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

Christian Badertscher
ETH Zurich
badi@inf.ethz.ch

Peter Gaži
IOHK
peter.gazi@iohk.io

Aggelos Kiayias*
University of Edinburgh and IOHK
akiayias@inf.ed.ac.uk

Alexander Russell†
University of Connecticut and IOHK
acr@cse.uconn.edu

Vassilis Zikas
University of Edinburgh and IOHK
vzikas@inf.ed.ac.uk

ABSTRACT

We present a novel Proof-of-Stake (PoS) protocol, Ouroboros Genesis, that enables parties to safely join (or rejoin) the protocol execution using only the genesis block information. Prior to our work, PoS protocols either required parties to obtain a trusted "checkpoint" block upon joining and, furthermore, to be frequently online or required an accurate estimate of the number of online parties to be hardcoded into the protocol logic. This ability of new parties to "bootstrap from genesis" was a hallmark property of the Bitcoin blockchain and was considered an important advantage of PoW-based blockchains over PoS-based blockchains since it facilitates robust operation in a setting with *dynamic availability*, i.e., the natural setting—without external trusted objects such as checkpoint blocks—where parties come and go arbitrarily, may join at any moment, or remain offline for prolonged periods of time. We prove the security of Ouroboros Genesis against a fully adaptive adversary controlling less than half of the total stake in a partially synchronous network with unknown message delay and unknown, varying levels of party availability. Our security proof is in the Universally Composable setting assuming the most natural abstraction of a hash function, known as the *strict* Global Random Oracle (ACM-CCS 2014); this highlights an important advantage of PoS blockchains over their PoW counterparts in terms of composability with respect to the hash function formalisation: rather than a strict GRO, PoW-based protocol security requires a "local" random oracle. Finally, 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.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; *Cryptography*; *Formal security models*;

KEYWORDS

Distributed ledgers, Blockchain, Proof-of-Stake

ACM Reference Format:
Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2018. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243848

1 INTRODUCTION

The primary real-world use of blockchains, thus far, has been to offer a platform for decentralized cryptocurrencies with various capabilities [7, 32]. A unique feature of blockchain protocols (in contrast to, say, classical consensus protocols) is the fact that the parties running 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 desynchronise, join at any moment of the execution or go offline for arbitrary periods of time, as long as a majority of hashing power is always following the protocol. We refer to this desirable set of execution features as *dynamic availability*. 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 cryptographic protocols. For example, a number of recent works [1, 2, 6, 27, 28] 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—without the need of a trusted third party [15, 29]; or how to use blockchains as bulletin boards in electronic voting [30]. Such developments—in conjunction with the direct applicability to cryptocurrencies—have motivated general, formal security analysis of the functionality that blockchain protocols provide, undertaken in steps of successive refinement in [4, 18, 19, 33].

However, blockchain protocols such as Bitcoin and Ethereum have led to concerns regarding their extraordinary energy demands as they rely on *proof-of-work* (in short, *PoW*), a cryptographic puzzle-solving procedure that increases in difficulty as more parties

*Partially supported by H2020 Project PRIViLEDGE #780477.
†Partially supported by NSF grant CCF-1717432.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3243848

join the system.[1] The desire for more efficient blockchain solutions gave rise to an exciting recent line of work that proposes to use alternative resources to achieve consensus and maintain a robust ledger. A popular such resource is *stake* in the system itself [5, 13, 26]. 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, referred to as *proof-of-stake* (in short, *PoS*), has yielded a number of proposals for PoS-based blockchains, including several supported by formal analysis: Algorand [21], Snow White [14], and Ouroboros/Ouroboros Praos [16, 25], which is implemented as part of the Cardano blockchain.[2] Additionally, Ethereum—a noted PoW scheme—aims to transition to PoS in the future [8].

Despite the clear advantages of PoS blockchains in terms of energy efficiency, their suitability as PoW blockchain replacements has been uncertain; see, e.g., [35]. Notably, such protocols restrict the *dynamic availability* of participants: for instance, Casper [8], Snow White [14], and Ouroboros/Ouroboros Praos [16, 25] require parties to maintain a "moving checkpoint"—specifically, new parties must receive trusted "advice" when they join and otherwise be frequently online—while Algorand [21] requires that a good estimate of the number of online parties is hardcoded in the protocol; in case the estimate becomes incorrect, due to some parties going offline, the protocol will halt, even when an active honest majority is present (while protocols like [14, 16] will merely slow down).

All the above point to the following open, and fundamental, question: Is it possible for PoS-blockchains to provide the same functionality guarantees as PoW-blockchains in the setting of full dynamic availability without access to any information beyond a trusted initialisation string (the "genesis block")?

Our Contributions. We propose a new protocol which features a novel chain selection rule that enables joining parties to "bootstrap from genesis", i.e., to safely join an operating blockchain protocol without any information other than the genesis block. In particular, this provides the joining party a blockchain possessing all the favorable properties (e.g., a large common prefix with other honest parties) that would be guaranteed if the party had fully participated during the entire history of the protocol. We then prove that the protocol implements the natural ledger functionality proposed in [4]—the very same functionality shown to be possessed by Bitcoin—under the assumption of standard cryptographic primitives. Our formalisation captures explicitly the setting of dynamic availability: parties are allowed to join and leave the system at will, as well as lose their network and/or clock synchronisation. We describe these contributions in more detail below.

Our primary contribution is the construction and analysis of a new protocol, Ouroboros Genesis, which builds on a recent PoS protocol, Ouroboros Praos [16]; the main novelty of the new protocol is a (significantly) differing chain selection rule—instantiating the so-called maxvalid procedure in [4, 16, 18, 25]—which allows parties to identify a chain that shares a large common prefix with a recent honest chain, *using only knowledge of the genesis block*. The protocol is organized around *epochs*, periods of the execution during which the stake distribution is treated as constant by the protocol. We prove that Ouroboros Genesis realizes the ledger functionality originally proposed in [4] to model the Bitcoin protocol under an appropriate honest stake majority assumption: specifically, the majority of stake in each epoch—as defined by the stake distribution in a recent previous epoch—favors the honest parties in a precise sense that reflects the dynamic availability setting. In order to express this guarantee formally, we refine the model of [4] to include the following events: (i) parties may be spawned and join the network at any time, (ii) parties may lose or regain their network connection at any time, and (iii) parties may lose or regain access to the random oracle (that in [4] models the hashing operation of the party) and/or clock functionality at any time. The honest parties that retain their network connection, clock and random oracle access for a sufficient amount of time—linear in the network delay—are called *alert*, while the set of alert and adversarial parties are called *active*. On the other hand, the parties that have lost synchronisation with the clock or are not connected to the random oracle are called *stalled*; such parties still have an active network connection and when they regain access to the clock and the random oracle are guaranteed to receive all pending messages. The set of stalled parties is a refinement of the concept of sleepy parties that appeared in [14, 34].

Using this fine grained formalisation of party behavior, we can express explicitly the dynamic availability guarantees under which Ouroboros Genesis is shown secure: (i) The ratio α of the stake of *alert* parties to that of the *active* parties is above $1/2$; the difference is by a constant that is sufficiently large to appropriately absorb the partial synchrony delay parameter Δ. In particular (and similarly to the Bitcoin blockchain [18, 33]) the protocol will use a "active slots coefficient" f and provide meaningful security guarantees for a range of Δ below $1/f$. (ii) The ratio β of the stake of the *active* parties to the stake of *all* parties is bounded below by some arbitrary constant that is unknown to the protocol participants.

The constraints above are arguably optimal (that is, necessary) in the dynamic availability setting. First, if the alert to active stake ratio α drops below $1/2$, it indicates that (despite a possible honest majority), the *effective* participating honest parties form a stake minority once we account for the honest players that are either offline or have so recently joined that they are not yet fully synchronised with the rest of the honest parties (due to message delays). Second, if the active to total stake ratio β approaches 0, this indicates that while there might be honest stake majority, almost all honest parties are stalled and thus the protocol cannot support "liveness", i.e., the guaranteed processing of newly submitted transactions.

Our analysis is in the (partial) synchronous model of universally composable (UC) functionalities [9, 11, 22] that was also used in [4] to analyze the bitcoin protocol. A significant improvement in our analysis, which highlights the higher composability potential of PoS protocols over their PoW counterparts, is that we are able to prove security in the *strict* global random oracle (sGRO or simply GRO) setting [12], which is the most natural random oracle abstraction of hash functions; this contrasts with the hash function formalisation of [4] that restricts access to the hash function in each protocol session. This latter limitation appears to be an inherent barrier

[1] Currently each single Bitcoin block requires more than 2^{72} operations to be performed, cf. https://en.bitcoin.it/wiki/Difficulty.

