Ouroboros-BFT: A Simple Byzantine Fault Tolerant Consensus Protocol

Aggelos Kiayias* Alexander Russell†

November 26, 2018

Abstract

We present a simple, deterministic protocol for ledger consensus that tolerates Byzantine faults. The protocol is executed by n servers over a synchronous network and can tolerate any number t of Byzantine faults with $t < n/3$. Furthermore, the protocol can offer (i.) transaction processing at full network speed in the optimistic case where no faults occur, (ii.) instant confirmation: the client can be assured in a single round-trip time that a submitted transaction will be settled, (iii.) instant proof of settlement: the client can obtain a receipt that a submitted transaction will be settled. An equally simple binary consensus protocol can be easily derived as well. We also analyze the protocol in case of network splits and temporary loss of synchrony, arguing the safety of the protocol when synchrony is restored. Finally, we examine the covert adversarial model showing that Byzantine resilience is increased to $t < n/2$.

1 Introduction

The consensus problem, introduced by seminal work of Shostak, Pease and Lamport [13, 10], is one of the fundamental problems in computer science. The problem has received renewed interest over the last decade due to its application to cryptocurrencies such as Bitcoin [11], where a particular variant of consensus known as *ledger consensus* plays a crucial role, cf. [4]. A hallmark feature of such protocols is their Byzantine fault tolerance (BFT), i.e., their ability to tolerate participants that arbitrarily deviate from the specification of the protocol, even when these deviations may be orchestrated by an adversarial entity.

The demand to deploy these protocols in the real world—especially in high-assurance settings—has motivated the design of simple protocols with accessible structure and analyses. This is exemplified by the RAFT protocol [12], a consensus protocol proven to be secure in the fail-stop model (a strictly weaker model than BFT).

Ouroboros-BFT is a new BFT ledger consensus protocol inspired by the design of the Ouroboros protocol [7], a proof-of-stake blockchain protocol. Ouroboros-BFT is a deterministic protocol with simplicity as one of its prime design criteria. It provides the ledger consensus properties of consistency and liveness assuming a number of Byzantine corruptions $t < n/3$. The protocol is analysed in the synchronous setting, though we also show that it is resilient to synchrony failures resulting from either network splits or unanticipated delays; in particular, it has the ability to converge back to safety once synchrony is restored. The protocol provides instant confirmation in the sense that

*University of Edinburgh and IOHK. akiayias@inf.ed.ac.uk. This research was partially supported by H2020 project PRIVILEDGE # 780477 and Cryptography LTD.

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

honest clients can receive instant assurance that a transaction will be eventually settled. The protocol can be also augmented to offer a proof of settlement, in the sense that a transferable receipt can be produced that ensures that settlement will take place. A binary consensus protocol based on Ouroboros-BFT can be easily derived that is also resilient to $t < n/3$ faults. Additionally, the protocol can be advantageously analysed in the covert adversary model [1], which may be enforced through a penalty mechanism; in this setting its resilience is $t < n/2$. In our exposition, we first provide a description of the protocol when parties have access to a global synchronized clock. Then, we show how the parties can simulate access to such a clock and, as a result, that the ledger can optimistically run at the maximum speed supported by the network.

Protocol Overview. Ouroboros-BFT is a simple, deterministic, blockchain-based protocol where servers take turns in a predetermined round-robin fashion diffusing blocks of transactions that extend the longest chain that is available to them. Servers can respond to clients with the (speculative) outcome of a transaction immediately, thus the outcome of a transaction can be obtained by a client in the optimal time of one round-trip. The simplicity of the protocol stems from the fact that servers perform exactly the same steps at each step: (i) extend their blockchain based on information transmitted, (ii) collect new transactions, (iii) issue the next block if they are eligible. Transactions acquire their final sequence in the ledger after $5t+2$ clock ticks, where t is the number of Byzantine nodes. The security analysis of the protocol is based on the forkable strings analysis of the Ouroboros proof-of-stake (POS) protocol [7].

Comparison with other deterministic BFT protocols. Ouroboros-BFT introduces a blockchain based approach in the context of deterministic BFT protocols. Compared to PBFT [2], Ouroboros-BFT (i.) processes speculatively all transactions and issues an instant confirmation or proof of settlement while it serializes the transactions "lazily" using its blockchain-based mechanism; the protocol has total communication complexity $\Theta(n)$ per round in the optimistic case, where no omission of messages occurs, and $\Theta(nt)$ in the worst-case. In PBFT, transactions are settled through a three-phase commit, full-server interaction protocol with communication complexity $\Theta(n^2)$ per block of transactions. (ii.) Servers in Ouroboros-BFT are executing the same basic protocol logic in each slot independently of their view: each server receives blocks and transactions, updates its local blockchain and finally issues responses to clients as well as the next block in case the server is eligible for that slot. Instead, in PBFT there are different steps that each server performs depending on its current state: pre-prepare, prepare, commit, reply, view-change, view-change-acknowledgement and new-view. (iii.) Both Ouroboros-BFT and PBFT provide liveness for final settlement that spans $O(t)$ network time-outs. In the first case, this stems from the fact that the blockchain-based mechanism will require $\Theta(t)$ rounds to settle. In the second case, this stems from the worst-case setting where t consecutive views are assigned to faulty nodes (in fact transaction censorship can be hard to deal with in PBFT since a client has no reliable way of proving that censorship is taking place and force a view change). (iv.) PBFT provides consistency with unbounded delays, while Ouroboros-BFT provides consistency assuming a worst case upper bound on network delay that does not slow down the protocol when no faults occur and if violated, the views of servers may fork but they are guaranteed to return back to a unique view once the upper bound becomes respected again by the network. (v.) In the optimistic case, PBFT will provide the transaction outcome in 5 rounds and $\Theta(n^2)$ communication. In the optimistic case, Ouroboros-BFT will provide a (speculative) transaction outcome and a proof that a transaction will settle in 2 rounds and $\Theta(n)$ communication, while it will assign to a transaction its final sequence after $5t+2$ rounds. Speculative execution is also performed by Zyzzyva, [8] which also produces 2 round responses to clients with

total $\Theta(n)$ communication. However Zyzzyva and Ouroboros-BFT use a different mechanism for aligning the view of the servers: Zyzzyva involves an active client who is responsible for collecting server responses and retransmitting them. Contrary to this, Ouroboros-BFT, as also is the case for PBFT, has clients that are completely passive from the perspective of ledger maintenance.

Tendermint [9] is a more recent deterministic BFT protocol that is based on the partially synchronous BFT protocol by Dwork et al. [3]. Similarly to PBFT described above, in Tendermint, transactions settle through a three-phase commit, full server interaction protocol. In all respects, from our perspective, PBFT and Tendermint are similar protocols and hence the comparison outlined above between Ouroboros-BFT and PBFT carries to the case of Tendermint as well. We note that Tendermint employs a round-robin structure instead of relying to a change-view subprotocol and hence it provides a liveness guarantee of the same type as Ouroboros-BFT (which in both cases requires $\Theta(t)$ rounds in the worst case). Contrary to Ouroboros-BFT, in Tendermint, a server has a variety of different tasks to do depending on its current state: propose, prevote, precommit and commit; note that these tasks mirror closely the steps that servers have to perform in the case PBFT.

SBFT [6] is another recent BFT algorithm in the line of works that follow PBFT. As in Zyzzyva, SBFT achieves $\Theta(n)$ communication by using intermediaries, however instead of outsourcing server alignment responsibilities to the clients it outsources them to "collectors" who are selected from the set of servers in round-robin fashion. In other respects, the protocol retains the three-phase commit structure of PBFT. Moreover, it further exploits the presence of intermediaries and using threshold signatures can minimise the size of messages that intermediaries are required to echo. In other respects the comparison between Ouroboros-BFT and SBFT is similar to that of PBFT, i.e., SBFT provides consistency with unbounded delays, and will (optimistically) provide the transaction outcome in 5 rounds while Ouroboros-BFT will provide a (speculative) transaction outcome in 2 rounds. Ouroboros-BFT provides consistency assuming a worst-case upper bound in network delays which however does not slow down the protocol when no faults occur; if violated, the views of servers may fork but they are guaranteed to return back to a unique view once the upper bound becomes respected again by the network. In both protocols it may take $\Theta(t)$ rounds for a transaction to settle in the worst case. The above provide a first overview of how Ouroboros-BFT compares to previous deterministic BFT protocols. A more thorough comparison is deferred for the next version of the paper.

