)$ satisfies the following properties.*

$$\phi_f\left(\sum_i \alpha_i\right) = 1 - \prod_i (1 - \phi_f(\alpha_i)) \leq \sum_i \phi_f(\alpha_i)\,, \qquad \textit{for any } \alpha_i \geq 0\,, \tag{18}$$

$$\alpha f \leq \phi_f(\alpha) \leq \alpha(-\ln(1-f)) = \alpha\left(f + \frac{f^2}{2} + \frac{f^3}{3} + \dots\right)\,, \qquad \textit{for any } \alpha \in [0,1]\,. \tag{19}$$

*Proof.* These inequalities are discussed and proven in [14] with the exception of the bound

$$\phi_f(\alpha) \leq \alpha(-\ln(1-f))\,.$$

This follows because

$$\frac{d\phi_f}{d\alpha}(0) = -\ln(1-f) \quad \text{and} \quad \frac{d^2\phi_f}{d\alpha^2}(\alpha) = -(1-f)^\alpha\big(\ln(1-f)\big)^2\,.$$

As the second derivative is everywhere negative, the linear approximation via the first derivative at zero is an upper bound. ◻

Our adversarial stake assumptions yield a characteristic string distribution $W_1, \dots, W_R$ governed by the (evolving) stake of the alert and active participants during each slot. In preparation for a detailed description, recall the definitions of quantities $\mathcal{S}^+(\cdot)$ and $\mathcal{S}^-(\cdot)$ given in Definition 1 (which we apply below to individual parties as well as party classes) and the notions of alert and participating stake ratios as per Definition 2.

**Lemma 9.** *The protocol* Ouroboros-Praos*, when executed in the single-epoch setting, induces characteristic strings $W_1, \dots, W_R$ (with each $W_t \in \{0, 1, \perp\}$) satisfying*

$$(1-f) \leq \Pr[W_t = \perp \mid W_1, \dots, W_{t-1}] \leq \prod_{U \in \mathcal{P}_{active}[t]} (1-f)^{\mathcal{S}^-}(U) = 1 - \phi_f(\mathcal{S}^-(\mathcal{P}_{active}[t]))\,,$$

*where $\mathcal{P}_{active}[t]$ denotes the set of active participants at time $t$. Furthermore,*

$$\Pr[W_t = 0 \mid W_1, \dots, W_{t-1}] \geq \phi_f(\mathcal{S}^-(\mathcal{P}_{alert}[t]))(1-f)^{\mathcal{S}^-(\mathcal{P}_{active}[t])} \geq \mathcal{S}^-(\mathcal{P}_{alert}[t]) f (1-f)\,,$$
$$\Pr[W_t \neq \perp \mid W_1, \dots, W_{t-1}] \leq \phi_f(\mathcal{S}^+(\mathcal{P}_{active}[t])) \leq \mathcal{S}^+(\mathcal{P}_{active}[t])(-\ln(1-f))\,,$$

*where $\mathcal{P}_{alert}[t]$ denotes the set of alert participants at time $t$.*

*Proof.* This follows from the definition of characteristic string, $\phi_f(\cdot)$ and the properties (18) and (19). ◻

Then it follows immediately that these random variables satisfy the characteristic conditions.

**Corollary 2.** *The protocol Ouroboros-Praos, when executed in the single-epoch setting, induces characteristic strings $W_1, \ldots, W_R$ (with each $W_t \in \{0, 1, \perp\}$) satisfying the $(f; c_f \cdot (1-f)\alpha)$-characteristic conditions, where $\alpha$ is a lower-bound on the alert stake ratio over the execution and*

$$c_f = \frac{f}{-\ln(1-f)}\,.$$

*Furthermore, as noted above, $\Pr[W_t = \perp \mid W_1, \ldots, W_{t-1}] \le 1 - \phi_f(\mathcal{S}^-(\mathcal{P}_{active}[t]))$.*

*For convenience, we note a weaker, but simpler, conclusion: the $W_1, \ldots, W_R$ satisfy the $(f; (1-f)^2\alpha)$-characteristic conditions and, additionally,*

$$\Pr[W_t = \perp \mid W_1, \ldots, W_{t-1}] \le 1 - f \cdot \mathcal{S}^-(\mathcal{P}_{active}[t])\,. \tag{20}$$