[2] Cardano, https://whycardano.com, is currently the largest pure PoS cryptocurrency according to market capitalisation, cf. https://coinmarketcap.com.

of the PoW setting since universal composition of a PoW-based blockchain would require that computational effort expended in the environment is inherently independent of the current session, something known to be untrue (for instance, merged mining, cf. [23], exploits exactly the transfer of computational effort across protocol sessions).

On the necessity of novel techniques. The formal security arguments supporting Ouroboros Genesis—as with other rigorously analysed blockchain protocols—ultimately rely on definition and analysis of a stochastic process that abstracts the underlying dynamics of block creation and dissemination. The Ouroboros Genesis stochastic process is given by a coupled pair of discrete random walks, very like the underlying process associated with the Ouroboros Proas protocol. However, the Ouroboros Genesis analysis must contend with a significant new analytic challenge: the "steps" of the random walk are not simply given by independent random variables (as in previous analyses), but may be correlated by the adaptive behavior of the adversary. The stake constraints, fortunately, provide limits on the worst-case correlation of these random variables, and our analyses shows that this can be advantageously expressed as a martingale to which powerful classical concentration results apply. This permits us to dovetail our development with the "forkable strings" analysis of [36] which, fortunately, can also be cast in a martingale setting. These techniques may have independent interest, as they can apply to quite general blockchain dynamics and thus may prove useful for analyzing other PoS-based blockchains.

Related Work. A number of recent works have studied—in a rigorous cryptographic manner—the security of several blockchain protocols adopting both PoW-based (e.g, [4, 18, 33]) and PoS-based (e.g., [14, 16, 21, 25, 34]) consensus mechanisms. In the PoW-based setting, [4] 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 various proposed schemes tolerate different types of adversaries in terms of adaptivity. For example, Ouroboros [25] achieves only "semi-adaptive" security (corruptions with delay), whereas among the adaptively secure ones, Algorand [21] requires less than 1/3 of the stake of the system to be held by malicious parties; Show White [14] and Ouroboros Praos [16] 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 [34] where those parties were referred to as *sleepers*. Our modeling of such parties differs from that of [34] 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 only affects 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. 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, [14] 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-sleepers in [14] (i.e., those that are in sleepy mode for a prolonged time) that required a safe initialisation string in [14]. Taking advantage our bootstrapping from genesis chain selection rule, all parties that are stalling, even for prolonged periods of time, can safely resynchronise 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. 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 begins by considering the interaction of the old chain selection procedure from [16] (this rule is called maxvalid-mc here; the protocol using it is dubbed Ouroboros-Praos) with online and stalled parties only (Section 4.2); then the proof gradually incorporates the new maxvalid-bg procedure which allows the protocol to bootstrap from the genesis block (Section 4.3), and establishes that this procedure is sufficient to provide all appropriate guarantees to newly joining and temporarily offline parties (Section 4.4). Finally, the results are transformed into the full UC statement in Section 4.5.

2 THE DYNAMIC-AVAILABILITY MODEL

This section presents the main components of the security model. We work in the universal composability (UC) framework [9] and assume some basic familiarity with the core concepts. The full version of this paper [3] includes a more detailed introduction. Following a modular approach, we make use of a number of functionalities in our descriptions and explain their behavior whenever needed to follow this exposition.

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 [9]). Informally, security requires that the attack of any adversary $\mathcal{A}$ against the (real-world) protocol can be simulated in the ideal world.

Typically, a protocol is given access to so-called hybrid functionalities, which capture the resources that parties have available, e.g., their communication network or shared randomness. Our results are in the GUC setting which allows parties access to *Global setups* that capture settings where different protocols might share a common state, e.g., a common hash-function [10]. On a more technical note, in order to preserve full UC composability in our statements, as in [4] we capture assumptions/restrictions on the UC adversary and environment as functionality wrappers that explicitly restrict their access to certain functionalities.

A significant extension in the model of computation in our work is the high-resolution treatment of the protocol participant's availability, which we term *(full) dynamic availability*. Concretely, as in [4] all functionalities, protocols, and global setups have a dynamic party set: they all include special instructions allowing parties—and in case of global setups also ideal functionalities—to register, deregister, and allow the adversary to learn the current set of registered parties. These registration commands are part of the specification 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.

Dynamic availability requires special care in the blockchain setting. For example, in [4] it is observed that due to network delays newly joining Bitcoin miners might be temporarily *desynchronized*, i.e., be tricked into working on a fake (adversarial) chain in the first time period after joining the network. As discussed in the end of this section (cf. Figure 1) our work goes one step further in modeling fine-grained, i.e., full, dynamic availability patterns of parties and capturing their respective security guarantees in the blockchain setting.

It is important to point out that a fine-grained availability model offers the capability to precisely model a wide range of real-world concerns. For example, it allows to reason about a protocol's resilience against networks failures or arbitrary message delays. The reason is that such scenarios can be reflected by a particular pattern of stalled and/or disconnected parties for an adversarially chosen amount of time.

The remainder of this section describes, in the dynamic availability model, the real-world resources that Ouroboros Genesis requires and describes the ideal-world functionality that is achieved by the protocol.

2.1 The Real World Execution

Protocol participants are represented as parties in a multi-party computation. The main aspects of this computation are as follows:

Communication. The parties communicate over a network of eventual delivery unicast channels [4]—informally, every party U_p has an open incoming-connections interface where he might receive messages from other parties. This captures the joining procedure of real-world blockchains where new parties find a point of contact and use it to communicate with other parties by means of gossiping. As shown in [4], assuming the honest parties are strongly connected, this setup can realize the *multicast network* with eventual delivery [18, 25, 33]. The abstraction of this network as a (local)[3] UC functionality is given in the full version [3]. For the remainder of this work we will assume parties have direct access to such a multicast network, denoted $\mathcal{F}_{\text{N-MC}}^{\Delta}$, with an upper bound Δ in the delay that the adversary can incur on the delivery of any message. Note that our protocol is oblivious of Δ and this bound in only used in the security statement. Hence from the protocol's point of view the network is no better than that of an eventual delivery network (without a concrete bound on delivery time).

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

Synchrony. 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 [24] and adapted to GUC in [4]. Concretely, the parties are assumed access to a global clock setup, denoted as $\mathcal{G}_{\text{CLOCK}}$ which roughly works as follows: Each registered party can signal the clock that it is done with the current round, and once all honest registered parties (and functionalities) in this session have done so, the clock advances its time counter. In addition, every party can query the clock to read the (logical) time.

As observed in [4], to obtain UC realization in such a globally synchronized setting, the target ideal functionality must keep track of the number of activations that an honest party gets—it can then enforce consistent clock "pace" in the ideal world and real world. This can be achieved by describing the protocol so that it has a predictable pattern of activations before it sends the clock an update command. The precise definition is given in [3] since the exact details are not needed to follow the rest of this paper.

Hash functions as global random oracles. Ouroboros Genesis assumes that parties can query a hash function. As typical in cryptographic proofs, the queries to hash function are modeled as queries 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 κ) 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 common abstraction of random oracles as UC functionalities raises issues with respect to its accuracy for capturing reality [12]. In this work, we adopt the more faithful abstraction given by a *global random oracle* (GRO) $\mathcal{G}_{\text{RO}}$. The fact that Ouroboros Genesis can be proved secure under such an assumption serves as an indication of the augmented composability that PoS can 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 and initial stake distribution of the parties that are present at the beginning of the protocol. This assumption—i.e., that the genesis block is properly created, reliably distributed to the initial parties, and that it is properly communicated to anyone who joins later—is captured in [16] by assuming access to a (local) functionality $\mathcal{F}_{\text{INIT}}$. For each stakeholder registered at the beginning of the protocol, $\mathcal{F}_{\text{INIT}}$ 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 w.l.o.g., as the actual genesis-round index is written on the genesis block and all parties have access to the global clock.

Hybrids used (only) in the security proof. Ouroboros Genesis requires as setup only the above local and global functionalities

$\mathcal{F}_{\text{N-MC}}^{\Delta}$, $\mathcal{F}_{\text{INIT}}$, $\mathcal{G}_{\text{CLOCK}}$, and $\mathcal{G}_{\text{RO}}$. However, for the sake of a clean modular treatment, we also assume hybrid access to two more functionalities from [16], capturing verifiable random functions (VRF) $\mathcal{F}_{\text{VRF}}$ and key-evolving signature schemes (KES) $\mathcal{F}_{\text{KES}}$. These functionalities (which we recall in Appendix A) are UC-realizable by cryptographic constructions [16]; therefore, they can be safely replaced by virtue of the UC composition theorem.