2 Protocol Description

The protocol treats time as divided into discrete slots, $sl_1, sl_2, \ldots$ and is executed by a set of servers $S_1, \ldots, S_n$. We first describe the protocol in a setting where servers are equipped with synchronized clocks which reliably report to them the current slot; in Section 10 we show how to simulate such clocks using a local timer and a conservative estimate of network delay. Furthermore, servers can "diffuse" a message in each slot which will be delivered to all servers in the next slot. At the discretion of the adversary, messages from adversarial parties may be selectively delivered to only a subset of the servers (and with arbitrary delays). Note that these network assumptions can be obtained by synchronous multicast; in particular, broadcast is not required.

The protocol is executed by n servers who each maintain a *blockchain*: this is a sequence of *blocks* $B_0, B_1, \ldots$ beginning with a special "genesis" block B_0 which contains the servers' public-keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$. (The corresponding secret-keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ are stored locally by each server.) Each subsequent block B_i, $i > 0$, is a quintuple of the form $(h, d, sl, \sigma_{sl}, \sigma_{\text{block}})$, where h is the hash of the previous block, sl is a (slot) time-stamp, d is a set of transactions, σ_{sl} is a signature of the

slot number, and σ_{block} is a signature (of the entire block). While the blocks must have strictly increasing timestamps (slots), there may be slots that are not the timestamp of any block. In fact, the protocol places additional constraints on the blockchain; see the definition of *validity* below.

Each server maintains a mempool of valid transactions with respect to its local blockchain $B_0 B_1 \dots B_l$ and executes iteratively the program shown in Figure 1.

The i-th server locally maintains a blockchain $B_0 B_1 \dots B_l$, an ordered sequence of transactions called mempool and carries out the following protocol:

Clock update and network delivery. With each advance of the clock to a slot sl_j, a collection of transactions and blockchains are pushed to the server by the network layer. Following this, the server proceeds as follows:

1. **Mempool update.** Whenever a transaction tx is received it is added to the mempool as long as it is consistent with the existing transactions in the mempool and the contents of the local blockchain. The transaction is maintained in the mempool for u rounds, where u is a parameter. Optionally, when the transaction enters the mempool the server can return a signed receipt back to the client that is identified as the sender.
2. **Blockchain update.** Whenever the server becomes aware of an alternative blockchain $B_0 B_1' \dots B_s'$ with $s > l$, it replaces its local chain with this new chain provided it is *valid*, i.e., each one of its blocks $(h, d, sl_j, \sigma_{sl}, \sigma_{\text{block}})$ contains proper signatures—one for time slot sl_j and one for the entire block—by server i such that $i - 1 = (j - 1) \bmod n$, h is the hash of the previous block and d is a valid sequence of transactions w.r.t. the ledger defined by the transactions found in the previous blocks.
3. **Blockchain Extension.** Finally, the server checks if it is responsible to issue the next block by testing if $i - 1 = (j - 1 \bmod n)$. In such case, this i-th server is the slot leader. It collects the set d of all valid transactions from its mempool and appends the block $B_{l+1} = (h, d, sl_j, \sigma_{sl}, \sigma_{\text{block}})$ to its blockchain, where $\sigma_{sl} = \mathsf{Sign}_{\mathsf{sk}_i}(sl_j)$, $\sigma_{\text{block}} = \mathsf{Sign}_{\mathsf{sk}_i}(h, d, sl_j, \sigma_{sl})$ and $h = H(B_l)$. It then diffuses B_{l+1} as well as any requested blocks from the suffix of its blockchain that covers the most recent $2t + 1$ slots.

Ledger Reporting. Whenever queried, the server reports as “finalised” the ledger of transactions contained in the blocks $B_0 \dots B_m$, $m \leq l$, where B_m has a slot time stamp more than $3t + 1$ slots in the past. Blocks $B_{m+1} \dots B_l$ are reported as “pending”.

Figure 1: The Ouroboros-BFT protocol with parameters n, t, u corresponding to the total number of servers and maximum number of Byzantine servers, respectively and u corresponding to the “time to live” of a transaction in the mempool.

Ledger states, transactions and receipts. We assume a deterministic parser that maps a given blockchain to a value q that captures the current state of the ledger. The initial state is denoted by q_0. If the ledger is at a state q, a transaction tx added to the ledger will transition the state to q' based on the blockchain parser. We write $q \xrightarrow{\text{tx}} q'$. The outcome of a transaction is defined by the function $R_{\text{tx}}(q')$. We will assume that the function R has the property that if $R_{\text{tx}}(q) \neq \perp$ for some state q then $R_{\text{tx}}(q) = R_{\text{tx}}(q')$, for any q' such that $q \xrightarrow{*} q'$, i.e., the output of a transaction, once it is defined at a certain state q, remains stable and is independent of other transactions being added to the ledger (as a result transactions that seek to read from the state of

the ledger have to specify a specific location or time bound). A receipt for a transaction includes three values: the hash of tx, the hash of q and $R_{\mathrm{tx}}(q')$ where $q \xrightarrow{\mathrm{tx}} q'$. A transaction is invalid for state q if $R_{\mathrm{tx}}(q) = \bot$ and in this case we insist that $q \xrightarrow{\mathrm{tx}} q$. Two transactions commute in case it holds that if $q \xrightarrow{\mathrm{tx}} q' \xrightarrow{\hat{\mathrm{tx}}} q''$ and $q \xrightarrow{\hat{\mathrm{tx}}} \hat{q}' \xrightarrow{\mathrm{tx}} \hat{q}''$, then $q'' = \hat{q}''$. A transaction $\hat{\mathrm{tx}}$ is in conflict with tx at state q if $q \xrightarrow{\mathrm{tx}} q' \xrightarrow{\hat{\mathrm{tx}}} q''$ and $q \xrightarrow{\hat{\mathrm{tx}}} \hat{q}'$ implies that $q' = q'', R_{\hat{\mathrm{tx}}}(q') = \bot$ and $R_{\hat{\mathrm{tx}}}(\hat{q}') \neq \bot$. A transaction tx is consistent with the ledger at state q and a sequence of transactions $\mathrm{tx}_1, \ldots, \mathrm{tx}_k$ in the mempool, if $q \xrightarrow{\mathrm{tx}_1} q_1 \ldots \xrightarrow{\mathrm{tx}_k} q_k$ and tx is valid at state q_k (we remark that some transactions in the mempool may be invalid for the state they are applied to; this does not affect the validity of tx).

3 Security Analysis

We will show that the Ouroboros-BFT protocol satisfies the properties of ledger consensus, namely persistence and liveness. For a definition of these properties we refer to [5] but we recall them below as well for completeness. Note that we define the variant property of "transaction log" consistency as persistence is easily implied by it.

Towards expressing formally the definitions, note that an execution of the protocol is fully determined by an *adversary* $\mathcal{A}$ and an *environment* $\mathcal{Z}$ that provides the input transactions to the nodes. Furthermore, recall that the operation of each server is divided in logical slots $sl_1, sl_2, \ldots$. Note that the honest servers may not necessarily be at the same slot at the same time.

Consistency. For any two servers S_1, S_2, and any two slots $sl_1 \leq sl_2$, if the settled transaction log of server S_1 at slot sl_1 equals LOG_1 and the total transaction log of server S_2 at slot sl_2 equals $\widetilde{\mathrm{LOG}}_2$ (note that the total transaction log includes all pending transactions in the view of S_2), it holds that LOG_1 is a prefix of $\widetilde{\mathrm{LOG}}_2$.

Liveness. If a transaction tx is provided by the environment to all honest servers at a point of the execution when the latest slot among the honest servers is sl, then any server whose clock advances u slots after sl to a slot sl', will have a ledger at a state q for which it holds $q_0 \xrightarrow{*} q_1 \xrightarrow{\mathrm{tx}} q_2 \xrightarrow{*} q$ for some states q_1, q_2; note that $\xrightarrow{*}$ includes only transactions produced by the environment $\mathcal{Z}$ up to slot sl'.

To analyse the protocol we use the forkable strings formalism from [7]. For completeness we recall this below.

Definition 3.1 (Characteristic string)**.** *Fix an execution with genesis block B_0, adversary $\mathcal{A}$, and environment $\mathcal{Z}$. Let $S = \{sl_i, \ldots, sl_j\}$ where $i < j$ be a sequence of slots of length $|S| = \ell$. The* characteristic string *$w \in \{0,1\}^\ell$ associated with this execution (and this sequence of slots) is defined so that $w_k = 1$ if and only if the adversary controls the slot leader of slot sl_k.*

Definition 3.2 (Fork)**.** *Let $w \in \{0,1\}^n$ be a characteristic string and let $H = \{i \mid w_i = 0\}$ denote the set of honest indices. A* fork *for the string w is a directed, rooted tree $F = (V, E)$ with a labeling $\ell : V \to \{0, 1, \ldots, n\}$ so that*

F1. each edge of F is directed away from the root;

F2. the root $r \in V$ is given the label $\ell(r) = 0$;

F3. the labels along any directed path in the tree are strictly increasing;

F4. each honest index $i \in H$ is the label of exactly one vertex of F;

F5. the function $\mathbf{d} : H \to \{1, \ldots, n\}$, defined so that $\mathbf{d}(i)$ is the depth in F of the unique vertex v for which $\ell(v) = i$, is strictly increasing. (Specifically, if $i, j \in H$ and $i < j$, then $\mathbf{d}(i) < \mathbf{d}(j)$.)