*Proof.* The first statement follows directly from Lemma 9 and Definition 14. The weaker conclusion follows from the first one, as we have

$$c_f = \frac{f}{-\ln(1-f)} = \frac{f}{f + f^2/2 + f^3/3 + \cdots} \ge \frac{1}{1 + f + f^2 + \cdots} = 1 - f\,,$$

and $\phi_f(a) \ge fa$. (We remark that the inequality $c_f \ge (1-f)(2+f)/2$ is an alternative polynomial approximation, somewhat more cumbersome than the bound above, which is tight to first order at $f \approx 0$.) $\square$

### E.5 Common Prefix, Chain Growth, and Chain Quality for a Single Epoch

**Corollary 3 (Common prefix).** *Let $W = W_1, \ldots, W_r$ denote the characteristic string induced by the Ouroboros-Praos protocol in the single-epoch setting over a sequence of $r$ slots. Assume that $\epsilon > 0$ satisfies*

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

*where $\alpha$ is a lower-bound on the alert stake ratio over the execution. Then*

$$\Pr[\mathrm{div}_\Delta(W) \ge k + \Delta] \le \frac{19r}{\epsilon^4}\exp(-\epsilon^4 k/18)\,,$$

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

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

*Proof.* The statement is a direct consequence of combining Theorem 6 with Corollary 2. $\square$

Following [17, 14], for a fixed characteristic string $w = w_1, \ldots, w_r$ we say that an index (or slot) $i \in [1, r - \Delta + 1]$ is *$\Delta$-right-isolated* if $w_i = 0$ and $w_{i+1} = w_{i+2} = \cdots = w_{i+\Delta-1} = \perp$.

In preparation for establishing chain growth and chain quality, we describe two further chain properties that will be instrumental in the arguments.

- **Honest-Bounded Chain Growth (HCG); with parameters $\tau \in (0, 1], s \in \mathbb{N}$.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a slot $\mathtt{sl}$. Let $\mathtt{sl}_1$ and $\mathtt{sl}_2$ be two previous slots for which $\mathtt{sl}_1 + s \le \mathtt{sl}_2 \le \mathtt{sl}$ and both $\mathcal{C}[\mathtt{sl}_1]$ and $\mathcal{C}[\mathtt{sl}_2]$ are honest blocks. Then $|\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]| \ge \tau \cdot s$.
- **Honest-Bounded Chain Quality (HCQ); with parameters $\tau \in (0, 1], s \in \mathbb{N}$.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a slot $\mathtt{sl}$. Let $\mathtt{sl}_1$ and $\mathtt{sl}_2$ be two previous slots for which $\mathtt{sl}_1 + s \le \mathtt{sl}_2 \le \mathtt{sl}$ and both $\mathcal{C}[\mathtt{sl}_1]$ and $\mathcal{C}[\mathtt{sl}_2]$ are honest blocks. Then $\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]$ must contain at least $\tau \cdot s$ honestly generated blocks.

Note that $\mathsf{HCQ}$ clearly implies $\mathsf{HCG}$ with the same parameters; however, looking ahead, we will establish stronger bounds for $\mathsf{HCG}$. These properties can be combined with existential chain quality ($\exists\mathsf{CQ}$, defined in Section 4.1) to establish chain growth ($\mathsf{CG}$) and chain quality ($\mathsf{CQ}$), as described in the lemma below.

**Lemma 10.** *Consider an execution of* Ouroboros-Praos *that satisfies* $\exists\mathsf{CQ}$ *with parameter* $s_{\exists\mathsf{CQ}}$*. Then the following hold:*

1. *If the execution satisfies* $\mathsf{HCG}$ *with parameters* $\tau_{\mathsf{HCG}}$ *and* $s_{\mathsf{HCG}}$*, then it satisfies* $\mathsf{CG}$ *with parameters*

$$s = 2s_{\exists\mathsf{CQ}} + s_{\mathsf{HCG}} \qquad \text{and} \qquad \tau = \tau_{\mathsf{HCG}} \cdot \left(\frac{s_{\mathsf{HCG}}}{s_{\mathsf{HCG}} + 2s_{\exists\mathsf{CQ}}}\right) .$$

*In particular, assuming* $s_{\mathsf{HCG}} \geq 2s_{\exists\mathsf{CQ}}$*, the execution satisfies* $\mathsf{CG}$ *with parameter* $\tau \geq \tau_{\mathsf{HCG}}/2$*.*

2. *If the execution satisfies* $\mathsf{HCQ}$ *with parameters* $\tau_{\mathsf{HCQ}}$ *and* $s_{\mathsf{HCQ}}$*, then it satisfies* $\mathsf{CQ}$ *with parameters*

$$k = 2s_{\exists\mathsf{CQ}} + s_{\mathsf{HCQ}} \qquad \text{and} \qquad \mu = \tau_{\mathsf{HCQ}} \cdot \left(\frac{s_{\mathsf{HCQ}}}{s_{\mathsf{HCQ}} + 2s_{\exists\mathsf{CQ}}}\right) .$$