2.2 The Ideal World Execution

We next turn to the functionalities available in the ideal-world. In this world, the parties execute the so-called dummy protocol. Since the clock and the random oracle are modeled as global setups, they are available also in the ideal world. However, 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 specify it below.

Overview. The ledger that Ouroboros Genesis realizes is almost identical to the one proposed in [4] and shown to be implemented by (the UC adaptation of) Bitcoin. Concretely, the ledger of [4] is parameterizable by a collection of four algorithms, and 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. We start with a brief recap of the abstract ledger from [4] and then show which ledger functionality Ouroboros Genesis realizes. The full description of the ledger as pseudo-code is found in [3].

The ledger from [4] 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. These dynamics can be seen as a sliding window over the sequence of state blocks, with width `wSize` and starting at the head of the state, and each party's pointer points to a location within this window (the adversary can control the exact position). As is common in UC, parties advance the ledger when they are activated with specific maintain-ledger input by their environment $\mathcal{Z}$. The ledger uses these queries along with the function predict-time$(\cdot)$ to ensure that the ideal world execution advances with the same pace (relatively to the clock) as the protocol does.[4]

Ledger inputs and state update. Any party can input a transaction to the ledger once instructed by $\mathcal{Z}$. The ledger first validates transactions using a predicate Validate and if valid, these are added to a buffer. Each new block of the state consists of transactions from the buffer. To give protocols syntactic freedom of defining their state block format, a vector of transactions, say $\vec{N}_i$ is mapped to the ith state block via function Blockify$(\vec{N}_i)$. Validate and Blockify are two of the ledger's parametrization algorithms.

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

One crucial property to specify a realistic ledger is the procedure to define when/how to extend `state`, as one needs to find the balance between allowing the adversary certain influence (to reflect real world impacts), and to enforce certain ideal policies/restrictions regarding state updates. For example, our ledger enforces a minimum chain growth rate, a certain chain quality level, and liveness of transactions. The procedure ExtendPolicy is responsible for enforcing such a policy. In nutshell, to enable adversarial influence, 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, ExtendPolicy can ignore the proposal and enforce a default extension.

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

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

where $\text{blockify}_{\text{OG}}$, $\text{predict-time}_{\text{OG}}$, and $\text{ValidTx}_{\text{OG}}$ are identical to what the real protocol prescribes (cf. Section 3).

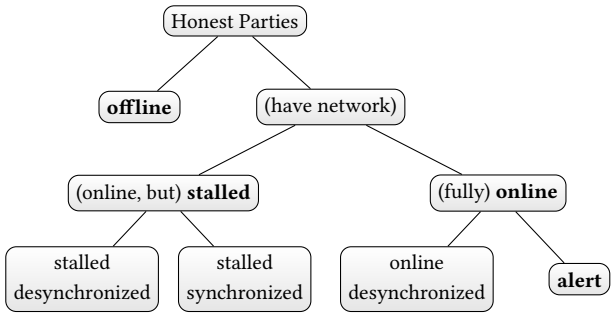
The procedure ExtendPolicy is trickier. It enforces the following properties:

1. All blocks of `state` are semantically valid.
2. The state grows at a minimal rate of blocks over a time interval. This is formalized by specifying a value $\texttt{maxTime}_{\texttt{window}}$ in which at least `wSize` blocks have to be inserted into the ledger state.
3. A certain fraction of blocks in a sequence of `wSize` blocks have to be honestly generated. This is enforced by requiring a limit $\texttt{advBlcks}_{\texttt{window}}$ of adversarial blocks in each window of `wSize` blocks.

A detailed specification of the concrete ExtendPolicy is given in [3].

Guarantees for dynamic availability. The ideal guarantees of [4] separates the active honest parties into two categories, called *synchronized* and *desynchronized*. Desynchronized denotes those parties that have registered with the protocol within the last `Delay` rounds, where `Delay` (usually a multiple of the network delay) is a parameter of the ledger that expresses how long a newly joining party is not considered synchronized. Because we cannot guarantee that these parties' view is consistent with the rest of the honest network, the ledger treats them as adversarial. However, as soon as the interval of `Delay` rounds from registration passes, they become synchronized and enjoy all guarantees for honest parties.

In this work, our goal is to achieve the highest granularity w.r.t. capturing the security of parties depending on their availability status. We go beyond the coarse-grained model of [4], where honest parties are either offline or otherwise fall into two categories, and separate honest parties into the following classes: *offline* parties are honest parties that are deregistered from the network functionality. We further separate parties which are not offline into two (sub-)categories, called *(fully) online*—parties which are registered with all their setups and ideal resources—and *(online but) stalled*—parties that are registered with their local network functionality, but are unregistered with at least one of the global setups $\mathcal{G}_{\text{CLOCK}}$



Honest party	"Synchronized" state	Registered with $\mathcal{G}_{\text{CLOCK}}$ and $\mathcal{G}_{\text{RO}}$	Registered with $\mathcal{F}_{\text{N-MC}}$
- alert	✓	✓	✓
- synchronized	✓	?	✓
- online	?	✓	✓
- stalled	?	×	✓
- offline	?	?	×

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

and $\mathcal{G}_{\text{RO}}$. Each of these (non-offline) subclasses is further split into two subcategories along the lines of [4]: those that have been in their current (non-offline) state for more that `Delay` rounds are *synchronized*, whereas the remainder are *desynchronized*. This classification is illustrated in Figure 1. We will call a party *active* if it is either online (and hence honest) or adversarial.

As in [4], the ledger keeps an updated track of registered parties with all global setups and knows which category each party belongs to. Desynchronized parties are treated as adversarial, whereas, offline and stalled parties remain silent (i.e., the ledger produces no output for them). We note in passing that, although not included in [4], this level of granularity is an interesting extension to the existing Bitcoin analysis.

PoS vs. PoW Ledgers. There is one minor point where the PoS ledger needs to deviate from the Bitcoin one. 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 distribution is reliably reflected (and recall that parties associated to this setup register in the very first round in the protocol execution). As we will see, to ensure that the ledger execution is indistinguishable from the real-world 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.

For the formal specification of the concrete ledger that Ouroboros Genesis realizes we refer to the full version [3].

3 OUROBOROS GENESIS AS A UC-PROTOCOL

The protocol Ouroboros-Genesis resembles the structure of its predecessor Ouroboros Praos [16], but differs drastically in its core, as it invokes a novel chain selection rule. This allows, for the first time in the PoS literature, parties to come and go (and loose and regain access to their resources) at any point without the need of external checkpointing. As already discussed, the protocol assumes access to the network functionalities and global setups, i.e., $\mathcal{F}_{\text{N-MC}}^{\Delta}, \mathcal{F}_{\text{INIT}}, \mathcal{G}_{\text{CLOCK}}$, and $\mathcal{G}_{\text{RO}}$. Due to space limitation, we give a detailed protocol overview in this section that is sufficient to follow and evaluate the protocol and the results, and we refer to [3] for the full specification.

Terminology and notation 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 C (i.e., the number of its blocks) by $\mathsf{len}(C)$. For a chain C and an interval of slots $I \triangleq [\mathtt{sl}_i, \mathtt{sl}_j]$, we denote by $C[I] = C[\mathtt{sl}_i : \mathtt{sl}_j]$ the sequence of blocks in 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 $\#_{i:j}(C) \triangleq \#_I(C) \triangleq |C[I]|$ the number of blocks in $C[I]$.

Before giving the formal specification we introduce some necessary terminology and notation. Each party U stores a local blockchain $C_{\text{loc}}^{U_p}$—U_p's local view of the blockchain.[5] Such a local blockchain is a sequence of blocks B_i ($i > 0$) where each $B \in C_{\text{loc}}$ 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, `st` is the encoded data of this block, and `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 `sl` by providing the output y of U_p's VRF evaluation for this slot, along with the corresponding VRF proof π. 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, σ is the signature by U_p on the entire block (using a key-evolving signature scheme).

If $C_{\text{loc}} = B_0 || \cdots || B_\ell$ 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 ith *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 ε 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 $\text{blockify}_{\text{OG}}$. Concretely, each state block `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.