As a matter of notation, we write $F \vdash w$ to indicate that F is a fork for the string w. We say that a fork is trivial *if it contains a single vertex, the root.*

Observe that any execution of the Ouroboros-BFT protocol that corresponds to a characteristic string w gives rise to a specific fork F such that $F \vdash w$. Each node of F corresponds to a block produced by one of the parties in the execution. The labeling $\ell(\cdot)$ corresponds to the slot time stamp of each block. Property *F4* is derived from the fact that honest nodes will never issue two blocks with the same slot timestamp. Property *F5* follows from the longest chain rule and synchronicity: an honest party will never issue a block on a shorter chain than the current best available, which must necessarily include all chains diffused by prior honest parties.

Definition 3.3 (Tines, depth, and height; the $\sim$ relation)**.** *A path in a fork F originating at the root is called a* tine*. For a tine t we let* $\text{length}(t)$ *denote its* length*, equal to the number of edges on the path. For a vertex v, we let* $\text{depth}(v)$ *denote the length of the (unique) tine terminating at v. The* height *of a fork (as usual for a tree) is defined to be the length of the longest tine.*

We overload the notation $\ell()$ so that it applies to tines, by defining $\ell(t) \triangleq \ell(v)$, where v is the terminal vertex on the tine t. For two tines t_1 and t_2 of a fork F, we write $t_1 \sim t_2$ if they share an edge. Note that $\sim$ is an equivalence relation on the set of nontrivial tines; on the other hand, if t_ϵ denotes the "empty" tine consisting solely of the root vertex then $t_\epsilon \not\sim t$ for any tine t.

The fundamental blockchain property associated with (a failure of) persistence is the existence of two chains (tines) which substantially diverge from each other, but appear equally valid to a honest observer. This notion of divergence and the precise formulation of "appearing valid to an honest observer" are reflected in the next definition.

Definition 3.4 (Viability; divergence)**.** *Let $F \vdash w$ be a fork for a characteristic string w. We say that a tine t is* viable *if for all honest slots $h \leq \ell(t)$,*

$$\text{length}(t) \geq \mathbf{d}(h)\,.$$

For two viable tines t_1 and t_2 of F, define their divergence *to be the quantity*

$$\text{div}(t_1, t_2) \triangleq \min\{\ell(t_1), \ell(t_2)\} - \ell(t_1 \cap t_2)\,,$$

where $t_1 \cap t_2$ denotes the common prefix of t_1 and t_2. We extend this notation to the fork F by maximizing over viable tines: $\text{div}(F) \triangleq \max_{t_1,t_2} \text{div}(t_1, t_2)$*, taken over all pairs of viable tines of F. We likewise define the divergence of a characteristic string w with the convention that*

$$\text{div}(w) = \max_{F \vdash w} \text{div}(F)\,.$$

Using the above definitions, we define the notion of forkable strings.

Definition 3.5. *We say that a fork is* flat *if it has two tines $t_1 \not\sim t_2$ of length equal to the height of the fork. A string $w \in \{0,1\}^*$ is said to be* forkable *if there is a flat fork $F \vdash w$.*

Theorem 3.6. *Let $w \in \{0,1\}^*$. Then there is forkable substring $\check{w}$ of w with $|\check{w}| \geq \text{div}(w)$.*

The proof is a straightforward adaptation of the proof of Theorem 4.26 of Kiayias, et al. [7]; we include a full proof in Appendix A with a discussion of the differences.

We proceed to show that that low Hamming weight strings are not forkable. For a string $w \in \{0,1\}^\ell$, we let $H(w) = |\{i \mid w_i = 1\}|$ denote the Hamming weight.

Proposition 3.7. *A string of* $\{0,1\}^n$ *with Hamming weight less than* $n/3$ *is not forkable.*

Proof. We prove the contrapositive, i.e., any forkable string has Hamming weight at least $n/3$. Consider a sequence of slots that has as characteristic string the string $w \in \{0,1\}^n$ that is forkable. Let t be the Hamming weight of w. It follows that there is a flat fork F and two tines t_1, t_2, with $t_1 \not\sim t_2$ with $\text{length}(t_1) = \text{length}(t_2)$ equal to the height of the fork.

We first observe that F has height at least $n - t$. This follows directly from the *F5* property. It follows that $\text{length}(t_1) = \text{length}(t_2) \geq n - t$.

Second observe that in each tine there is at most one block that can have index corresponding to a particular 1 in the characteristic string. This follows from the *F3* property. Given this we derive the fact that each 1 in the characteristic string can account for at most two nodes in the tines t_1, t_2.

We conclude the proof by a simple counting argument. First, by the length of the tines, the total number of nodes of F that belong in the tines t_1, t_2 are at least $2(n - t)$. Second, by assumption there are at most $2t$ nodes that can be derived by the 1 positions of the characteristic string. Third, the 0 positions of the characteristic string contribute $n - t$ nodes in F by the *F4* property.

It follows that the total number of nodes of F that are available for t_1, t_2 is at most $n - t + 2t = n + t$. As a result $2(n - t) \leq n + t$ i.e., $t \geq n/3$, as desired. □

Theorem 3.8. *Ouroboros-BFT satisfies persistence and liveness with liveness parameter* $5t + 2$ *under the assumption there are at most* $t < n/3$ *Byzantine parties and the security of the underlying digital signature scheme.*

Proof. (Sketch) We first consider persistence. Suppose the property is violated. It follows that at two different slots sl_1, sl_2 for which $sl_1 \leq sl_2$, the transaction logs $\text{LOG}_1, \widetilde{\text{LOG}}_2$ do not satisfy $\text{LOG}_1 \preceq \widetilde{\text{LOG}}_2$ where LOG_1 is the ordered sequence of transactions in the blockchain of S_1 that is reported as finalised and $\widetilde{\text{LOG}}_2$ is the ordered sequence of transactions in the blockchain of S_2 (at slot sl_2) including pending transactions. Considering the fork that represents the blockchains held by honest players during this execution, it follows that there are two viable tines (note that any tine terminating in an honest vertex is necessarily viable) that diverge over $3t + 1$ slots. This contradicts Theorem 3.6 and Proposition 3.7.

We then consider liveness. Consider a period of $2(3t + 1) - t = 5t + 2$ slots and $n \geq 3t + 1$. The chain of any honest server is guaranteed to advance by $t + 1$ blocks during the first $2t + 1$ slots. Moreover given there are t malicious parties at least one of those blocks will be produced by an honest party. It follows that such block will have a time stamp more than $3t + 1$ slots prior and as a result we obtain liveness with the desired parameter. □

4 Instant Confirmation

In this section we study the protocol from the perspective of instant confirmation of transactions. This operates as follows: whenever a server accepts a transaction in its mempool it issues a signed receipt to the client that includes the hash of the transaction, the current slot, the transaction outcome (if any) and the hash of the state of the blocckhain. A client terminates, accepting a transaction, provided that it receives r receipts where r is a parameter of the protocol. Note that the receipt does not guarantee the position of the transaction in the ledger or its outcome but it does ensure that the transaction will be included eventually as long as its effect is not altered by a conflicting transaction. As a result, instant confirmation is not transferrable and only acts as a promise to settle assuming no other transaction occurs that interferes the given transaction. As a

result, a client cannot implicate the servers in case a transaction that was issued a receipt was not eventually included.

Instant confirmation with parameter u is a liveness property that is defined as follows: for any transaction tx that is issued r receipts where the latest slot among all receipts is sl, it holds that when the clock of any honest server advances u slots after sl to a slot sl', the server will have a *settled* ledger at a state q for which it holds $q_0 \xrightarrow{*} q_1 \xrightarrow{\text{tx}} q_2 \xrightarrow{*} q$ for some states q_1, q_2.