*In particular, assuming* $s_{\mathsf{HCQ}} \geq 2s_{\exists\mathsf{CQ}}$*, the execution satisfies* $\mathsf{CQ}$ *with parameter* $\mu = \tau_{\mathsf{HCQ}}/2$*.*

*Proof.* Regarding the first statement of the lemma, consider a portion of a chain $\mathcal{C}$ held by an alert party spanning $\widehat{s} \geq s = 2s_{\exists\mathsf{CQ}} + s_{\mathsf{HCG}}$ slots. By $\exists\mathsf{CQ}$, there must be an honest block associated with the first $s_{\exists\mathsf{CQ}}$ and last $s_{\exists\mathsf{CQ}}$ slots. Between these two honest blocks, which are separated by at least $s_{\mathsf{HCG}}$ slots, $\mathsf{HCG}$ guarantees that at least

$$\tau_{\mathsf{HCG}} \cdot (\widehat{s} - 2s_{\exists\mathsf{CQ}}) = \tau_{\mathsf{HCG}} \cdot \underbrace{\left(\frac{\widehat{s} - 2s_{\exists\mathsf{CQ}}}{\widehat{s}}\right)}_{(\dagger)} \widehat{s} \geq \tau_{\mathsf{HCG}} \cdot \left(\frac{s_{\mathsf{HCG}}}{s_{\mathsf{HCG}} + 2s_{\exists\mathsf{CQ}}}\right) \widehat{s}$$

blocks appear. (The last inequality follows because the function $f_\lambda(x) = (x - \lambda)/x$, for any $\lambda > 0$, is strictly increasing for $x > 0$—thus ($\dagger$) is minimized when $\widehat{s} = s_{\mathsf{HCG}} + 2s_{\exists\mathsf{CQ}}$.) The statement of the lemma follows.

Likewise, for the second statement of the lemma, consider a portion of a chain $\mathcal{C}$ containing $\widehat{k} \geq k = 2s_{\exists\mathsf{CQ}} + s_{\mathsf{HCQ}}$ blocks; of course, this portion must span at least $\widehat{k}$ slots. Applying $\exists\mathsf{CQ}$ to the $s_{\exists\mathsf{CQ}}$ slots on either side of the interval (as above) and $\mathsf{HCQ}$ to the remaining $\widehat{k} - 2s_{\exists\mathsf{CQ}}$ slots, the chain $\mathcal{C}$ must contain at least

$$\tau_{\mathsf{HCQ}} \cdot (\widehat{k} - 2s_{\exists\mathsf{CQ}}) = \tau_{\mathsf{HCQ}} \cdot \left(\frac{\widehat{k} - 2s_{\exists\mathsf{CQ}}}{\widehat{k}}\right) \widehat{k} \geq \tau_{\mathsf{HCQ}} \cdot \left(\frac{s_{\mathsf{HCQ}}}{s_{\mathsf{HCG}} + 2s_{\exists\mathsf{CQ}}}\right) \widehat{k}$$

honestly-generated blocks. ◻

We now establish concrete bounds on $\mathsf{HCG}$, $\mathsf{HCQ}$, and $\exists\mathsf{CQ}$ for Ouroboros-Praos in the single-epoch setting.

**Lemma 11.** *Let* $W = W_1, \ldots, W_r$ *denote the characteristic string induced by the protocol* Ouroboros-Praos *in the single-epoch setting over a sequence of* $r$ *slots. Let* $\alpha, \beta \in [0, 1]$ *denote lower bounds on the alert stake ratio and the participating stake ratio over the execution as per Definition 2, and assume that for some* $\varepsilon \in (0, 1)$ *the parameter* $\alpha$ *satisfies*

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

*Then* $\mathsf{HCG}$*,* $\mathsf{HCQ}$*, and* $\exists\mathsf{CQ}$ *are guaranteed with the following parameters:*

$\mathsf{HCG}$**:** *For* $s \geq 8\Delta/(\beta f)$ *and* $\tau = \beta f/8$*,*

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

$\mathsf{HCQ}$**:** *For* $s \geq 16\Delta/(\epsilon\beta f)$ *and* $\tau = \epsilon\beta f/8$*,*