[5] For brevity, wherever clear from the context we omit the party ID from the local chain notation, i.e., write C_{loc} instead of C_{loc}^{U}.

To enable dynamic availability every party stores in a variable t_{on} (initially set to 1) the time/slot it was last online (and not stalled). It also store in a variable t_{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 C_{loc} and the local buffer `buffer` (corresponding to the transactions seen so far on the network and not added on the blockchain); $\vec{\mathtt{st}}, C_{\text{loc}}$ and `buffer` are all initially empty.

For brevity, whenever in the protocol we say that a party *uses the clock to update,* τ, ep, *and* sl we mean the following step:

- Send (CLOCK-READ, sid_C) to $\mathcal{G}_{\text{CLOCK}}$; receive the current time τ and update $\mathtt{ep} := \lceil \tau/R \rceil$ and slot index $\mathtt{sl} = \tau$, accordingly.[6]

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 or outputting a message to the environment, result into the protocol machine loosing the 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 [24]. 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, ..., 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.

Protocol overview. 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 be τ.

In each slot 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 end, 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 multicasts the new, extended (by one block) chain to its peers. We remark that, as in [16], in order to achieve adaptive security the blocks are signed using a key-evolving signature scheme $\mathcal{F}_{\text{KES}}$ 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, [17], modeled by $\mathcal{F}_{\text{VRF}}$) using the secret key associated with its stake, and providing as inputs to the VRF both the slot index sl and the so-called epoch randomness η (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 π 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 π 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.[7] The idea 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 ep, the stake distribution $\mathbb{S}_{\mathsf{ep}}$ that is used for deriving the threshold T_p^{ep} used for the slot-leader election corresponds to the distribution recorded in the ledger up to the last block of epoch $\mathsf{ep} - 2$. Additionally, the *epoch randomness* η_{ep} for sampling slot leaders in epoch ep is derived as a hash of additional VRF-values y_ρ that were included (together with their respective VRF-proofs π_ρ) into blocks from the first two thirds of epoch $\mathsf{ep} - 1$ 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 $\mathbb{S}_{\mathsf{ep}}$ and η_{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 maxvalid-bg. In fact, the power of the protocol Ouroboros-Genesis and its superiority over all existing

[6] 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).

[7] Unlike [16], where R is fixed, in this work we treat R as a protocol parameter, which will be bounded appropriately by our security statements.

PoS-based blockchains stems from this new chain-selection rule which we discuss in detail below.

The formal structure of the Ouroboros Genesis protocol is given in Figure 2. For completeness, the description includes a block of commands (in the bottom of the description) which specify what parties do when they receive external, protocol-unrelated queries to their setups, such as independent queries to the global random oracle. Because the ideal-world (dummy) parties would forward such queries to their setups, the protocol needs to do the same.

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

Registration/Deregistration *(cf. Section 3.1):*

- Upon receiving input (REGISTER, $\mathcal{R}$), where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol Registration-Genesis(U_p, sid, Reg, $\mathcal{R}$).
- Upon receiving input (DE-REGISTER, $\mathcal{R}$), where $\mathcal{R} \in \{\mathcal{G}_{\text{LEDGER}}, \mathcal{G}_{\text{CLOCK}}, \mathcal{G}_{\text{RO}}\}$ execute protocol Deregistration-Genesis(U_p, sid, Reg, $\mathcal{R}$).

Interacting with the Ledger *(cf. Section 3.2):*

Upon receiving a ledger-specific input $I \in \{$(SUBMIT, ...), (READ, ...), (MAINTAIN-LEDGER, ...)$\}$ verify first that all resources are available. **If** not all resources are available, **then** ignore the input; **else** execute one of the following steps depending on the input I:

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

Handling external (protocol-unrelated) calls to the clock and the RO:

- Upon receiving (CLOCK-READ, sid$_C$) forward it to $\mathcal{G}_{\text{CLOCK}}$ and output $\mathcal{G}_{\text{CLOCK}}$'s response.
- Upon receiving (CLOCK-UPDATE, sid$_C$), record that a clock-update was received in the current round.
- Upon receiving (EVAL, sid$_{RO}$, x) forward the query to $\mathcal{G}_{\text{RO}}$ and output $\mathcal{G}_{\text{RO}}$'s response.

Figure 2: The Ouroboros Genesis Protocol

3.1 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 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 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. Since the registration and deregistration commands are addressed to setups or to the ledger, they only affect 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. W.l.o.g., we do not write the session IDs of global setups and refer to them simply with their name.

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_{on}. (Recall that t_{on} stores the last time the party was online, i.e., connected to all its resources.) The deregistration is performed analogously.

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 and deregistration processes Registration-Genesis and Deregistration-Genesis are specified as pseudo-code in [3].

3.2 Interacting with the Ledger

At the core of the Ouroboros Genesis protocol is the process that maintains 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.[8]

- The command (SUBMIT, sid, 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 $\vec{\text{st}}^{\lceil k}$ of the state $\vec{\text{st}}$ 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).

[8] 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.

- 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 data—i.e., asks the clock for the current time τ, updates its epoch counter ep, its slot counter 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_{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 τ to t_{on} and querying the network the corresponding number of times. The relevant sub-processes involved in the handling of a MAINTAIN-LEDGER query are detailed in [3]. Here we provide a high-level description and discussion of these protocols.

3.2.1. 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 3 for detailed description). As a first step the party receives its keys from $\mathcal{F}_{\text{VRF}}$ and $\mathcal{F}_{\text{KES}}$. 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 $\mathcal{F}_{\text{INIT}}$ to claim its stake.
- In the non-genesis mode, i.e., when $\tau > 1$, the protocol Initialization-Genesis queries $\mathcal{F}_{\text{INIT}}$ to receive the genesis block and uses the received stake distribution to determine the initial threshold T_p^{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 and 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 $\mathcal{F}_{\text{INIT}}$ throws an exception (halts with an error) if the environment does not allow all stakeholders 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 ← true (to make sure that it is never re-initialized) and $t_{\text{on}} \leftarrow \tau$ to remember the last time it became online—which in this case is also the first one.

Protocol Initialization-Genesis(U_p, sid, R)

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

1: Send (KeyGen, sid, U_p) to $\mathcal{F}_{\text{VRF}}$ and $\mathcal{F}_{\text{KES}}$; receiving (VerificationKey, sid, v_p^{vrf}) and (VerificationKey, sid, v_p^{kes}), respectively.
2: Use the clock to update τ, ep ← $\lceil \tau/R \rceil$, and sl ← τ.
// The following brunch in only executed if this is the genesis round
3: **if** $\tau = 0$ **then** execute the following steps in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:
4: Send (ver_keys, sid, U_p, v_p^{vrf}, v_p^{kes}) to $\mathcal{F}_{\text{INIT}}$ to claim stake from the genesis block.
5: Send (CLOCK-UPDATE, sid_C) to $\mathcal{G}_{\text{CLOCK}}$.
6: Use the clock to update τ, ep ← $\lceil \tau/R \rceil$, and sl ← τ. and give up the activation.
7: **while** $\tau = 0$ **do**
8: Use the clock to update τ, ep, and sl and give up the activation.
end while
// The following executed if this is a non-genesis round
9: **else**
10: Send (genblock_req, sid, U_p) to $\mathcal{F}_{\text{INIT}}$. If $\mathcal{F}_{\text{INIT}}$ signals an error then halt. Otherwise, receive from $\mathcal{F}_{\text{INIT}}$ the response (genblock, sid, $\mathbf{G} = (\mathbb{S}_1, \eta_1)$), where
$$\mathbb{S}_1 = \left((U_1, v_1^{\text{vrf}}, v_1^{\text{kes}}, s_1), \ldots, (U_n, v_n^{\text{vrf}}, v_n^{\text{kes}}, s_n)\right) .$$
11: Set $\mathcal{C}_{\text{loc}} \leftarrow (\mathbf{G})$.
12: Set $T_p^{\text{ep}} \leftarrow 2^{\ell_{\text{VRF}}} \phi_f(\alpha_p^{\text{ep}})$ as the threshold for stakeholder U_p for epoch ep, where α_p^{ep} is the relative stake of stakeholder U_p in $\mathbb{S}_{\text{ep}}$ and ℓ_{VRF} denotes the output length of $\mathcal{F}_{\text{VRF}}$.
13: Send (HELLO, sid, U_p, v_p^{vrf}, v_p^{kes}) to $\mathcal{F}_{\text{N-MC}}^{\text{new}}$.
end if
14: Set isInit ← true and $t_{\text{on}} \leftarrow \tau$.