Theorem 4.1. *Ouroboros-BFT satisfies instant confirmation with parameter $5t + 2 + n - r$ under the assumption there are at most $t < n/3$ Byzantine parties, $r \in (2t, n]$ and the security of the underlying digital signature scheme.*

Proof. Consider a certain slot sl' which is after $5t + 2 + n - r$ slots from the latest slot sl reported in $r > 2t$ mempool confirmations for a certain transaction tx; we examine the blockchain $\mathcal{C}$ of an arbitrary honest server at slot sl'. Let $y = n + t - r + 1$; specifically, we will examine the period of $5t + 2 + n - r = (t + y) + (3t + 1)$ slots. During the first $t + y$ slots of this period we know $\mathcal{C}$ has advanced by y blocks since $y \leq n - t$. Furthermore, it must be that $y - t$ of those blocks are honestly produced by distinct honest parties (all these blocks were produced in a sequence of n consecutive slots; uniqueness follows from the round-robin structure of the protocol). Since $r + y > n + t$ (by definition of y), we have that $(r - t) + (y - t) > n - t$ and as a result at least one of these $y - t$ honest blocks belongs to an honest party that has issued one of the r receipts given to the client; it follows that the transaction will be included in the block this honest party issues and hence it will affect its state. Finally the remaining $3t + 1$ slots will ensure the transaction will be in the settled part of the $\mathcal{C}$. □

5 Security Analysis in the Covert Setting

The covert setting refers to the scenario when the adversary does not want to produce any independently verifiable evidence of its misbehaviour. The covert setting significantly simplifies forkable string analysis as shown in [7]. In more detail a covert fork is a fork where the labeling ℓ in Definition 3.2 is injective, which means that the adversary also does not sign with respect to the same slot twice. Given this we have the following.

Proposition 5.1. *([7]) A string $w \in \{0, 1\}^n$ with Hamming weight less than $n/2$ is not covertly forkable.*

Armed with the above proposition, the following theorem can be easily shown as in the case of Theorem 3.8 modifying the protocol so that the finalised party of the ledger includes all blocks before the last $2t + 1$ slots (instead of $3t + 1$).

Theorem 5.2. *Ouroboros-BFT, in the covert setting, satisfies persistence and liveness with liveness parameter $4t+2$ under the assumption there are at most $t < n/2$ Byzantine parties and the security of the underlying digital signature scheme.*

Proof. The proof is essentially identical with that of Theorem 3.8. □ □

We note that, in practice, enforcing covert behaviour can also be achieved by imposing penalties to the misbehaving parties. For instance, submitting the two conflicting signatures to a smart contract can produce a payment to the submitter drawn from an initial escrow deposit that was made by the server.

6 A Binary Consensus Protocol

In the previous section we showed how we can solve ledger consensus. It is also easy to extract an analogous, simple standard (binary) consensus protocol from our construction using the reduction of consensus to ledger consensus suggested by the first construction of [5].

The protocol is as follows. Each server i starts with some input value $v_i \in \{0, 1\}$. When they produce a block they add their input to the data of the block (transactions are ignored). The protocol will terminate after $2n$ slots. The parties will observe their ledger state (excluding any blocks with slot time stamp $n+1, \ldots, 2n$) and output the majority bit. Based on Lemma 3.7, we can easily derive that the honest parties will agree on the same sequence of blocks $B_1, \ldots, B_m$ and hence the same majority bit; this implies agreement. Furthermore we know that $m \geq n - t$. The number of blocks that are contributed by Byzantine parties is at most t as a result there are $m - t$ blocks contributed by honest parties. Given that $t < n/3$, we have that $m - t \geq n - 2t > t$. It follows that if all honest parties initially agree on a value $v \in \{0, 1\}$, this value will have the majority vote among $B_1, \ldots, B_m$ and hence will be the output. This implies validity.

7 Alternative Threat Models

We analysed Ouroboros-BFT in the Byzantine synchronous setting showing its resiliency for any number of malicious parties $t < n/3$. It is worth considering how the protocol behaves in alternative threat models.

Fail-stop corruptions. In the fail-stop model, servers simply fail and stop operating. It is easy to see that in this model the protocol can tolerate any number of failures $f < n$. Namely, as long as one server is still operational consistency is achieved and transactions will continue to be processed with liveness parameter $2n$.

Network splits. In the case of a network split, the network is temporarily partitioned into s connected components for some $s \geq 2$, each one containing $n_1, \ldots, n_s$ servers for a sequence of slots D. Assuming no other failures, it is easy to see that transaction processing will continue normally in each connected component. Furthermore, by slot $\max D + n$, all servers will be activated and the system will converge to a unique blockchain. Indeed, let i be the maximal connected component that includes the server that controls the earliest time slot in the n slots that follow D, say sl_j. It easy to see that after sl_j all servers will converge to the blockchain emitted by this server. It follows that Ouroboros-BFT is resilient to network splits. Note that transactions processed within any other connected component other than the maximal component with the winning server may be lost and hence they have to be resubmitted.

Partial Synchrony. In partial synchrony there is an unknown parameter Δ that determines the maximum delay in message delivery between two honest nodes and the scheduling of messages is adversarial. If Δ still fits within the selected slot length the protocol is unaffected. If it is exceeded and the protocol is allowed to advance, a simple adversarial strategy can create two alternative blockchains: assuming w.l.o.g. that n is even and partition the servers in two sets; then delivery messages with a delay Δ that amounts to two slots, giving a preference w.r.t. parity for each receiving server (i.e., odd parity servers hear from odd parity servers first and likewise for even parity servers). This will produce two blockchains advancing in parallel akin to a network split. In

case the Δ delay returns within the normal range though the protocol converges back to a single blockchain in a similar fashion as described in the network split case.

8 Instant Proof of Settlement

Earlier we showed how the protocol can instantly produce a confirmation to a client that a transaction will settle. However, such statement is not transferable since another conflicting transaction may change the outcome of the transaction (e.g., a double-spending client can cause a transaction to be reverted). Ouroboros-BFT can be amended in the following way to produce an instant proof of settlement.

The core idea of the amendment is to allow transactions in the blockchain even if they are conflicting and have them accumulate votes based on the number of times they are added by different servers. Once they reach a certain threshold, to be determined below, they will be considered settled. As in the case of instant confirmation, the servers will issue a certificate that a transaction will enter their mempool which will serve as part of the proof of settlement. However, if a proof is to be issued, once a server issues its certificate, it runs the risk of accepting another transaction that, in case it settles first, it might change the outcome of the first transaction. To address this we consider the following definition and setting. A sequence of transactions $\mathrm{tx}_1, \ldots, \mathrm{tx}_k$ is permutation safe at state q, if for any permutation π, it holds that in the sequence of updates $q \stackrel{\mathrm{tx}_{\pi(1)}}{\rightarrow} q_1 \ldots \stackrel{\mathrm{tx}_{\pi(k)}}{\rightarrow} q_k$, the k transactions maintain the same k outputs. We describe the protocol in a setting where permutation-safety can be checked efficiently.

First variant ($t < n/4$). Consider the following modification to the core Ouroboros-BFT protocol.

- Ledger Reporting. A transaction tx may be entered multiple times in the ledger. Moreover, the ledger may contain transactions that are in conflict with tx. Each entry of a given tx in the ledger counts as a vote for tx. Parsing the ledger counts only transactions that are finalised; these are the transactions that have received $t+1$ of the votes in the settled part of the ledger. If two finalised transactions are conflicting, only the first one reaching $t+1$ votes in the order determined by the ledger is retained. Recall that the settled part of the ledger contains all the blocks that are more than $3t+1$ slots in the past.

- Mempool Update. The server collects all received transactions as before. A transaction tx enters the updated mempool as long as it is valid and the mempool remains permutation-safe with respect to the ledger state. Recall that the state of the ledger is determined based on finalised transactions only. Thus a transaction may enter the mempool as long as no other transaction has reached $t+1$ votes in the settled part of the ledger despite the fact that some conflicting transactions may exist in the ledger.

The instant proof of settlement is $n-t$ signatures that a transaction has been included in the mempool of the same number of servers and produces the same output.

Instant proof of settlement with parameter u is a liveness property that indicates the following: any transaction by a client that has been issued $n-t$ receipts will be eventually settled in the ledger after u slots, or the client will obtain a proof that a server is corrupt.

