On Finality in Blockchains

Emmanuelle Anceaume, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni

On Finality in Blockchains

ANONYMOUS AUTHOR(S)

This paper focuses on blockchain finality, which refers to the time when it becomes impossible to remove a block that has previously been appended to the blockchain. Blockchain finality can be deterministic or probabilistic, immediate or eventual. To favor availability against consistency in the face of partitions, most blockchains only offer probabilistic eventual finality: blocks may be revoked after being appended to the blockchain, yet with decreasing probability as they sink deeper into the chain. Other blockchains favor consistency by leveraging the immediate finality of Consensus – a block appended is never revoked – at the cost of additional synchronization.

In this paper, we focus on necessary and sufficient conditions to implement a blockchain with deterministic eventual finality, which ensures that selected main chains at different processes share a common increasing prefix. This is a much weaker form of finality that allows us to provide a solution in an asynchronous system subject to unlimited number of Byzantine failures. We also show that the classical selection mechanism, such as in Bitcoin, that appends blocks at the longest chain is not compliant with a solution to eventual finality. We then study stronger forms of eventual finality strengthened with bounded revocation. By bounded revocation we mean that the number of blocks that can be revoked from the current blockchain is bounded. This problem reduces to consensus or eventual consensus depending on whether the bound is known or not. We finally provide the first solution for unknown bounded revocation eventual finality with f<n/2 Byzantine processes.

1 INTRODUCTION

This paper focuses on blockchain finality, which refers to the time when it becomes impossible to remove a block that has previously been appended to the blockchain. Blockchain finality can be deterministic or probabilistic, immediate or eventual. Informally, immediate finality guarantees, as its name suggests, that when a block is appended to a local copy, it is immediately finalized and thus will never be revoked in the future. Most of the permissionned blockchains satisfy the deterministic form of immediate consistency, including Red Belly blockchain [8] and Hyperledger Fabric blockchain [3], while the probabilistic form of immediate consistency is typically achieved by permissionless pure proof-of-stake blockchains such as Algorand [7]. Designing blockchains with immediate finality favors consistency against availability in presence of transient partitions of the system. It leverages the properties of Consensus (i.e a decision value is unique and agreed by everyone), at the cost of synchronization constraints. On the other hand eventual finality ensures that all local copies of the blockchain share a common increasing prefix, and thus finality of their blocks increases as more blocks are appended to the blockchain. The majority of cryptoassets blockchains, with Bitcoin [20] and Ethereum [26] as celebrated examples, guarantee eventual finality with some probability: blocks may be revoked after being appended to the blockchain, yet with decreasing probability as they sink deeper into the chain. More recently, a huge effort has been devoted to propose alternatives to the energy-wasting proof-of-work method of Bitcoin and Ethereum. These proposals (e.g. [12, 15, 16, 21]), often based on a proof-of-stake method, offer a form of delayed finality not yet formalised in distributed computing terms.

Formalization of blockchains in the lens of distributed computing has been recognized as an extremely important topic [14]. Garay et al. [10] have been the first to analyse the Bitcoin backbone protocol and to define invariants this protocol has to satisfy to verify with high probability an eventual consistent prefix, i.e. probabilistic eventual finality. The authors analyzed the protocol in a synchronous system, while Pass et al. [22] extended this line of work considering a more adversarial network. Anta et al. [4] proposed a formalization of distributed ledgers modeled as an ordered list of records along with implementations for sequential consistency and linearizability using a total order broadcast

abstraction. Not related to the blockchain data structure, authors of [13] formalized the notion of cryptocurrency showing that Consensus is not needed.

While probabilistic eventual finality has been widely studied in the context of Bitcoin [5, 10, 22], only a few studies have started to lay the foundations of the computation power of blockchains with deterministic eventual finality consistency. Anceaume et al. [2] have been the first to propose a formal blockchain specification as a composition of abstract data types all together with a hierarchy of consistency criteria. This work captured the convergence process of two distinct classes of blockchain systems: the class providing strong prefix (for each pair of chains returned at two different processes, one is the prefix of the other) and the class providing eventual prefix, in which multiple chains can co-exist but the common prefix eventually converges. Interestingly, the authors of [2] show that to solve strong prefix, Consensus is needed, however the work does not address solvability of eventual prefix, which is the focus of this paper.