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

$\exists$CQ: *For $s \geq 12\Delta/(\epsilon\beta f)$,*

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

*Proof.* For convenience, let us call a slot *good* if it is $\Delta$-right-isolated uniquely alert, and *bad* if it is neither empty nor good. We extend this terminology to blocks by calling a block good (resp. bad) if it is associated with a good (resp. bad) slot. For the discussion of honest-bounded properties below, consider a chain $\mathcal{C}$ held by an alert party at slot $\mathtt{sl}$ and two prior slots $\mathtt{sl}_1$ and $\mathtt{sl}_2$ for which (i.) $\mathtt{sl}$, $\mathtt{sl}_1$ and $\mathtt{sl}_2$ belong to the sequence of $r$ slots inducing $W$; (ii.) both $\mathcal{C}[\mathtt{sl}_1]$ and $\mathcal{C}[\mathtt{sl}_2]$ are honestly generated blocks, and (iii.) $\mathtt{sl}_1 + s \leq \mathtt{sl}_2 \leq \mathtt{sl}$. Let $T$ denote the interval

$$T \triangleq \{\mathtt{sl}_1 + 1, \ldots, \mathtt{sl}_2\}$$

and let $\widehat{\mathtt{sl}}_1, \ldots, \widehat{\mathtt{sl}}_g$ be the increasing sequence of all good slots in $T$ (here the notion of isolation refers to this block of slots: in particular, a good slot must be at least $\Delta$ slots from the right end of $T$). Let $V$ denote the portion of $W$ associated with the slots in $T$ and let $X = \rho_\Delta(V)$. Note that the good (resp., bad) slots appear as 0 (resp., 1) symbols in $X$, and hence $g = \#_0(X)$. Let also $b \triangleq \#_1(X)$ denote the number of bad slots of $T$.

**HCG**: Recall that honest-bounded chain growth demands that $|\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]| \geq \tau s$. To argue this, first observe that the uniquely alert slot leader associated with $\widehat{\mathtt{sl}}_2$ will consider the chain $\mathcal{C}[0 : \mathtt{sl}_1]$ in the chain selection rule, as $\mathcal{C}[0 : \mathtt{sl}_1]$ was diffused by a slot leader in $\mathtt{sl}_1$ and $\widehat{\mathtt{sl}}_2 \geq \widehat{\mathtt{sl}}_1 + \Delta \geq \mathtt{sl}_1 + \Delta$. In particular, the chain diffused by the unique slot leader in $\widehat{\mathtt{sl}}_2$ (after block addition) must have length at least $|\mathcal{C}[0 : \mathtt{sl}_1]| + 1$. By the same argument, the chains diffused by the uniquely alert players associated with $\widehat{\mathtt{sl}}_2, \ldots, \widehat{\mathtt{sl}}_g$ must grow monotonically: specifically, the chain diffused by the leader at slot $\widehat{\mathtt{sl}}_g$ must have length at least $|\mathcal{C}[0 : \mathtt{sl}_1]| + (g-1)$. Finally, note that the player generating the (honest) block $\mathcal{C}[\mathtt{sl}_2]$ will have received the chain diffused by the leader of $\widehat{\mathtt{sl}}_g$. We conclude that

$$|\mathcal{C}[0 : \mathtt{sl}_2]| \geq |\mathcal{C}[0 : \mathtt{sl}_1]| + g = |\mathcal{C}[0 : \mathtt{sl}_1]| + \#_0(X) .$$

Observe now that for $\tau = (1+\epsilon)\beta f/4 - \Delta/s$,

$$\begin{aligned}
\Pr[W \text{ admits } (\tau, s)\text{-}\mathsf{HCG} \text{ violation for } (\mathtt{sl}_1, \mathtt{sl}_2)] &\leq \Pr[\#_0(X) \leq \tau s] \\
&= \Pr[\#_0(X) \leq \beta f(1+\epsilon)s/4 - \Delta] \\
&\leq 2\exp\left(-(f\beta)^2 s/64\right) ,
\end{aligned}$$

where the last inequality follows from (13) by using $n := s$ and $a := \beta f$, the latter being justified by (20). By the union bound, applied over all pairs of slots, we conclude that

$$\Pr[W \text{ admits a } (\tau, s)\text{-}\mathsf{HCG} \text{ violation}] \leq 2r^2 \exp(-(f\beta)^2 s/64) .$$

The simpler bound appearing in the theorem statement can be obtained by assuming that $s \geq 8\Delta/(\beta f)$ and taking $\tau' = \beta f/8$. Then any $(s, \tau')$-HCG violation is a $(s, \tau)$-HCG violation, as $\tau' < \tau$ for such $s$.

