Time and slots. We consider a setting where time is divided into discrete units called *slots*. A ledger, described in more detail above, associates with each time slot (at most) one ledger *block*. Players are equipped with (roughly synchronized) clocks that indicate the current slot. This will permit them to carry out a distributed protocol intending to collectively assign a block to this current slot. In general, each slot sl_r is indexed by an integer $r \in \{1, 2, \ldots\}$, and we assume that the real time window that corresponds to each slot has the following two properties: (1) The current slot is determined by a publicly-known and monotonically increasing function of current time. (2) Each player has access to the current time. Any discrepancies between parties' local time are insignificant in comparison with the length of time represented by a slot.

Security Model. We adopt the model introduced by [GKL15] for analysing security of blockchain protocols enhanced with an ideal functionality $\mathcal{F}$. We note that multiple different "functionalities" can be encompassed by $\mathcal{F}$. In our model we employ the "Delayed Diffuse" functionality, which allows for adversarially-controlled delayed delivery of messages diffused among stakeholders.

The Diffuse Functionality. This functionality is parameterized by $\Delta \in \mathbb{N}$ and denoted as $\mathsf{DDiffuse}_\Delta$. It keeps rounds, executing one round per slot. $\mathsf{DDiffuse}_\Delta$ interacts with the environment $\mathcal{Z}$, stakeholders $U_1, \ldots, U_n$ and an adversary $\mathcal{A}$, working as follows for each round:

1. $\mathsf{DDiffuse}_\Delta$ maintains an incoming string for each party U_i that participates. A party, if activated, is allowed at any moment to fetch the contents of its incoming string, hence one may think of this as a mailbox. Furthermore, parties can give an instruction to the functionality to diffuse a message. Activated parties are allowed to diffuse once in a round.
2. When the adversary $\mathcal{A}$ is activated, it is allowed to: (a) Read all inboxes and all diffuse requests and deliver messages to the inboxes in any order it prefers; (b) For any message m obtained via a diffuse request and any party U_i, $\mathcal{A}$ may move m into a special string $\mathsf{delayed}_i$ instead of the inbox of U_i. $\mathcal{A}$ can decide this individually for each message and each party; (c) For any party U_i, $\mathcal{A}$ can move any message from the string $\mathsf{delayed}_i$ to the inbox of U_i.
3. At the end of each round, the functionality also ensures that every message that was either (a) diffused in this round and not put to the string $\mathsf{delayed}_i$ or (b) removed from the string $\mathsf{delayed}_i$ in this round is delivered to the inbox of party U_i. If any message currently present in $\mathsf{delayed}_i$ was originally diffused at least Δ slots ago, then the functionality removes it from $\mathsf{delayed}_i$ and appends it to the inbox of party U_i.
4. Upon receiving $(\mathsf{Create}, U, \mathcal{C})$ from the environment, the functionality spawns a new stakeholder with chain $\mathcal{C}$ as its initial local chain (as it was the case in [KRDO17]).

Modelling Protocol Execution and Adaptive Corruptions. Given the above we will assume that the execution of the protocol is with respect to a functionality $\mathcal{F}$ that incorporates DDiffuse as well as possibly additional functionalities to be explained in the following sections. The environment issues transactions on behalf of any stakeholder U_i by requesting a signature on the transaction as described in Protocol π_{SPoS} of Figure 4 and handing the transaction to stakeholders to put them into blocks. Beyond any restrictions imposed by $\mathcal{F}$, the adversary can only corrupt a stakeholder U_i if it is given permission by the environment $\mathcal{Z}$ running the protocol execution. The permission is in the form of a message $(\mathsf{Corrupt}, U_i)$ which is provided to the adversary by the environment. Upon receiving permission from the environment, the adversary immediately corrupts U_i without any delay, differently from [KRDO17,DPS16], where corruptions only take place after a given delay. Note that a corrupted stakeholder U_i will relinquish its entire state to $\mathcal{A}$; from this point on, the adversary will be activated in place of the stakeholder U_i. The adversary is able to control transactions and blocks generated by corrupted parties by interacting with $\mathcal{F}_{\mathsf{DSIG}}$,$\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{VRF}}$, as described in Protocol π_{SPoS} of Section 3. In summary, regarding activations we have the following: (a) At each slot sl_j, the environment $\mathcal{Z}$ activates all honest stakeholders.[3] (b) The adversary is activated at least as the last entity in each sl_j (as well as during all adversarial party activations and invocations from the ideal functionalities as prescribed); (c) If a stakeholder does not fetch in a certain slot the messages written to its incoming string from the diffuse functionality they are flushed.