Theorem 8.1. *Ouroboros-BFT, amended as above, satisfies persistence and instant proof of settlement with parameter $n+3t+1$ under the assumption there are at most $t < n/4$ Byzantine parties, and the security of the underlying digital signature scheme.*

Proof. We only need to prove the liveness part. Consider a certain slot which is after $n + 5t + 2$ slots from the slot where a client has received $n - t$ mempool receipts for a certain transaction tx. We examine the blockchain $\mathcal{C}$ of an arbitrary honest server at that moment and specifically how it progressed during the period of $n+(3t+1)$ slots after the receipts were issued. After the first n slots of this period we know all honest parties that have issued a receipt for tx had the chance to include it in their blocks. During these n slots $\mathcal{C}$ has advanced by $n-t$ blocks. Moreover it holds that $n-2t$ of these blocks are honestly produced. Since the transaction tx is backed up by $n - t$ mempool receipts, it holds that at least $n - 3t$ honest servers will provide at least $n - 3t \geq t + 1$ votes for tx up to this block. Consider now the case that none of the servers who issued a mempool receipt includes any transaction that invalidates tx in its endorsed input. We call this the non-violation case. (In the case of a violation, the transaction tx may not settle but the client will obtain a proof of server misbehaviour). Consider now another transaction tx$'$ that is in conflict with tx and also receives at least $t + 1$ votes. It follows that at least one server who issued a mempool receipt for tx also voted for tx$'$ conflicting with the non-violation case. It follows that tx will settle after the last $3t + 1$ slots; moreover, it will produce the same output as the one promised: indeed if the output of tx is different it will be because a conflicting transaction tx$'$ has settled first receiving $t + 1$ votes. This means that at least one honest server that voted for tx also produced a receipt for tx$'$ hence violating the permutation safety of its mempool. □

Second variant ($t < n/3$). The second variant is similar to the first with the following addition. The structure of blocks is slightly different. Each block will feature an endorsed-input component which is issued and signed by the block producer and contains all transactions the server is able to include at that slot; in addition to that, the block will be able to carry endorsed input components from previous time slots up to $n + 2t + 1$ slots in the past. Formally now the core protocol is modified as follows.

- Ledger Reporting. A transaction tx may be entered multiple times in the ledger as part of various endorsed inputs; moreover, the ledger may contain endorsed inputs with transactions that are in conflict with tx. Each entry of a given tx in the ledger counts as a vote for tx. Parsing the ledger counts only transactions that are finalised; these are the transactions that have received $t + 1$ of the votes in the settled part of the ledger. If two finalised transactions are conflicting, only the first one reaching $t + 1$ votes in the order determined by the ledger is retained. Recall that the settled part of the ledger contains all the blocks that are more than $3t + 1$ slots in the past.

- Mempool Update. The server collects transactions as well as endorsed-inputs of previous slots up to $n + 2t + 1$ slots in the past. A transaction tx enters the updated mempool as long as it is valid and the mempool remains permutation-safe with respect to the ledger state. Recall that the state of the ledger is determined based on finalised transactions only. Thus a transaction may enter the mempool as long as no other transaction has reached $t + 1$ votes in the settled part of the ledger despite the fact that some conflicting transactions may be part of some endorsed inputs.

- Blockchain Extension. Each server checks if it is responsible for extending the ledger, in which case it operates as before, except for the fact that it prepares the endorsed-input with the set of transactions first and also caches it for $n + 2t + 1$ slots. In the case the server is not responsible for extending the blockchain, it checks whether it should resubmit its endorsed input. This will happen only in case the local blockchain does not include its endorsed-input.

Theorem 8.2. *Ouroboros-BFT satisfies persistence and instant proof of settlement with parameter $n + 5t + 2$ under the assumption there are at most $t < n/3$ Byzantine parties, and the security of the underlying digital signature scheme.*

Proof. We only need to prove the liveness part. Consider a certain slot which is after $n + 5t + 2$ slots from the slot where a client has received $n - t$ mempool receipts for a certain transaction tx. We examine the blockchain $\mathcal{C}$ of an arbitrary honest server at that moment and specifically how it progressed during the period of $n + (2t + 1) + (3t + 1)$ slots after the receipts were issued. After the first n slots of this period we know all honest parties that have issued a receipt for tx had the chance to include it in their endorsed-inputs. In the next $2t + 1$ slots, we know that the chain has advanced by $t + 1$ blocks and at least one of these blocks is honest. Given the cache time-out for endorsed-inputs is $n + 2t + 1$, it holds that up to this honest block, $\mathcal{C}$ will carry all endorsed-inputs of the honest parties that include this transaction. Consider now the case that none of the servers who issued a mempool receipt includes any transaction that invalidates tx in its endorsed input. We call this the non-violation case. (In the case of a violation, the transaction tx may not settle but the client will obtain a proof of server misbehaviour). Since the transaction tx is backed up by $n - t$ mempool receipts, it holds that honest servers will provide at least $n - 2t \geq t + 1$ votes for tx up to this block. Consider now another transaction tx' that is in conflict with tx and also receives at least $t + 1$ votes. It follows that at least one server who issued a mempool receipt for tx also voted for tx' conflicting with the non-violation case. It follows that tx will settle after the last $3t + 1$ slots; moreover, it will produce the same output as the one promised: indeed if the output of tx is different it will be because a conflicting transaction tx' has settled first receiving $t + 1$ votes. This means that at least one server that voted for tx also produced a receipt for tx' hence violating the permutation safety of its mempool. □

9 Bootstrapping from Genesis

Consider a client that wants to connect to the ledger. It asks for blocks and collects as many as possible in $\Delta_{\max}$ local clock ticks. (See below for a discussion about appropriate values for $\Delta_{\max}$.) Subsequently it builds the forest of blocks received and searches for a blockchain segment $B_{s-1}B_s \ldots B_l$ that satisfies the following (i) B_j is signed by the correct server as determined by the timestamp $\text{time}(\text{B}_\text{j})$ for $j = s - 1, \ldots, l$, (ii) $\text{time}(B_l) - \text{time}(B_s) < 3t + 1$, (iii) $\text{time}(B_l) - \text{time}(B_{s-1}) \geq 3t + 1$, and (iv) $l - s + 1 \geq 2t + 1$. The segment $B_s \ldots B_l$ is called a *dense witness* for B_{s-1}. If multiple such segments are found the one with the latest $\text{time}(B_{s-1})$ is chosen. If no dense witness is found, the client collects blocks for another $\Delta_{\max}$ period. Subsequently, the local chain is set to $B_0 \bot B_{s-1} \ldots B_l$ where $\bot$ expresses the gap in the knowledge of the client w.r.t. the public blockchain. The client continues to operate following the servers' programs (executing only steps 2 and 4).

Theorem 9.1. *For any client performing the bootstrapping from genesis process above, it holds that that B_{s-1} is a finalised block according to the view of all honest parties assuming $\Delta_{\max}$ is sufficient time to receive $3t + 1$ blocks from an honest party.*

Proof. First, observe that in the period of $3t + 1$ slots prior to the bootstrapping event for a client, the number of blocks added to any particular honest party's chain is at least $2t + 1$ and at most $3t + 1$. As a result each honest party has added a dense witness to its chain and thus such a witness will be transmitted to the client in $\Delta_{\max}$ steps. Second, suppose that the bootstrapping client disagrees with an honest server S regarding a block that the client considers as finalised. It follows that the chain of S is disjoint from the chain fragment selected by the client over the period of the

$3t+1$ slots of the segment. Given that the segment has at least $2t+1$ blocks, it follows that it includes $t+1$ blocks from honest parties and as a result the chain of S has at most $2t$ blocks in this period of slots, which is a contradiction: all honest parties' chains advance by $2t+1$ slots in a period of $3t+1$ slots. □

Remark 1. *Observe that in the above bootstrapping process, the client does not need to parse the blockchain prior to the block that has the dense witness. Thus, it is possible to complete the bootstrapping step in time independent of the length of the blockchain (of course in this case, at the application layer, it would be impossible to validate information that is stored in the earlier part of the ledger).*

Remark 2. *The above dense witness approach suggests an alternative way to test the validity of a chain $\mathcal{C}$. Consider the following modified dense witness rule. B_{s-1} has $B_s \dots B_l$ as dense witness provided that (i) B_j is properly signed by some server for $j = s-1, \dots, l$, (ii) $\mathrm{time}(B_l) - \mathrm{time}(B_s) < 3t+1$, (iii) $\mathrm{time}(B_l) - \mathrm{time}(B_{s-1}) \geq 3t+1$, and (iv) there are at least $2t+1$ distinct servers that have signed blocks in $B_s \dots B_l$. Observe that condition (i) is relaxed since the timestamp $\mathrm{time}(\cdot)$ is not taken into account. Despite this relaxation the proof of Theorem 9.1 can proceed in exactly the same way due to the modification of condition (iv) that requires at least $2t+1$ distinct servers to support the dense witness. The modified chain selection rule now is as follows: starting from genesis, ensure that each block is followed in the chain by a dense witness. If all blocks pass, this leaves the blocks during the latest $3t+1$ slots to be checked individually as in the standard chain validity rule.*

10 Logical Clocks

We next consider an enhancement of the protocol that does not require synchronized clocks, can safely tolerate message delays of Δ, and can optimistically proceed at maximum network speed.