**HCQ.** Recall that honest-bounded chain quality demands that $\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]$ contains at least $\tau s$ honestly generated blocks. Note that, as argued above, $|\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]| \geq g$. On the other hand, the total number of adversarially-generated blocks in $\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]$ can be no more than $b$. It follows that at least $g - b$ blocks in $\mathcal{C}[\mathtt{sl}_1 + 1 : \mathtt{sl}_2]$ are honest. Observe then that for $\tau = \epsilon\beta f/4 - 2\Delta/s$,

$$\begin{aligned}
\Pr[W \text{ admits a } (\tau, s)\text{-}\mathsf{HCQ} \text{ violation for } (\mathtt{sl}_1, \mathtt{sl}_2)] &\leq \Pr[\#_0(X) - \#_1(X) \leq \tau s] \\
&= \Pr[\#_0(X) - \#_1(X) \leq \epsilon\beta f s/4 - 2\Delta] \\
&\leq (s+1)\exp\left(-(\epsilon f\beta)^2 s/64\right) ,
\end{aligned}$$

where the last inequality follows from (14). Applying the union bound over all pairs of slots yields

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

The simpler bound appearing in the theorem statement can be obtained by assuming $s \geq 16\Delta/(\epsilon\beta f)$ and taking $\tau' = \epsilon\beta f/8$. Then any $(s, \tau')$-$\mathsf{HCQ}$ violation is a $(s, \tau)$-violation, as $\tau < \tau'$ for such $s$.

**$\exists$CQ.** We now consider the probability of an $s$-$\exists\mathsf{CQ}$-violation. Recall that an $s$-$\exists\mathsf{CQ}$ violation is described by a chain $\mathcal{C}$, eventually held by an alert party, and a pair of slots $\mathtt{sl}_1 < \mathtt{sl}_2$ for which $\mathtt{sl}_1 + s \leq \mathtt{sl}_2$ and $\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]$ contains no honestly generated blocks. Note that in this setting we no longer assume that $\mathcal{C}[\mathtt{sl}_1]$ and $\mathcal{C}[\mathtt{sl}_2]$ are honest.

First, observe that all blocks in $\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]$ are bad (as they are not even honestly generated). Let $G_1$ denote the latest honestly-generated block in $\mathcal{C}[0 : \mathtt{sl}_1 - 1]$ (note that at least $\mathcal{C}[0]$ is considered honest) and let $\overline{\mathtt{sl}}_1$ denote the slot associated with $G_1$; likewise, let $G_2$ denote the earliest honestly-generated block appearing in $\mathcal{C}[\mathtt{sl}_2 + 1 : \mathtt{sl}]$ (or the last block of $\mathcal{C}$, if there is no honest one) and let $\overline{\mathtt{sl}}_2$ denote the slot associated with $G_2$. Note that all blocks between $G_1$ and $G_2$ are bad.

Denote by $S$ the continuous sequence of slots

$$S = \{\overline{\mathtt{sl}}_1 + 1, \ldots, \overline{\mathtt{sl}}_2\}\,.$$

If $G_2 = \mathcal{C}[\overline{\mathtt{sl}}_2]$ is honest, note that by the same argument as above $|\mathcal{C}[0 : \overline{\mathtt{sl}}_2]| \geq |\mathcal{C}[0 : \overline{\mathtt{sl}}_1]| + g'$, where $g'$ is the number of good slots in $S$. However, in chain $\mathcal{C}$ we have $|\mathcal{C}[0 : \overline{\mathtt{sl}}_2]| \leq |\mathcal{C}[0 : \overline{\mathtt{sl}}_1]| + b' + 1$, where $b'$ is the number of bad slots in the same sequence $S$, since by assumption $\mathcal{C}[\overline{\mathtt{sl}}_1 + 1 : \overline{\mathtt{sl}}_2 - 1]$ contains no honestly generated blocks. These two conditions can only be satisfied at the same time if $g' \leq b' + 1$. On the other hand, if $G_2$ was not honestly-generated, we can only conclude that $|\mathcal{C}[0 : \overline{\mathtt{sl}}_2]| \geq |\mathcal{C}[0 : \overline{\mathtt{sl}}_1]| + g' - 1$; specifically, note that $\mathcal{C}$ has been adopted by an alert player at slot $\mathtt{sl}$, and so must have length at least that of the chain diffused during the last good slot of $S$. However, in this case we have $|\mathcal{C}[0 : \overline{\mathtt{sl}}_2]| \leq |\mathcal{C}[0 : \overline{\mathtt{sl}}_1]| + b'$, where $b'$ is the number of bad slots in $S$, since $\mathcal{C}[\overline{\mathtt{sl}}_1 + 1 : \overline{\mathtt{sl}}_2]$ contains no honestly generated blocks. Again we find that $g' \leq b' + 1$.