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

Figure 3: The initialization protocol of Ouroboros Genesis (run only the first time a party joins).

3.2.2. 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

Protocol StakingProcedure(k, U_p, ep, sl, buffer, C_{loc})

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

// Determine leader status
1: Send (EvalProve, sid, $\eta_j \,\|\, \texttt{sl} \,\|\, \texttt{NONCE}$) to $\mathcal{F}_{\text{VRF}}$, denote the response from $\mathcal{F}_{\text{VRF}}$ by (Evaluated, sid, y_ρ, π_ρ).
2: Send (EvalProve, sid, $\eta_j \,\|\, \texttt{sl} \,\|\, \texttt{TEST}$) to $\mathcal{F}_{\text{VRF}}$, denote the response from $\mathcal{F}_{\text{VRF}}$ by (Evaluated, sid, y, π).
3: **if** $y < T_p^{\text{ep}}$ **then**
// Generate a new block
4: Set $\texttt{buffer}' \leftarrow \texttt{buffer}$, $\vec{N} \leftarrow \texttt{tx}_{U_p}^{\text{base-tx}}$, and $\texttt{st} \leftarrow \text{blockify}_{\text{OG}}(\vec{N})$
5: **repeat**
6: Parse $\texttt{buffer}'$ as sequence $(\texttt{tx}_1, \ldots, \texttt{tx}_n)$
7: **for** $i = 1$ to n **do**
8: **if** $\text{ValidTx}_{\text{OG}}(\texttt{tx}_i, \vec{\texttt{st}}||\texttt{st}) = 1$ **then**
9: $\vec{N} \leftarrow \vec{N}||\texttt{tx}_i$
10: Remove tx from $\texttt{buffer}'$
11: Set $\texttt{st} \leftarrow \text{blockify}_{\text{OG}}(\vec{N})$
end if
end for
until $\vec{N}$ does not increase anymore
12: Set $crt = (U_p, y, \pi)$, $\rho = (y_\rho, \pi_\rho)$ and $h \leftarrow H(\text{head}(C_{\text{loc}}))$.
13: Send (USign, sid, U_p, $(h, \texttt{st}, \texttt{sl}, crt, \rho)$, sl) to $\mathcal{F}_{\text{KES}}$; denote the response from $\mathcal{F}_{\text{KES}}$ by (Signature, sid, $(h, \texttt{st}, \texttt{sl}, crt, \rho)$, sl, σ).
14: Set $B \leftarrow (h, \texttt{st}, \texttt{sl}, crt, \rho, \sigma)$ and update $C_{\text{loc}} \leftarrow C_{\text{loc}} \,\|\, B$.
// Multicast the extended chain and wait.
15: Send (MULTICAST, sid, C_{loc}) to $\mathcal{F}_{\text{N-MC}}^{\text{bc}}$ and proceed from here upon next activation of this procedure.
end if
16: **while** A (CLOCK-UPDATE, sid_C) has not been received during the current round **do**
Give up activation. Upon next activation of this procedure, proceed from here.
end while

Figure 4: The Ouroboros Genesis staking procedure. The value y is used to evaluate slot leadership: if $y < T_p^{\text{ep}}$ 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, for example the proof of leadership (y, π) and the additional VRF-output (y_ρ, π_ρ) that influences the future epoch-randomness.

to its network. A network latency of, say, Δ rounds, in the delivery of any given messages is then captured by the network withholding this message until Δ FETCH commands are issued (cf. [24]). 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 τ from the clock, and then sets a counter fetchcount to $\tau - t_{\text{on}}$. (Since t_{on} stores the last round that the party was online, fetchcount will be the number of rounds this party was stalled.) Subsequently the party issues 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[9] —by detecting a chain that is guaranteed to originate from an honest and synchronized party. This party does not get to retroactively 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 $t_{\text{on}} = \tau$.

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

3.2.3. The Staking Procedure The next part of the ledger-maintenance protocol is the staking procedure—denoted by StakingProcedure and found in Figure 4—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 sl in an epoch ep if its VRF-output (for an input dependent on sl) is smaller than a threshold value T_p^{ep}. We next discuss how this threshold is computed for the party's current (local) blockchain, where we use the following notation: ℓ_{VRF} denotes the VRF output length in bits. The (local) stake distribution $\mathbb{S}_{\text{ep}}$ at epoch ep corresponding to the (local) blockchain C_{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 C_{loc} (and the genesis block).[10] The relative stake of U_p in the stake distribution $\mathbb{S}_{\text{ep}}$, denoted as $\alpha_p^{\text{ep}} \in [0, 1]$, is the fraction of stake that is associated with this party (more precisely, its public key) in $\mathbb{S}_{\text{ep}}$ out of all stake. The mapping $\phi_f(\cdot)$ is defined as

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

[10]The exact encoding is not of primary relevance. A possible, straightforward encoding is given in [16].

and is parametrized by a quantity $f \in (0, 1]$ called the *active slots coefficient* [16], which is an important parameter of the protocol Ouroboros-Genesis (and its predecessor).

Given the above, the threshold T_p^{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 ϕ has an important property called "independent aggregation" [16]:

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

In particular, when leadership is determined according to ϕ_f, the probability of a stakeholder becoming a slot leader in a particular slot is independent of whether this stakeholder acts as a single party in the protocol, or splits its stake among several "virtual" parties. Therefore, we can conclude that under arbitrary stake distribution, a particular slot has *some* slot leader with probability f (if everyone is participating), giving the active slots coefficient its intuitive meaning.

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 [4], our ledger also has such a transaction filter in place (this makes it suitable for applications like cryptocurrencies). This filter is 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 $\vec{\mathtt{st}} = \mathtt{st}_0 || \ldots || \mathtt{st}_\ell$ includes transactions that are valid with respect to the state $\mathtt{st}_0 || \ldots || \mathtt{st}_{i-1} || \mathtt{st}_i^{-\mathtt{tx}}$, 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. [4, 18]).* [11] *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 [4] 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.*

3.2.4. 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 from a given round—denote the set of all these chains by $\mathcal{N} = \{C_1, \ldots, C_M\}$—and must decide whether to keep his current local chain C_{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 C_{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 maxvalid-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., τ, 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 C is done by an invocation of protocol IsValidChain described in Figure 5. Finally, the party applies our new chain selection rule maxvalid-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. 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 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 α_p^{ep}, and epoch randomness η_{ep}, and the staking threshold T_p^{ep}. This is achieved by the protocol UpdateLocal (see Figure 6). The algorithm used to update the stake parameters, in particular the threshold T_p^{ep}, was discussed in Section 3.2.

Step 2: Filtering out invalid chains. The protocol IsValidChain which filters out invalid chains is the same as the corresponding protocol from [16] (cf. [3] for the full specification.)

Step 3: The new chain selection rule. The chain selection rule maxvalid from [16] (which, to avoid confusion, we hereafter refer to as maxvalid-mc for "moving checkpoint", cf. Section 4) prefers longer chains, unless the new chain C_i forks more than k blocks relative to the currently held chain C_{max} (in which case the new

[11] 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 IsValidChain(U_p, k, C, h, f, R)