The protocol will substitute "real-time" slots with "logical" slots determined on-the-fly by message delivery.

The logical clock layer. Specifically, the *clock update and network delivery* step of the protocol (of Figure 1) is now implemented by a new *logical clock* layer which interacts with both the network and the protocol. This layer is responsible for receiving all data from the network (including transactions and posted blocks) and is responsible for delivering "clock advance" events to the blockchain protocol along with appropriate network traffic. The layer will additionally generate (and process) a new type of multicast message: specifically, upon receiving any block from the network, the layer immediately multicasts the block's signature, thus echoing this portion of the block. These messages are used to approximately synchronize the logical clock layers of various parties. More precisely, the logical clock layer maintains a "logical" slot number L, an infinite table $D(s)$ with one entry for each positive integer, and a set of transactions $\textsc{Tx}$. The layer also depends on a local timer. Initially, the layer sets its logical slot number L to sl_0, initializes $\textsc{Tx} = \emptyset$, initializes its "delivery table" so that $D(0)$ contains the genesis block, $D(s) = \emptyset$ for each $s > 1$, and sets its local timer to 0. It then repeatedly carries out the following (written in an event-driven style):

Block delivery event. If the network delivers a valid block B with timestamp sl_s, add the block B to $D(s)$. Extract the slot signature σ from this block, and treat σ as a newly delivered signature; see below.

Signature delivery event. If the network delivers a valid slot signature σ for timeslot sl_s, determine whether $\sigma \in D(s)$. If not, immediately multicast the signature σ and add the signature to the set $D(s)$. If $L = sl_s$, this generates a *logical fast forward*; see below.

Transaction delivery event. If the network delivers a transaction tx, this is added to the transaction set $\textsc{Tx}$.

Timeout event. If the local timer reaches $\Delta_{\max}$, this triggers a *logical clock tick*; see below.

Logical fast forward event. If $D(L) \neq \emptyset$, this triggers a *logical clock tick*; see below.

Logical clock tick (and protocol update) event. A clock advancement event is delivered to the blockchain protocol. Writing $L = sl_\ell$, all blocks appearing in $\bigcup_{s \leq \ell} D(s)$ are delivered to the blockchain protocol and removed from $D(s)$. (Signatures are retained in D to avoid re-broadcasting previous signatures.) All transactions appearing in $\textsc{Tx}$ are delivered to the blockchain protocol and $\textsc{Tx}$ is set to $\emptyset$. The timer is set to 0 and the logical clock L is set to $sl_{\ell+1}$. Any message sent by the blockchain protocol is multicast to the network (and the blocks of this message immediately generate block delivery events).

In the theorem below we show that as long as actual network delays Δ are less than $\Delta_{\max}/2$ then the logical clocks as described above can simulate a common global clock with enough accuracy to guarantee liveness and persistence (as long as the underlying digital signature is secure). We remark that an adversary can arrange for the logical clocks of various honest parties to be—at least momentarily—very far out of synchrony. As an example, if the first t slots of the round robin schedule happen to be associated with adversarial participants, the adversary can immediately deliver t blocks to a particular honest party P, thus instantaneously advancing its logical clock to sl_{t+1}. Other honest parties' clocks may not "catch up" until they finally receive the signature echo messages from P. Despite this, the proof below will argue that the blockchains broadcast by honest participants of this system still satisfy the fork axioms, and hence that security is maintained.

Theorem 10.1. *The logical clock implementation provides no advantage to the adversary, provided that digital signature security holds and $\Delta_{\max} > 2\Delta$, where Δ time is sufficient to enable the complete propagation of $2t+1$ blocks in the network.*

In preparation for the proof we set down some notation. For an execution of Ouroboros-BFT with logical clocks and an absolute time t, we let $L^P(t)$ denote the value of the logical clock L of participant P at this time and let $\mathrm{timer}^P(t)$ denote the value of the timer of participant P at this time. It is convenient to treat these quantities together: Define

$$\hat{L}^P(t) = L^P(t) + \frac{\mathrm{timer}^P(t)}{\Delta_{\max}} \qquad \text{and note that} \qquad L^P(t) = \lfloor \hat{L}^P(t) \rfloor \,.$$

Similarly, let $O^P(t)$ denote the set of slots which player P considers to have been "occupied" by this time t; that is, the set of all slots for which the player's logical clock layer has received a valid signature.

Lemma 10.2. *Consider an execution of Ouroboros-BFT with logical clocks. Then for all pairs of honest players P and P' and all $t \geq 0$, $O^{P'}(t) \subset O^P(t+\Delta)$ and $\hat{L}^P(t+\Delta) \geq \hat{L}^{P'}(t)$.*

Proof. Considering that honest participants immediately multicast any signature they receive and that messages propagate in time Δ, it is clear that $O^{P'}(t) \subset O^P(t+\Delta)$ for all pairs P, P' of honest participants. Consider now the logical clocks $\hat{L}^P(t)$ and $\hat{L}^{P'}(t)$ for a pair of honest participants

P, P'. In general, the behavior of a logical clock $\hat{L}^P(t)$ over an interval $t \in [0, t_{\max})$ is determined entirely by the schedule of delivery of signatures to P over this interval (as these can cause the logical clock to instantaneously "fast forward" over slots "occupied" by signatures). In particular, (assuming a finite number of signature deliveries) a logical clock function may have a finite number of discontinuities between which the function is simply linear with slope $1/\Delta_{\max}$. For concreteness, we define these functions so that they are left continuous (which is to say that at points of discontinuity the logical clock is set to the larger of the relevant times associated with the fast forward events). Note then that $\hat{L}^P(\Delta) \geq \hat{L}^{P'}(0)$, as any delivered signatures that may have affected $\hat{L}^{P'}$ at time 0 have been delivered to P by time Δ. To complete the proof, we must show that there is no positive time t for which $\hat{L}^{P'}(t) > \hat{L}^P(t + \Delta)$; if such a time exists we may consider the infimum $M = \inf\{t \mid \hat{L}^{P'}(t) > \hat{L}^P(t+\Delta)\}$. Considering that the functions are piecewise linear (with common slope) as described above it follows that $\hat{L}^{P'}(M) > \hat{L}^P(M + \Delta)$ and that a signature σ must have been delivered to P' at this time M, causing a logical clock tick event. Note, however, that any signatures delivered to P' at time M have been delivered to P by time $M + \Delta$, and hence that any signatures "fast-forwarded over" by P' (at M) must also be fast-forwarded over by P at time $M + \Delta$ or previously. Hence $\hat{L}^P(M + \Delta) \geq \hat{L}^{P'}(M)$, a contradiction. □

Proof of Theorem 10.1. In order for the collection of blockchains held by honest participants during an execution of the protocol to satisfy the fork axioms, it suffices to ensure that if honest participant P appears prior to honest participant P' in the round robin schedule then any message multicast by P will be delivered to P' "in time". (Note that the other fork axioms will necessarily be satisfied, as the protocol still ensures that honest participants sign no more than one blockchain block per slot, and that valid blockchains must consist of blocks with increasing slot numbers.)

Consider, then, two slots $sl < sl'$ associated with honest participants P and P'. We must ensure that any message multicast by P associated with slot sl will arrive at P' before the logical clock of P' reaches sl'. Let t be the absolute time at which $\hat{L}^{P'}(t) = sl'$; for the sake of contradiction, suppose that the message generated by P for slot sl has not yet arrived at P'. In this case, $sl \notin O^{P'}(t)$ and it must have taken the logical clock of P' at least $\Delta_{\max}$ time just to cross the slot sl. In particular, $\hat{L}^{P'}(t - \Delta_{\max}) \geq sl$ and, in light of Lemma 10.2, $\hat{L}^P(t - \Delta_{\max} + \Delta) \geq \hat{L}^{P'}(t - \Delta_{\max}) \geq sl$. Thus the message generated by P for slot sl will be delivered to P' by time $t - \Delta_{\max} + 2\Delta < t$ so long as $2\Delta < \Delta_{\max}$, a contradiction. Note that the proof implicitly demands secure signatures; it is critical that P is the only participant that can generate a signature that will induce P' to fast-forward over the slot sl. □

11 Evolving the Slot Leader Selection Function - Proofs of Membership

The main protocol uses a round-robin schedule for selecting the slot leaders. However this is not essential for the protocol's security. In this section we consider some variations as well as a setting where the protocol evolves its slot leader selection function and the server membership list itself.