Observe that good slots are associated with 0s in the string $X' = \rho_\Delta(V')$, where $V'$ is the portion of $W$ associated with the interval $S$; likewise, bad slots are associated with 1s in this sequence. Specifically,

$$\Pr[W \text{ admits an } s\text{-}\exists\mathsf{CQ} \text{ violation for } (\mathtt{sl}_1, \mathtt{sl}_2)] \leq \Pr[\#_0(X') - \#_1(X') \leq 1]\,.$$

For $s \geq 12\Delta/(\epsilon\beta f)$, $\epsilon s(\beta f)/4 \geq 2\Delta + 1$ and hence, by (14),

$$\begin{aligned}
\Pr[W \text{ admits an } s\text{-}\exists\mathsf{CQ} \text{ violation for } (\mathtt{sl}_1, \mathtt{sl}_2)] &\leq \Pr[\#_0(X') - \#_1(X') \leq 1] \\
&\leq \Pr[\#_0(X') - \#_1(X') \leq \epsilon s(\beta f)/4 - 2\Delta] \\
&\leq (s+1)\exp\left(-(\epsilon\beta f)^2 s/64\right)\,.
\end{aligned}$$

The union bound, applied over all pairs of slots, then yields

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

$\square$

**Corollary 4 (Chain Growth).** *Let $W = W_1, \ldots, W_r$ denote the characteristic string induced by the protocol Ouroboros-Praos in the single-epoch setting over a sequence of $r$ slots. Let $\alpha, \beta \in [0,1]$ denote lower bounds on the alert stake ratio and the participating stake ratio over the execution as per Definition 2, and assume that for some some $\varepsilon \in (0,1)$ the parameter $\alpha$ satisfies*

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

*Then for*

$$s = 48\Delta/(\epsilon\beta f) \qquad \text{and} \qquad \tau = \beta f/16 \tag{21}$$

*we have*

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

*Proof.* The corollary follows directly by combining Lemmas 10 and 11, using $s_{\exists\mathsf{CQ}} = 12\Delta/(\epsilon\beta f)$, $s_{\mathsf{HCG}} = 2s_{\mathsf{ECQ}}$, and $\tau_{\mathsf{HCG}} = \beta f/8$. ◻

**Corollary 5 (Chain Quality).** *Let $W = W_1, \ldots, W_r$ denote the characteristic string induced by the protocol Ouroboros-Praos in the single-epoch setting over a sequence of $r$ slots. Let $\alpha, \beta \in [0,1]$ denote lower bounds on the alert stake ratio and the participating stake ratio as per Definition 2, and assume that for some some $\varepsilon \in (0,1)$ the parameter $\alpha$ satisfies*

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

*Then for*

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

*we have*

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

*Proof.* The corollary follows directly by combining Lemmas 10 and 11, using $s_{\exists\mathsf{CQ}} = 12\Delta/(\epsilon\beta f)$, $s_{\mathsf{HCQ}} = 2s_{\exists\mathsf{CQ}}$, and $\tau_{\mathsf{HCQ}} = \epsilon\beta f/8$. ◻

### E.6 Lifting to Multiple Epochs

The above analysis gives bounds for common prefix, chain growth, and variants of chain quality (denoted $\bar{\epsilon}_{\mathsf{CP}}$, $\bar{\epsilon}_{\mathsf{CG}}$, $\bar{\epsilon}_{\mathsf{CQ}}$, and $\bar{\epsilon}_{\exists\mathsf{CQ}}$, respectively) for a single-epoch run of the protocol with static stake distribution and perfect randomness. We now conclude our proof of Theorem 1 by showing conditions under which these blockchain properties hold throughout the whole lifetime of the system consisting of many epochs.