if C contains future blocks, empty epochs, starts with a block other than **G**, or encodes an invalid state with isvalidstate($\vec{\mathtt{st}}$) = 0 **then**
  return false
end if
for each epoch ep **do**
  // Derive stake distribution and randomness for this epoch from // chain C
  Set $\mathbb{S}_{\mathsf{ep}}^{C}$ to be the stakeholder distribution at the end of epoch ep − 2 in C.
  Set $\alpha_{p'}^{\mathsf{ep},C}$ to be the relative stake of any party $U_{p'}$ in $\mathbb{S}_{\mathsf{ep}}^{C}$ and $T_{p'}^{\mathsf{ep},C} \leftarrow 2^{\ell_{\mathsf{VRF}}}\phi_f(\alpha_{p'}^{\mathsf{ep},C})$.
  Set $\eta_{\mathsf{ep}}^{C} \leftarrow H(\eta_{\mathsf{ep}-1}^{C} \| \mathsf{ep} \| v)$ where v is the concatenation of the VRF outputs y_ρ from all blocks in C from the first $16k/f$ slots of epoch ep − 1, and $\eta_1^C \triangleq \eta_1$ from **G**.
  for each block B in C from epoch ep **do**
    Parse B as $(h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$.
    // Check hash
    Set badhash ← $(h \neq H(B^{-1}))$, where B^{-1} is the last block in C before B.
    // Check VRF values
    Parse crt as $(U_{p'}, y, \pi)$ for some p'.
    Send (Verify, sid, $\eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{TEST}$, y, π, $v_{p'}^{\mathrm{vrf}}$) to $\mathcal{F}_{\mathsf{VRF}}$,
      get response (Verified, sid, $\eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{TEST}$, y, π, b_1).
    Send (Verify, sid, $\eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{NONCE}$, y_ρ, π_ρ, $v_{p'}^{\mathrm{vrf}}$) to $\mathcal{F}_{\mathsf{VRF}}$,
      get response (Verified, sid, $\eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{NONCE}$, y_ρ, π_ρ, b_2).
    Set badvrf ← $\left(b_1 = 0 \vee b_2 = 0 \vee y \geq T_{U_{p'}}^{\mathsf{ep},C}\right)$.
    // Check signature
    Send (Verify, sid, $(h, \mathtt{st}, \mathtt{sl}, crt, \rho)$, $\mathtt{sl}$, σ, $v_{p'}^{\mathrm{kes}}$) to $\mathcal{F}_{\mathsf{KES}}$,
      get response (Verified, sid, $(h, \mathtt{st}, \mathtt{sl}, crt, \rho)$, $\mathtt{sl}$, b_3).
    Set badsig ← $(b_3 = 0)$.
    if (badhash ∨ badvrf ∨ badsig) **then**
      return false
    end if
  end for
end for
return true

Figure 5: The chain validation (filtering) protocol

chain would be discarded). This so-called *moving checkpointing* is crucial for the security proof in [16]; 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 [20] for a detailed discussion).

Our new chain selection rule, formally specified as algorithm maxvalid-bg(·) (see Figure 7), adapts maxvalid-mc by adding an additional condition (Condition B). When satisfied, the new condition can lead to a party adopting a new chain C_i even if this chain did fork more than k blocks relative to the currently held chain $C_{\max}$. Specifically, the new chain would be preferred if it grows more

Protocol UpdateLocal(k, U_p, R, f)

1: Use the clock to update τ, ep ← $\lceil \tau/R \rceil$, and $\mathtt{sl} \leftarrow \tau$.
2: Set $\mathbb{S}_{\mathsf{ep}}$ to be the stakeholder distribution at the end of epoch ep − 2 in C_{loc}.
3: Set α_p^{ep} to be the relative stake of U_p in $\mathbb{S}_{\mathsf{ep}}$ and $T_p^{\mathsf{ep}} \leftarrow 2^{\ell_{\mathsf{VRF}}}\phi_f(\alpha_p^{\mathsf{ep}})$.
4: Set $\eta_{\mathsf{ep}} \leftarrow H(\eta_{\mathsf{ep}-1} \| \mathsf{ep} \| v)$ where v is the concatenation of the VRF outputs y_ρ from all blocks in C_{loc} from the first $2R/3$ slots of epoch ep − 1.

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

Figure 6: The protocol for updating the local stake distribution parameters.

quickly in the s slots following the slot associated with the last block common to both C_i and $C_{\max}$ (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 maxvalid-bg(C_{loc}, $\mathcal{N} = \{C_1, \ldots, C_M\}$, k, s, f)

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

Figure 7: 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 $C_{\max}$. 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 $\mathcal{F}_{\text{INIT}}$) 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 8.

Protocol SelectChain(C_{loc}, $\mathcal{N} = \{C_1, \ldots, C_M\}$, k, s, R, f)

// Step 1: Updating the local state

1: Invoke protocol UpdateLocal(k, U_p, R, f) and denote the output as τ, ep, sl, $\mathbb{S}_{\text{ep}}$, α_p^{ep}, T_p^{ep}, and η_{ep}.

// Step 2: Filter out invalid chains

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

3: **for** $i = 1 \ldots M$ **do**
Invoke Protocol IsValidChain(C_i); if it returns true then update $\mathcal{N}_{\text{valid}} \leftarrow \mathcal{N}_{\text{valid}} \cup C_i$
end for

// Step 3: Applying the chain selection rule.

4: Execute Algorithm
maxvalid-bg(C_{loc}, $\mathcal{N}_{\text{valid}} = \{C_1, \ldots, C_M\}$, k, s, f) and receive its output C_{max}.

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

Figure 8: The protocol for parties to adopt a (new) chain.

The main ledger-maintenance protocol LedgerMaintenance which stitches together the previously introduced sub-processes can be found in Figure 9.

3.2.5. Reading the State The last command related to the interaction with the ledger is the read command (READ, 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,[12] and then it computes and outputs the prefix of its local chain (chopping of k blocks.) The protocol ReadState is detailed in Figure 10.

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

Protocol LedgerMaintenance(C_{loc}, U_p, sid, k, s, R, f)

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

1: **if** isInit is false **then** invoke Initialization-Genesis(U_p, sid, R); if Initialization-Genesis halts then halt (this will abort the execution); otherwise, use the list of initialized variables v_p^{vrf}, v_p^{kes}, τ, ep, sl, C_{loc}, T_p^{ep}, isInit, t_{on} for the ongoing computations.
end if

2: Execute FetchInformation to receive the newest messages for this round; denote the output by ($C_1, \ldots, C_M$), ($\text{tx}_1, \ldots, \text{tx}_k$), and read the flag WELCOME.

3: **if** WELCOME = 1 **then**

4: Send (MULTICAST, sid, C_{loc}) to $\mathcal{F}_{\text{N-MC}}^{\text{bc}}$.

5: **for** each tx ∈ buffer **do**
Send (MULTICAST, sid, tx) to $\mathcal{F}_{\text{N-MC}}^{\text{tx}}$.
end for
end if

6: Use the clock to update τ, ep ← $\lceil \tau/R \rceil$, and sl ← τ.

7: Set buffer ← buffer||($\text{tx}_1, \ldots, \text{tx}_k$), $t_{\text{on}} \leftarrow \tau$, $\mathcal{N} \leftarrow \{C_1, \ldots,$ and $C_M\}$

8: Invoke Protocol SelectChain(C_{loc}, $\mathcal{N} = \{C_1, \ldots, C_M\}$, k, s, R, f).

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

10: Invoke protocol StakingProcedure(k, U_p, ep, sl, buffer, C_{loc}) (in a (MAINTAIN-LEDGER, sid, minerID)-interruptible manner).

11: Set $t_{\text{work}} \leftarrow \tau$ and send (CLOCK-UPDATE, sid_C) to $\mathcal{G}_{\text{CLOCK}}$.
end if

Figure 9: The main ledger maintenance protocol.

Protocol ReadState(k, C_{loc}, U_p, sid, R, f)

1: **if** isInit is false **then** invoke Initialization-Genesis(U_p, sid, R); if Initialization-Genesis halts then halt (this will abort the execution); otherwise, use the list of initialized variables v_p^{vrf}, v_p^{kes}, τ, ep, sl, C_{loc}, T_p^{ep}, isInit, t_{on} for the ongoing computations.
end if

2: Execute FetchInformation to receive the newest messages for this round; denote the output chains by ($C_1, \ldots, C_M$) (the list of transactions ($\text{tx}_1, \ldots, \text{tx}_k$) and the flag WELCOME can be ignored).

3: Invoke protocol UpdateLocal(k, U_p, R, f) and denote the output as τ, ep, sl, $\mathbb{S}_{\text{ep}}$, α_p^{ep}, T_p^{ep}, and η_{ep}.

4: Use the clock to update τ, ep ← $\lceil \tau/R \rceil$, and sl ← τ.

5: Set $t_{\text{on}} \leftarrow \tau$, $\mathcal{N} \leftarrow \{C_1, \ldots, C_M\}$.

6: Invoke Protocol SelectChain(C_{loc}, $\mathcal{N} = \{C_1, \ldots, C_M\}$, k, s, R, f).

7: Extract the state $\vec{\text{st}}$ from the current local chain C_{loc}.

8: Output (READ, sid, $\vec{\text{st}}^{\lceil k}$) (to $\mathcal{Z}$). // $\vec{\text{st}}^{\lceil k}$ denotes the prefix of $\vec{\text{st}}$ with the last k state blocks chopped off

Figure 10: The protocol for parties to adopt a (new) chain.