Let us divide the protocol execution in periods of slots called epochs. Each epoch is of length m slots and a set of servers $\mathcal{S}_e$ that are determined based on registration information that is posted up to the latest block of the previous epoch that has settled (note that this imposes a lower bound on m) or the genesis block. Moreover, each epoch e, will have its own schedule determined by a function $L_e(\cdot)$, so that $L_e(sl)$ returns the identity of a server $S_i \in \mathcal{S}_e$ that is elected to be a leader of the slot.

Note that the set of servers $\mathcal{S}_1$ that corresponds to the first epoch is hardcoded into the genesis

block. Subsequent sets $\mathcal{S}_2, \mathcal{S}_3, \ldots$ take into account server registration certificates that are posted on the blockchain. A server registration certificate is a special transaction that contains the identity and public-key of the server as well as a set of signatures from the existing servers that conform to a ledger-wide membership rule. For instance, an example of a threshold rule would be that that t-out-of-n servers should approve the new member where t is a fixed parameter. At the same time resignation messages can be similarly permitted. These can be signed unilaterally or again require the approval of some of the existing servers.

The function $L_e(\cdot)$ takes into account the set $\mathcal{S}_e$ and may also consider load balancing considerations. The details of $L_e(\cdot)$ are determined in the latest of block of the previous epoch or the genesis block in the case of $L_1(\cdot)$. In such case posting any block can be viewed as a "proof-of-membership" where each server produces a block as dictated by the function $L_1(\cdot)$. It is easy to see that there are wide classes of $L_e(\cdot), \mathcal{S}_e$, where the results of the previous sections can be easily ported. We leave the details of characterising these classes of functions for future work.

12 Checkpointing POW-Blockchains

Finally, we show how to use the protocol to "checkpoint" a POW-blockchain. The objective is to protect a POW-based blockchain protocol from 51% attacks.

A checkpoint refers to a specific block of the underlying POW-blockchain. We assume (1) a predicate $C(\cdot)$ that determines whether a block B is a checkpoint candidate, (2) a relation $R(\cdot,\cdot,\cdot)$ that determines whether block B is a valid block with respect to a previous block B' using possibly a witness w; specifically we write $R(B', w, B)$ to mean that B is a valid block extending previous checkpoint block B' as attested by witness w. When extending a blockchain $\mathcal{C}$ whose latest checkpoint is B' it holds that when a block B is produced that satisfies $C(\cdot)$ it holds that the miner is capable to infer a witness w that satisfies $R(B', B, \cdot)$.

The POW-based protocol is modified as follows:

- Whenever an update of the local blockchain is performed, it is verified that new candidate chain is consistent with the latest checkpoint as reported by an Ouroboros-BFT node. In case of a discrepancy, the update is dropped.
- Prior to extending the local blockchain it is tested whether it contains the latest checkpoint by querying an Ouroboros-BFT node.
- When a block B is mined extending a chain $\mathcal{C}$ for which $C(B)$ holds, a transaction containing (B', B, w) is transmitted to the Ouroboros-BFT nodes to register B as a checkpoint.

Ouroboros-BFT nodes store in their blockchain transactions of the form (B', B, w). The rule is that if their blockchain contains a transaction of the form (B', B, w) then any other transaction (B', B^*, w^*) with $B^* \neq B$ is rejected (irrespectively of whether it satisfies predicate R).

In order to save storage only the headers of blocks B', B need to be included in the transaction. However, the Ouroboros-BFT nodes can cache the contents of blocks B and relay them on demand on the POW protocol network. We note that the relation $R(\cdot,\cdot,\cdot)$ should verify the validity of the contents of the blocks; if this is not performed there is a danger of checkpointing a block with invalid contents with respect to the POW-based blockchain. Observe that an attacker can momentarily fork two honest Ouroboros-BFT nodes into checkpointing two different POW blocks. Such a fork will also translate to a fork in the underlying POW-based blockchain. This will be resolved once the Ouroboros-BFT blockchain advances and one of the checkpoints becomes finalised.

Acknowledgements

We are grateful to Peter Gaži and Roman Oliynykov for comments and helpful discussions.

References

[1] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[3] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[4] Juan Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. Cryptology ePrint Archive, Report 2018/754, 2018. `https://eprint.iacr.org/2018/754`.

[5] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[6] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

[7] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.

[8] Ramakrishna Kotla, Allen Clement, Edmund L. Wong, Lorenzo Alvisi, and Michael Dahlin. Zyzzyva: speculative byzantine fault tolerance. *Commun. ACM*, 51(11):86–95, 2008.

[9] Jae Kwon. Tendermint : Consensus without mining. https://tendermint.com/static/docs/tendermint.pdf.

[10] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

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

[12] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319. USENIX Association, 2014.

[13] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

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

A Divergence and forkability

In order to discuss the detailed relationship with the original definitions of Kiayias et al. [7] and the amplifications of Blum et al. [14], we begin with a more detailed definition of divergence.

Definition A.1 (Divergence)**.** *Let F be a fork for a string $w \in \{0,1\}^*$. For two viable tines t_1 and t_2 of F, we wish to measure the extent to which they provide "differing histories" of the system. There are two natural measures, one focusing on the number of blocks of disagreement and one focusing on the number of slots over which the chains disagree. Specifically, we define their* block divergence *to be the quantity*

$$\operatorname{div}_B(t_1, t_2) = \min_i(\operatorname{length}(t_i) - \operatorname{length}(t_1 \cap t_2))$$

and their slot divergence *(or, simply,* divergence*) to be the quantity*

$$\operatorname{div}_S(t_1, t_2) = \min_i(\ell(t_i) - \ell(t_1 \cap t_2))\,;$$

here $t_1 \cap t_2$ denotes the common prefix of t_1 and t_2. We define suitable versions of these quantities to forks by maximizing over tines: specifically, for each of these two notions of divergence, we define:

$$\operatorname{div}_\Box(F) = \max_{\substack{t_1,\, t_2 \text{ viable} \\ \text{tines of } F}} \operatorname{div}_\Box(t_1, t_2)\,,$$

where $\Box$ is a placeholder for B or S. Finally, define the divergence of w *to be the maximum such divergence over all possible forks for w:*

$$\operatorname{div}_\Box(w) = \max_{F \vdash w} \operatorname{div}_\Box(F)\,.$$

For consistency with the prior parts of the paper, when we drop the subscript we use the convention that $\operatorname{div}(\cdot) = \operatorname{div}_S(\cdot)$*.*

A few historical remarks about divergence: The original paper [7] where divergence was defined and studied used the block-based notion $\operatorname{div}_B(\cdot,\cdot)$; that article established an analogue of Theorem 3.6 for $\operatorname{div}_B()$. Later refinements [14] focused on the slot-based notion $\operatorname{div}_S(\cdot,\cdot)$, but studied this using quite different tools (e.g., the notion of "relative margin"). The proof of Theorem 3.6 below is nearly identical to the original $\operatorname{div}_B()$ version of Kiayias et al., though some parts of the argument can be slightly simplified in this setting.

Returning now to the proof, we wish to establish a direct relationship between divergence and the existence of a forkable substring (Theorem 3.6 of the main text).

Theorem A.2 (Restatement of Theorem 3.6; cf. Theorem 4.26 of [7])**.** *Let $w \in \{0,1\}^*$. Then there is forkable substring $\check{w}$ of w with $|\check{w}| \geq \operatorname{div}(w)$.*

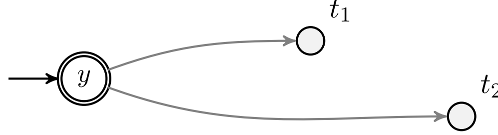
Proof. Consider a fork $F \vdash w$ and a pair of viable tines (t_1, t_2) for which

$$\operatorname{div}(t_1, t_2) = \operatorname{div}(w)\,. \tag{1}$$

For simplicity, we assume the tines have been labeled so that $\ell(t_1) < \ell(t_2)$ and further that

$$|\ell(t_2) - \ell(t_1)| \quad \text{is minimum among all pairs of tines for which (1) holds.} \tag{2}$$

We begin by identifying the substring $\check{w}$; the remainder of the proof is devoted to constructing a flat fork for $\check{w}$ to establish forkability. Let y denote the last vertex on the tine $t_1 \cap t_2$, as in the diagram below, and let $\alpha \triangleq \ell(y) = \ell(t_1 \cap t_2)$.