[3] We assume this to simplify our formal treatment, a variant of our protocol can actually accomodate "lazy honesty" as introduced in [Mic16]. In this variant, honest stakeholders only come online at the beginning of each epoch and at a few infrequent, predictable moments, see Appendix H.

Restrictions imposed on the environment. It is easy to see that the model above confers such sweeping power on the adversary that one cannot establish any significant guarantees on protocols of interest. It is thus important to restrict the environment suitably (taking into account the details of the protocol) so that we may be able to argue security. We require that in every slot, the adversary does not control more than 50% of the stake in the view of any honest stakeholder. If this is violated, an event $\mathsf{Bad}^{\frac{1}{2}}$ becomes true for the given execution. When the environment spawns a new stakeholder by sending message $(\mathsf{Create}, U, \mathcal{C})$ to the Key and Transaction functionality, the initial local chain $\mathcal{C}$ can be the chain of any honest stakeholder even in the case of "lazy honest" stakeholders as described in Appendix H, without requiring this stakeholder to have been online in the past slot as in [KRDO17]. Finally, we note that in all our proofs, whenever we say that a property Q holds with high probability over all executions, we will in fact argue that $Q \vee \mathsf{Bad}^{\frac{1}{2}}$ holds with high probability over all executions. This captures the fact that we exclude environments and adversaries that trigger $\mathsf{Bad}^{\frac{1}{2}}$ with non-negligible probability.

Random Oracle. We also assume the availability of a random oracle. As usually, this is a function $\mathsf{H}: \{0,1\}^* \to \{0,1\}^w$ available to all parties that answers every fresh query with an independent, uniformly random string from $\{0,1\}^w$, while any repeated queries are answered consistently.

Erasures. We assume that honest users can do secure erasures, which is argued to be a reasonable assumption in protocols with security against adaptive adversaries, see e.g., [Lin09].

3 The Static Stake Protocol

We first consider the static stake case, where the stake distribution is fixed throughout protocol execution. The general structure of the protocol in the semi-synchronous model is similar to that of (synchronous) Ouroboros [KRDO17] but introduces several fundamental modifications to the leader selection process: not all slots will be attributed a slot leader, some slots might have multiple slot leaders, and slot leaders' identities remain unknown until they act. The first modification is used to deal with delays in the semi-synchronous network as the *empty slots*—where no block is generated—assist the honest parties to synchronize. The last modification is used to deal with adaptive corruptions, as it prevents the adversary from learning the slot leaders' identity ahead of time and using this knowledge to strategically corrupt coalitions of parties with large (future) influence. Moreover, instead of using concrete instantiations of the necessary building blocks, we describe the protocol with respect to *ideal functionalities*, which we later realize with concrete constructions. This difference allows us to reason about security in the ideal model through a combinatorial argument without having to deal with the probability that the cryptographic building blocks fail. Before describing the specifics of the new leader selection process and the new protocol, we first formally define the static stake scenario and introduce basic definitions as stated in [KRDO17] following the notation of [GKL15].

In the static stake case, we assume that a fixed collection of n stakeholders $U_1, \ldots, U_n$ interact throughout the protocol. Stakeholder U_i is attributed stake s_i at the beginning of the protocol.

Definition 1 (Genesis Block). *The* genesis block B_0 *contains the list of stakeholders identified by a label U_i, their respective public keys and respective stakes*

$$\mathbb{S}_0 = \left((U_1, v_1^{\mathrm{vrf}}, v_1^{\mathrm{kes}}, v_1^{\mathrm{dsig}}, s_1), \ldots, (U_n, v_n^{\mathrm{vrf}}, v_n^{\mathrm{kes}}, v_n^{\mathrm{dsig}}, s_n) \right) ,$$

and a nonce η.

We note that the nonce η will be used to seed the slot leader election process and that $v_i^{\mathrm{vrf}}, v_i^{\mathrm{kes}}, v_i^{\mathrm{dsig}}$ will be determined by $\mathcal{F}_{\mathsf{VRF}}$, $\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{DSIG}}$, respectively.

Definition 2 (State, Block Proof, Block, Blockchain, Epoch). *A* state *is a string $st \in \{0,1\}^\lambda$. A* block proof *is a value (or set of values) B_π containing information that allows stakeholders to verify if a block is valid. A* block *$B = (sl_j, st, d, B_{\pi_j}, \sigma_j)$ generated at a slot $sl_j \in \{sl_1, \ldots, sl_R\}$ contains the current state $st \in \{0,1\}^\lambda$, data $d \in \{0,1\}^*$, the slot number sl_j, a block proof B_{π_j}*