4 SECURITY ANALYSIS

After introducing the main desiderata for blockchain protocols in Section 4.1, the security analysis proceeds in four modular steps. In Section 4.2, we present the results of analyzing the predecessor protocol, namely Ouroboros Praos, in a setting where parties could potentially be stalled dynamically. As outlined above, the analysis of this setting requires fundamentally new techniques. For the same setting, the analysis of Ouroboros Genesis follows in Section 4.3 by adopting the new chain selection rule (which is the difference between Praos and Genesis) and analyzing the impact of this change. In Section 4.4 we extend the analysis of Ouroboros Genesis to the setting where new parties can join (and leave) the protocol execution at any time. Finally, in Section 4.5, we establish UC-security, i.e., that Ouroboros Genesis realizes the ledger functionality in a setting with dynamic availability.

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 C_1, C_2 possessed by two alert parties at the onset of the slots $\mathtt{sl}_1 < \mathtt{sl}_2$ are such that $C_1^{\lceil k} \preceq C_2$, where $C_1^{\lceil k}$ denotes the chain obtained by removing the last k blocks from C_1, and $\preceq$ denotes the prefix relation.

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

Chain Quality (CQ**); with parameters** $\mu \in (0,1]$ **and** $k \in \mathbb{N}$**.** Consider any portion of length at least k of the chain possessed by an alert party at the onset of a slot; the ratio of blocks originating from the adversary is at most $1-\mu$. We call μ 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** $\tau \in (0,1], s \in \mathbb{N}$**.** Consider the chains C_1, C_2 possessed by two alert parties at the onset of two slots $\mathtt{sl}_1, \mathtt{sl}_2$ with $\mathtt{sl}_2$ at least s slots ahead of $\mathtt{sl}_1$. Then it holds that $\mathsf{len}(C_2) - \mathsf{len}(C_1) \geq \tau \cdot s$.

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*:

Existential Chain Quality (∃CQ**); with parameter** $s \in \mathbb{N}$**.** Consider a chain 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 $C[\mathtt{sl}_1 : \mathtt{sl}_2]$ contains at least one honestly generated block.

As a side remark, the CG (resp. CQ) property follows from ∃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 as part of our full analysis given in [3].

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.

4.2 Security of Ouroboros Genesis with maxvalid-mc

The original Ouroboros Praos protocol given in [16] differs from Ouroboros Genesis in its chain selection rule, which we call maxvalid-mc here and outline below. For the sake of better comparison, we show maxvalid-mc in Appendix B. The difference in maxvalid-mc compared to maxvalid-bg is that if the considered chain C_i forks from the current chain C_{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.

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 either online or stalled); 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 4.3, the rest of Section 4.2 will be dedicated to sketching its proof, which is fully spelled out in [3]. First, we need to define some relevant quantities.

Definition 4.1 (Classes of parties and their relative stake). Let $\mathcal{P}[t]$ denote the set of all parties at time 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 time t. For a set of parties $\mathcal{P}_{\mathsf{type}}[t]$, let $\mathcal{S}(\mathcal{P}_{\mathsf{type}}[t]) \in [0,1]$ denote the relative stake of the parties in $\mathcal{P}_{\mathsf{type}}[t]$ with respect to the stake distribution used for sampling stake leaders in time t.

Definition 4.2 (Alert ratio, participating ratio). At any time t during the execution, we let: (1) the *alert stake ratio* be the fraction $\mathcal{S}(\mathcal{P}_{alert}[t])/\mathcal{S}(\mathcal{P}_{active}[t])$ of the alert stake out of all active stake; and (2) the *participating stake ratio* be the fraction $\mathcal{S}(\mathcal{P}_{active}[t])$ of all active stake out of all stake. Note that in the setting with static $\mathcal{F}_{\text{N-MC}}$-registration, the set of active parties consists only of alert and adversarial parties, while in general it also contains honest

parties that are online but desynchronized (we will discuss these in detail in Section 4.4).

Theorem 4.3. *Consider the execution of Ouroboros-Praos with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ in the setting with static $\mathcal{F}_{N\text{-}MC}$-registration. Let f be the active-slot coefficient, let Δ be the upper bound on the network delay and let Q be an upper bound on the total number of queries issued to $\mathcal{G}_{RO}$. 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 $R \geq 36\Delta/\epsilon\beta f$ then Ouroboros-Praos achieves the following guarantees:

- ***Common prefix.*** *The probability that Ouroboros-Praos violates the* CP *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* CG *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 the protocol Ouroboros-Praos violates the* $\exists$CQ *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* CQ *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 ϵ_{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 Overview. We give here a proof overview and refer to the full version of this work [3] for the complete analysis. The proof is inspired by the proof of property-based security of Ouroboros Praos given in [16]; 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 [16] 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 CP, CG, and 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 4.3 over the analysis in [16] is that it captures stalled parties (and making honest parties stalled is a fully adaptive decision of the environment). Unfortunately, this 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 [16] 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 work since its generality might form the basis of future analyses of blockchain protocols. We give all the details of the approach in [3]. □

4.3 Adopting the New maxvalid-bg Rule

We now show that essentially the same guarantees as provided by Theorem 4.3 still hold when we replace the chain selection rule maxvalid-mc with maxvalid-bg, arriving at the protocol Ouroboros Genesis. The proof of Theorem 4.4 is found in [3].

Theorem 4.4. *Consider the protocol Ouroboros-Genesis using maxvalid-bg as described in Section 3, executed in the setting with static $\mathcal{F}_{N\text{-}MC}$-registration, under the same assumptions as in Theorem 4.3. 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 4.3 for common prefix, chain growth, chain quality, and existential chain quality are still valid except for an additional error probability

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

4.4 Newly Joining Parties

We next show that the above proven guarantees on common prefix, chain growth and (existential) chain quality 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 α or β. 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 $\tilde{U}$).

To argue about the joining process of U, we consider the above-described execution and look at the first chain that $\tilde{U}$ adopts as an update to its state after $\mathtt{sl}_{\mathsf{join}}$. We call it the *synchronizing chain* of U, denote it C_{sync}. Since it is a rather intuitive notion, we omit a formal definition here and refer to the full version [3]. The heart of our argument is then captured in the following lemma, proven in [3].

Lemma 4.5. *In the same setting as Theorem 4.4 but with dynamic $\mathcal{F}_{\text{N-MC}}$-registrations, any newly joining party will adopt its synchronizing chain, except with probability (5).*

In the full version [3], we discuss and formally analyze the so-called *synchronization time* t_{sync} it takes for the synchronizing chain to appear after U joins the execution, for several variants of the protocol. Our main observations are: (1) Using the default request mechanism presented in Section 3 we have $t_{\mathsf{sync}} = 2\Delta$. (2) If alert parties did multicast their local state every (constant) T rounds, we have $t_{\mathsf{sync}} := T + \Delta$ even without any active request by the newly joining party. (3) The protocol also has a *self-synchronization property*, in the sense that even without any active requests, the party will receive a synchronizing chain eventually.

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 been 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 C_{sync}, it starts participating in the protocol immediately after rejoining. Hence, before it receives C_{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 4.2, and hence we have established the following corollary.

Corollary 4.6. *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 4.4, the guarantees it gives for common prefix, chain growth, and chain quality are valid also in this general setting.*

4.5 Composable Guarantees

We conclude our analysis by showing how the property-focused statement of Corollary 4.6 can be turned into a UC security statement. The statement is conditioned again on the honest majority assumption introduced above. As explained in [4] for fully composable statements, it is desirable not to restrict the environment, but rather model these restrictions as part of the setup. In [4], 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 the full version [3].

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 can enforce its state to grow. The remaining arguments are given in the full proof in [3]. We now state the composable version of Corollary 4.6 (again for the default $t_{\mathsf{sync}} = 2\Delta$ case) as a theorem:

Theorem 4.7. *Let k be the common-prefix parameter and let R be the epoch-length parameter (restricted as in Theorem 4.4), let Δ be the network delay, let τ_{CG} and μ be the speed and chain-quality coefficients, respectively (both defined as in Theorem 4.3), and let α and β refer to the respective bounds on the participation ratios (as in Theorem 4.3). Let $\mathcal{G}_{\text{LEDGER}}$ be the ledger functionality defined in Section 2.2 and instantiate its parameters by*

$$\mathtt{wSize} = k \quad \text{and} \quad \mathtt{Delay} = 2\Delta$$

$$\mathtt{maxTime}_{\mathtt{window}} = \frac{\mathtt{wSize}}{\tau_{\text{CG}}} \quad \text{and} \quad \mathtt{advBlcks}_{\mathtt{window}} = (1-\mu)\mathtt{wSize}.$$

The protocol Ouroboros-Genesis (with access to its specified hybrids) securely UC-realizes $\mathcal{G}_{\text{LEDGER}}$ under the assumptions required by Theorem 4.3. In addition, the corresponding simulation is perfect except with negligible probability in the parameter k when setting $R \geq \omega(\log k)$.