**Theorem 7.** *Consider the execution of Ouroboros-Praos with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration. Let $f$ be the active-slot coefficient, let $\Delta$ be the upper bound on the network delay. Let $\alpha, \beta \in [0,1]$ denote a lower bound on the alert and participating stake ratios throughout the whole execution, respectively. Let $R$ and $L$ denote the epoch length and the total lifetime of the system (in slots), and let $Q$ be the total number of queries issued to $\mathcal{G}_{\text{RO}}$. If for some $\varepsilon \in (0,1)$ we have*

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

*then Ouroboros-Praos achieves the same guarantees for common prefix (resp. chain growth, chain quality, existential chain quality) as given in Corollary 3 (resp. Corollary 4, Corollary 5, Lemma 11) except with an additional error probability of*

$$QL \cdot \left(2\bar{\epsilon}_{\mathsf{CG}}\left(\tau, R/3; R, \epsilon\right) + 2\bar{\epsilon}_{\mathsf{CP}}\left(\tau R/3; R, \Delta, \epsilon\right) + \bar{\epsilon}_{\exists\mathsf{CQ}}\left(R/3; R, \epsilon\right)\right)\,, \tag{22}$$

*where $\tau = \beta f/16$. If $R \geq 144\Delta/\epsilon\beta f$ then this term can be upper-bounded by*

$$\epsilon_{\mathsf{lift}} \triangleq QL \cdot \left[R^3 \cdot \exp\left(-\frac{(\epsilon\beta f)^2 R}{768}\right) + \frac{38R}{\epsilon^4} \cdot \exp\left(\Delta - \frac{\epsilon^4\tau R}{54}\right)\right]\,. \tag{23}$$

*Proof (sketch).* This part of the analysis proceeds similarly as in Section 5 of [14] and hence we only sketch it. When moving from the single-epoch setting to a setting with several epochs, two new aspects need to be considered:

- **Stake distribution updates.** The stake distribution used for sampling slot leaders changes in every epoch (this is why we consider epochs in the first place). In Ouroboros-Praos (and Ouroboros-Genesis), the distribution used for sampling in epoch `ep` is set to be the stake distribution recorded on the blockchain up to the last block of the epoch `ep` − 2.

– **Randomness updates.** Every epoch needs new public randomness to be used for sampling slot leaders from the above distribution. For epoch $\mathtt{ep}$, this randomness is obtained by hashing together VRF-outputs put into blocks in epoch $\mathtt{ep}-1$ by their creators. More precisely, the protocol hashes together these values from the blocks in the first $2R/3$ slots of epoch $\mathtt{ep}-1$ (out of its $R$ slots).

To argue that the above process of updating stake distribution and public randomness does not noticeably deviate the execution from the single-epoch analysis, we rely on the single-epoch setting bounds proven above. In particular, we make the following three observations:

– Chain growth and common prefix imply that during the first $R/3$ slots of each epoch, each alert player's chain grows by at least $\tau R/3$ blocks (for $\tau$ as in (21)) and therefore after these slots, all alert players agree on the stake distribution at the end of the previous epoch except with probability

$$\bar{\epsilon}_{\mathsf{CG}}(\tau, R/3; R, \epsilon) + \bar{\epsilon}_{\mathsf{CP}}(\tau R/3; R, \Delta, \epsilon) \ .$$

– Existential chain quality implies that during the second $R/3$ slots of each epoch, each alert player's chain contains at least one honest block except with probability

$$\bar{\epsilon}_{\exists\mathsf{CQ}}(R/3; R, \epsilon) \ .$$

That implies that the randomness that will be derived for the next epoch will be influenced by at least one honest VRF-output determined *after* the stake distribution is fixed.

– Chain growth and common prefix imply that during the last $R/3$ slots of each epoch, each alert player's chain grows by at least $\tau R/3$ blocks and therefore after these slots, all alert players agree on the randomness for the next epoch except with probability

$$\bar{\epsilon}_{\mathsf{CG}}(\tau, R/3; R, \epsilon) + \bar{\epsilon}_{\mathsf{CP}}(\tau R/3; R, \Delta, \epsilon) \ .$$

Hence, if we assumed perfect randomness in each epoch, all the above desired properties would be satisfied throughout the lifetime of the system $L$ except with probability

$$L \cdot (2\bar{\epsilon}_{\mathsf{CG}}\left(\tau, R/3; R, \epsilon\right) + 2\bar{\epsilon}_{\mathsf{CP}}\left(\tau R/3; R, \Delta, \epsilon\right) + \bar{\epsilon}_{\exists\mathsf{CQ}}\left(R/3; R, \epsilon\right))$$

by union bound.