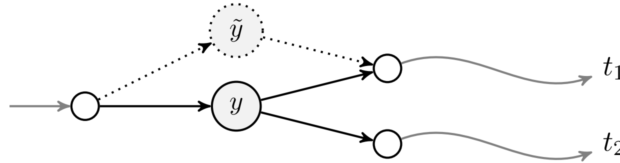
Let β denote the smallest honest index of w for which $\beta \geq \ell(t_2)$, with the convention that if there is no such index we define $\beta = n + 1$. Observe that, in any case, $\ell(t_1) < \ell(t_2)$ and hence that $\beta - 1 \geq \ell(t_1)$. These indices, α and β, distinguish the substring $\check{w} = w_{\alpha+1} \dots w_{\beta-1}$, which will be the subject of the remainder of the proof. As the function $\ell(\cdot)$ is strictly increasing along any tine, observe that

$$|\check{w}| = \beta - \alpha - 1 \geq \ell(t_1) - \ell(y) \geq \min(\ell(t_1), \ell(t_2)) - \ell(t_1 \cap t_2) = \operatorname{div}(w)\,,$$

so $\check{w}$ has the desired length and it suffices to establish that it is forkable.

We briefly summarize the proof before presenting the details. We begin by establishing several structural properties of the tines t_1 and t_2 that follow from the assumptions (1) and (2) above. To establish that $\check{w}$ is forkable we then extract from F a flat fork (for $\check{w}$) in two steps: (i.) the fork F is subjected to some minor restructuring to ensure that all "long" tines pass through y; (ii.) a flat fork is constructed by treating the vertex y as the root of a portion of the subtree of F labeled with indices of $\check{w}$. At the conclusion of the construction, segments of the two tines t_1 and t_2 will yield the required "long, disjoint" tines satisfying the definition of forkable.

Step I: general structural comments. We observe, first of all, that the vertex y cannot be adversarial: otherwise it is easy to construct an alternative fork $\tilde{F} \vdash w$ and a pair of tines in $\tilde{F}$ that achieve larger divergence. Specifically, construct $\tilde{F}$ from F by adding a new (adversarial) vertex $\tilde{y}$ to F for which $\ell(\tilde{y}) = \ell(y)$, adding an edge to $\tilde{y}$ from the vertex preceding y, and replacing the edge of t_1 following y with one from $\tilde{y}$; then the other relevant properties of the fork are maintained, but the divergence of the resulting tines has increased by at least one. (See the diagram below.)



A similar argument implies that the fork $F_0 \vdash w_1 \dots w_\alpha$ obtained by including only those vertices of F with labels less than or equal to $\alpha = \ell(y)$ has a unique vertex of depth $\operatorname{depth}(y)$ (namely, y itself). In the presence of another vertex $\tilde{y}$ (of F_0) with depth $\operatorname{depth}(y)$, "redirecting" t_1 through $\tilde{y}$ (as in the argument above) would likewise result in a fork with larger divergence. Note that $\ell(\cdot)$

would indeed be increasing along this new tine (resulting from redirecting t_1) because $\ell(\tilde{y}) \leq \ell(y)$ according to the definition of F_0. As α is the last index of the string, this additionally implies that F_0 has no vertices of depth exceeding depth(y).

We remark that the assumptions (2) and (1) imply that there are no honest indices h for which $\ell(t_1) < h < \ell(t_2)$. Otherwise, consider the alleged tine t_h for which $\ell(t_h) = h$ and recall that t_h is viable by definition. Note that t_h must be disjoint with one of the two tines t_1, t_2 beyond the slot α; in case t_h is disjoint from t_1 in this region, the pair t_1, t_h achieve the same divergence (as t_1, t_2) but $|\ell(t_1) - \ell(t_h)| < |\ell(t_1) - \ell(t_2)|$, which contradicts (2). On the other hand, if t_h is disjoint from t_2 in this region, the pair t_2, t_h achieve strictly larger divergence (than t_1, t_2) which contradicts (1). As both t_1 and t_2 are viable, it follows immediately that any honest index h for which $h < \beta$ has depth no more than $\min(\text{length}(t_1), \text{length}(t_2))$: specifically,

$$h < \beta \quad \Longrightarrow \quad \mathbf{d}(h) \leq \min(\text{length}(t_1), \text{length}(t_2))\,. \tag{3}$$

Step II. Pinching the fork at y. In light of the remarks above, we observe that the fork F may be "pinched" at y to yield an essentially identical fork $F^{\rhd y \lhd} \vdash w$ with the exception that all tines of length exceeding depth(y) pass through the vertex y. Specifically, the fork $F^{\rhd y \lhd} \vdash w$ is defined to be the graph obtained from F by changing every edge of F directed towards a vertex of depth depth$(y) + 1$ so that it originates from y. To see that the resulting tree is a well-defined fork, it suffices to check that $\ell(\cdot)$ is still increasing along all tines of $F^{\rhd y \lhd}$. For this purpose, consider the effect of this pinching on an individual tine t terminating at a particular vertex v—it is replaced with a tine $t^{\rhd y \lhd}$ defined so that:

- If length$(t) \leq$ depth(y), the tine t is unchanged: $t^{\rhd y \lhd} = t$.
- Otherwise, length$(t) >$ depth(y) and t has a vertex z of depth depth$(y) + 1$; note that $\ell(z) > \ell(y)$ because F_0 contains no vertices of depth exceeding depth(y). Then $t^{\rhd y \lhd}$ is defined to be the path given by the tine terminating at y, a (new) edge from y to z, and the suffix of t beginning at z. (As $\ell(z) > \ell(y)$ this has the increasing label property.)

Thus the tree $F^{\rhd y \lhd}$ is a legal fork on the same vertex set; note that depths of vertices in F and $F^{\rhd y \lhd}$ are identical.

By excising the tree rooted at y from this pinched fork $F^{\rhd y \lhd}$ we may extract a fork for the string $w_{\alpha+1} \ldots w_n$. Specifically, consider the induced subgraph $F^{y \lhd}$ of $F^{\rhd y \lhd}$ given by the vertices $\{y\} \cup \{z \mid \text{depth}(z) > \text{depth}(y)\}$. By treating y as a root vertex and suitably defining the labels $\ell^{y \lhd}$ of $F^{y \lhd}$ so that $\ell^{y \lhd}(z) = \ell(z) - \ell(y)$, this subgraph has the defining properties of a fork for $w_{\alpha+1} \ldots w_n$. In particular, considering that α is honest it follows that each honest index $h > \alpha$ has depth $\mathbf{d}(h) >$ length(y) and hence labels a vertex in $F^{y \lhd}$. For a tine t of $F^{\rhd y \lhd}$, we let $t^{y \lhd}$ denote the suffix of this tine beginning at y, which forms a tine in $F^{y \lhd}$. (If length$(t) \leq$ depth(y), we define $t^{y \lhd}$ to consist solely of the vertex y.) Note that $t_1^{y \lhd}$ and $t_2^{y \lhd}$ share no edges in the fork $F^{y \lhd}$.

Step III. Extracting the fork for $\check{w}$. Finally, let $\check{F}$ denote the tree obtained from $F^{y \lhd}$ as the union of all tines t of $F^{y \lhd}$ so that all labels of t are drawn from $\check{w}$ (as it appears as a prefix of $w_{\alpha+1} \ldots w_n$), and

$$\text{length}(t) \leq \max_{\substack{h \leq |\check{w}| \\ h \text{ honest}}} \mathbf{d}(h)\,.$$

It is immediate that $\check{F} \vdash \check{w}$. To conclude the proof, we show that $\check{F}$ is flat. For this purpose, we consider the tines $t_1^{y \lhd}$ and $t_2^{y \lhd}$. As mentioned above, they share no edges in $F^{y \lhd}$, and hence the prefixes $\check{t_1}$ and $\check{t_2}$ (of $t_1^{y \lhd}$ and $t_2^{y \lhd}$) appearing in $\check{F}$ share no edges. We wish to see that these prefixes

have maximum length in $\check{F}$, in which case $\check{F}$ is flat, as desired. This is immediate for the tine $\check{t_1}$ because all labels of $t_1^{y\lhd}$ are drawn from $\check{w}$ and, considering (3), its depth is at least that of all relevant honest vertices. As for $\check{t_2}$, observe that if $\ell(t_2)$ is not honest then $\beta > \ell(t_2)$ so that, as with $\check{t_1}$, the tine $\check{t_2}$ is labeled by $\check{w}$ so that the same argument, relying on (3), ensures that $\check{t_2}$ has length at least that of all relevant honest vertices. If $\ell(t_2)$ is honest, $\beta = \ell(t_2)$, and the terminal vertex of $t_2^{y\lhd}$ does not appear in $\check{F}$ (as it does not index $\check{w}$). In this case, however, $\text{length}(t_2^{y\lhd}) > \mathbf{d}(h)$ for any honest index of $\check{w}$, and it follows that $\text{length}(\check{t_2}) = \text{length}(t_2^{y\lhd}) - 1$ is at least the depth of any honest index of $\check{w}$, as desired. $\square$