A HYBRID FUNCTIONALITIES IN OUROBOROS GENESIS

Recall that we consider functionalities that handle a dynamic party set. As introduced in [4], the employed mechanism roughly works as follows: the 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. For sake of simplicity, we do not explicitly state these commands in the descriptions below.

Key-Evolving Signatures. The key-evolving signature scheme is employed for signing blocks and a specification is given in Figure 11.

Verifiable Random Function. The verifiable random function functionality is employed during slot-leader election and a specification is given in Figure 11.

Both hybrid functionalities are shown to be implementable by standard cryptographic constructions.

B THE CHAIN SELECTION RULE FROM [16]

To better compare the main step between Ouroboros Genesis and its predecessor Ouroboros Praos, we depict the chain selection rule of Ouroboros Praos in Figure 12.

C LIST OF SYMBOLS

We give here a reference on the symbols and their associated meanings that we used in the main body.

The communication model:

Δ	maximum message delay in slots

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

$\mathcal{F}_{\mathsf{VRF}}$ interacts with parties called $U_1, \ldots, U_n$ as follows:

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

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 stakeholders U_i as follows:

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

Figure 11: The VRF and the key-evolving signatures functionalities from [16].

Functionalities:

$\mathcal{G}_{\mathsf{CLOCK}}$	global clock
$\mathcal{G}_{\mathsf{RO}}$	global random oracle
$\mathcal{F}_{\text{N-MC}}^{\mathsf{bc},\Delta}$	Δ-delayed network for diffusing blockchains
$\mathcal{F}_{\text{N-MC}}^{\mathsf{tx},\Delta}$	Δ-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
$\mathcal{G}_{\mathsf{LEDGER}}$	the ledger functionality

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 ep
α_p^{ep}	relative stake of party U_p in $\mathbb{S}_{\mathsf{ep}}$
η_{ep}	randomness used to sample slot leaders in epoch ep

Analysis:

α	alert stake ratio (Def. 4.2)
β	participating stake ratio (Def. 4.2)
L	total length of the execution (in slots)
Q	total number of queries to the random oracle

Protocol maxvalid-mc(C_{loc}, C_1, ..., C_ℓ)

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

Figure 12: The chain selection rule of Ouroboros Praos.

REFERENCES

[1] Marcin Andrychowicz and Stefan Dziembowski. 2015. PoW-Based Distributed Cryptography with No Trusted Setup. In *CRYPTO 2015, Part II (LNCS)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.), Vol. 9216. Springer, Heidelberg, 379–399. https://doi.org/10.1007/978-3-662-48000-7_19

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 443–458. https://doi.org/10.1109/SP.2014.35

[3] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2018. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. Cryptology ePrint Archive, Report 2018/378. https://eprint.iacr.org/2018/378.

[4] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO 2017, Part I (LNCS)*, Jonathan Katz and Hovav Shacham (Eds.), Vol. 10401. Springer, Heidelberg, 324–356.

[5] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. 2014. Cryptocurrencies without Proof of Work. *CoRR* abs/1406.5694 (2014). http://arxiv.org/abs/1406.5694

[6] Iddo Bentov and Ranjit Kumaresan. 2014. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO 2014, Part II (LNCS)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8617. Springer, Heidelberg, 421–439. https://doi.org/10.1007/978-3-662-44381-1_24

[7] Vitalik Buterin. 2013. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper.

[8] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017). arXiv:1710.09437 http://arxiv.org/abs/1710.09437

[9] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145.

[10] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *TCC 2007 (LNCS)*, Salil P. Vadhan (Ed.), Vol. 4392. Springer, Heidelberg, 61–85.

[11] Ran Canetti and Marc Fischlin. 2001. Universally Composable Commitments. In *CRYPTO 2001 (LNCS)*, Joe Kilian (Ed.), Vol. 2139. Springer, Heidelberg, 19–40.

[12] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. 2014. Practical UC security with a Global Random Oracle. In *ACM CCS 14*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, 597–608.

[13] The NXT Community. 2014. NXT Whitepaper. https://bravenewcoin.com/assets/Whitepapers/NxtWhitepaper-v122-rev4.pdf.

[14] Phil Daian, Rafael Pass, and Elaine Shi. 2016. Snow White: Provably Secure Proofs of Stake. Cryptology ePrint Archive, Report 2016/919. https://eprint.iacr.org/2016/919.

[15] Bernardo David, Rafael Dowsley, and Mario Larangeira. 2018. ROYALE: A Framework for Universally Composable Card Games with Financial Rewards and Penalties Enforcement. *IACR Cryptology ePrint Archive* 2018 (2018), 157. http://eprint.iacr.org/2018/157

[16] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2017. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. Cryptology ePrint Archive, Report 2017/573. http://eprint.iacr.org/2017/573. To appear at EUROCRYPT 2018.

[17] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A Verifiable Random Function with Short Proofs and Keys. In *PKC 2005 (LNCS)*, Serge Vaudenay (Ed.), Vol. 3386. Springer, Heidelberg, 416–431.

[18] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *EUROCRYPT 2015, Part II (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9057. Springer, Heidelberg, 281–310. https://doi.org/10.1007/978-3-662-46803-6_10

[19] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2017. The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Jonathan Katz and Hovav Shacham (Eds.), Vol. 10401. Springer, 291–323. https://doi.org/10.1007/978-3-319-63688-7_10

[20] Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2018. Stake-Bleeding Attacks on Proof-of-Stake Blockchains. Cryptology ePrint Archive, Report 2018/248. https://eprint.iacr.org/2018/248.

[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. Cryptology ePrint Archive, Report 2017/454. http://eprint.iacr.org/2017/454.

[22] Martin Hirt and Vassilis Zikas. 2010. Adaptively Secure Broadcast. In *EUROCRYPT 2010 (LNCS)*, Henri Gilbert (Ed.), Vol. 6110. Springer, Heidelberg, 466–485.

[23] Aljosha Judmayer, Alexei Zamyatin, Nicholas Stifter, Artemios G. Voyiatzis, and Edgar R. Weippl. 2017. Merged Mining: Curse or Cure?. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings (Lecture Notes in Computer Science)*, Joaquín García-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí (Eds.), Vol. 10436. Springer, 316–333. https://doi.org/10.1007/978-3-319-67816-0_18

[24] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *TCC 2013 (LNCS)*, Amit Sahai (Ed.), Vol. 7785. Springer, Heidelberg, 477–498. https://doi.org/10.1007/978-3-642-36594-2_27

[25] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *CRYPTO 2017, Part I (LNCS)*, Jonathan Katz and Hovav Shacham (Eds.), Vol. 10401. Springer, Heidelberg, 357–388.

[26] Sunny King and Scott Nadal. 2012. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. https://peercoin.net/assets/paper/peercoin-paper.pdf.

[27] Ranjit Kumaresan and Iddo Bentov. 2014. How to Use Bitcoin to Incentivize Correct Computations. In *ACM CCS 14*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, 30–41.

[28] Ranjit Kumaresan and Iddo Bentov. 2016. Amortizing Secure Computation with Penalties. In *ACM CCS 16*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 418–429.

[29] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. 2015. How to Use Bitcoin to Play Decentralized Poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 195–206. https://doi.org/10.1145/2810103.2813712

[30] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Aggelos Kiayias (Ed.), Vol. 10322. Springer, 357–375. https://doi.org/10.1007/978-3-319-70972-7_20

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

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

[33] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the Blockchain Protocol in Asynchronous Networks. In *EUROCRYPT 2017, Part II (LNCS)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.), Vol. 10211. Springer, Heidelberg, 643–673.

[34] Rafael Pass and Elaine Shi. 2017. The Sleepy Model of Consensus. In *ASIACRYPT 2017, Part II (LNCS)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.), Vol. 10625. Springer, Heidelberg, 380–409.

[35] Andrew Poelstra. 2014. Distributed Consensus from Proof of Stake is Impossible. https://download.wpsoftware.net/bitcoin/old-pos.pdf.

[36] Alexander Russell, Cristopher Moore, Aggelos Kiayias, and Saad Quader. 2017. Forkable Strings are Rare. Cryptology ePrint Archive, Report 2017/241. https://eprint.iacr.org/2017/241.