However, the above properties are not sufficient to infer that the public randomness used for leader election in the next epoch will be perfect. Instead, the process of deriving it described above still allows a limited amount of grinding by the adversary, who can decide whether to include blocks (with VRF outputs) in slots where he is a slot leader. In [14], it is shown that this grinding effect can be crudely upper-bounded by limiting the number of queries to the random oracle that the adversary makes (of course, more fine-grained bounds are possible). The same argument applies here, and hence we need to introduce the quantity $Q$ into our bound (22). Since we model the random oracle as a global functionality $\mathcal{G}_{\mathrm{RO}}$, the quantity $Q$ is an upper bound on the *total* number of queries to $\mathcal{G}_{\mathrm{RO}}$ that were asked during the execution, including queries from the environment.

Finally, the bound (23 is obtained by instantiating (22) with the concrete bounds of Corollaries 3 and 4, and Lemma 11 (where Lemma 11 requires the assumption $R \geq 36\Delta/\epsilon\beta f$, while 4 requires a stricter bound $R \geq 144\Delta/\epsilon\beta f$). □

# F Large Deviation Bounds

We apply a variety of large deviation bounds in our probabilistic arguments, which we record here for concreteness. See, e.g., [25] for proofs and further discussion.

**Theorem 8 (Chernoff bound).** *Let $X_1, \ldots, X_T$ be independent random variables with $\mathbb{E}[X_i] = p_i$ and $X_i \in [0,1]$. Let $X = \sum_{i=1}^{T} X_i$ and $\mu = \sum_{i=1}^{T} p_i = \mathbb{E}[X]$. Then, for all $\Lambda \geq 0$,*

$$\Pr[X \geq (1+\Lambda)\mu] \leq e^{-\frac{\Lambda^2}{2+\Lambda}\mu}\,;$$
$$\Pr[X \leq (1-\Lambda)\mu] \leq e^{-\frac{\Lambda^2}{2+\Lambda}\mu}\,.$$

**Theorem 9 (Azuma's inequality (Azuma; Hoeffding). See [25, 4.16] for discussion).** *Let $X_0, \ldots, X_n$ be a sequence of real-valued random variables so that, for all $t$, $|X_{t+1} - X_t| \leq c$ for some constant $c$. If $\mathbb{E}[X_{t+1} \mid X_0, \ldots, X_t] \leq X_t$ for all $t$ then for every $\Lambda \geq 0$*

$$\Pr[X_n - X_0 \geq \Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right).$$

*Alternatively, if $\mathbb{E}[X_{t+1} \mid X_0, \ldots, X_t] \geq X_t$ for all $t$ then for every $\Lambda \geq 0$*

$$\Pr[X_n - X_0 \leq -\Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right).$$

## G List of Symbols

The communication model:

| | |
|---|---|
| $\Delta$ | maximum message delay in slots |

Functionalities:

| | |
|---|---|
| $\mathcal{G}_{\text{CLOCK}}$ | global clock |
| $\mathcal{G}_{\text{RO}}$ | global random oracle |
| $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc},\Delta}$ | $\Delta$-delayed network for diffusing blockchains |
| $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx},\Delta}$ | $\Delta$-delayed network for diffusing transactions |
| $\mathcal{F}_{\mathsf{INIT}}$ | init functionality providing the genesis block |
| $\mathcal{F}_{\mathsf{VRF}}$ | verifiable random function |
| $\mathcal{F}_{\mathsf{KES}}$ | key-evolving signature scheme |
| $\mathcal{G}_{\text{LEDGER}}$ | the ledger functionality |

Functionality $\mathcal{G}_{\text{LEDGER}}$:

| | |
|---|---|
| $\tau_L$ | current time |
| $\vec{\tau}_{\mathtt{state}}$ | sequence of time stamps of state blocks |
| $\vec{\mathcal{I}}_H^T$ | timed honest-input sequence |
| $\mathcal{S}_{\text{initStake}}$ | initial stakeholder set |

Protocol Ouroboros-Genesis:

| | |
|---|---|
| $f$ | active slots coefficient |
| $\phi(\cdot)$ | slot-leader probability function (Eq. (1)) |
| $R$ | epoch length in slots |
| $\mathbb{S}_{\mathsf{ep}}$ | stake distribution used to sample slot leaders in epoch $\mathtt{ep}$ |
| $\alpha_p^{\mathsf{ep}}$ | relative stake of party $U_p$ in $\mathbb{S}_{\mathsf{ep}}$ |
| $\eta_{\mathsf{ep}}$ | randomness used to sample slot leaders in epoch $\mathtt{ep}$ |

Analysis:

| | |
|---|---|
| $\alpha$ | alert stake ratio (Def. 2) |
| $\beta$ | participating stake ratio (Def. 2) |
| $L$ | total length of the execution (in slots) |
| $Q$ | total number of queries to the random oracle |