The objective of this paper is thus to push further this line of inquiry by presenting an in-depth study of deterministic eventual and immediate finality. We reinvestigate in Section 2 the definition of (deterministic) eventual prefix consistency presented in [2] to fit a broader context in which an infinite number of blocks are appended to the blockchain. We introduce the notion of bounded revocation, which informally says that the number of blocks that can be pruned from the current blockchain is bounded. Providing solutions that guarantee a known and bounded revocation reveals to be an important crux in the construction of blockchains. Specifically, in Section 3 we show that known bounded revocation eventual finality, that is eventual finality guaranteeing a known and bounded revocation, is equivalent to consensus. We also demonstrate that unknown bounded revocation eventual finality, that is eventual finality guaranteeing a unknown but bounded revocation, is equivalent to eventual immediate finality. Finally we show that eventual immediate finality is not weaker than eventual consensus, an abstraction that captures eventual agreement among all participants. Then in Section 4 we provide an algorithm that guarantees eventual finality in an asynchronous environment with an unbounded number of Byzantine processes. We are not aware of any such solution in the literature. We also show that it is impossible to build a blockchain that guarantees eventual finality when the rule to select a chain out of multiple ones, i.e. after a fork, is the longest chain rule. Note that such a selection function is used by many cryptoassets blockchains, from proof-of-work to proof-of-stake solutions ([12, 15, 16, 20, 21, 26] to name a few). Finally, we discuss impossibilities and possibilities of unbounded revocation eventual finality. In particular, we propose an algorithm that solves unknown bounded revocation eventual finality in an eventually synchronous environment in presence of less than a majority of Byzantine processes. Once again, we are not aware of any such solution in the literature. To summarize, we close the gap between eventual finality and (eventual) immediate finality and we established clear boundaries for the solvability of eventual finality.

2 DEFINITIONS

2.1 Preliminary Definitions

Similarly to [2], we describe a blockchain object as an abstract data type which allows us to completely characterize a blockchain by the operations it exports [18]. The basic idea underlying the use of abstract data types is to specify shared objects using two complementary facets: a sequential specification that describes the semantics of the object, and a consistency criterion over concurrent histories, i.e. the set of admissible executions in a concurrent environment [23]. Prior to presenting the blockchain abstract data type we first recall the formalization used to describe an abstract data type (ADT).

Abstract data types. An abstract data type (*ADT*) is a tuple of the form $T = (A, B, Z, z_0, \tau, \delta)$. Here A and B are countable sets called the *inputs* and *outputs*. Z is a countable set of abstract object *states*, $z_0 \in Z$ being the initial state of the object. The map $\tau : Z \times A \rightarrow Z$ is the *transition function*, specifying the effect of an input on the object state and the map $\delta : Z \times A \rightarrow B$ is the *output function*, specifying the output returned for a given input and an object local state. An input represents an operation with its parameters, where *(i)* the operation can have a side-effect that changes the abstract state according to transition function τ and *(ii)* the operation can return values taken in the output B, which depends on the state in which it is called and the output function δ.

Concurrent histories of an ADT. Concurrent histories are defined considering asymmetric event structures, i.e., partial order relations among events executed by different processes.

Definition 1. **(Concurrent history H [2])** *The execution of a program that uses an abstract data type $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ defines a concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$, where*

- $\Sigma = A \cup (A \times B)$ *is a countable set of operations;*
- E *is a countable set of events that contains all the ADT operations invocations and all ADT operation response events;*
- $\Lambda : E \rightarrow \Sigma$ *is a function which associates events to the operations in* Σ*;*
- $\mapsto$*: is the process order, irreflexive order over the events of* E*. Two events* $(e, e') \in E^2$ *are ordered by* $\mapsto$ *if they are produced by the same process,* $e \neq e'$ *and* e *happens before* e'*, that is denoted as* $e \mapsto e'$*.*
- $\prec$*: is the operation order, irreflexive order over the events of* E*. For each couple* $(e, e') \in E^2$ *if* e' *is the invocation of an operation occurred at time* t' *and* e *is the response of another operation occurred at time* t *with* $t < t'$ *then* $e \prec e'$*;*
- $\nearrow$*: is the program order, irreflexive order over* E*, for each couple* $(e, e') \in E^2$ *with* $e \neq e'$ *if* $e \mapsto e'$ *or* $e \prec e'$ *then* $e \nearrow e'$*.*

2.2 The blockchain ADT

Following [2], we represent a blockchain structure as a a tree of blocks. Indeed, while consensus-based blockchains prevent forks or branching in the tree of blocks, most blockchain systems based on proof-of-work allow the occurrence of forks to happen hence presenting blocks under a tree structure. The blockchain object is thus defined as a blocktree abstract data type (Blocktree ADT). However, we modify the original version of the blocktree ADT proposed in [2] so that distinct selection functions $f_r()$ and $f_a()$ for respectively the read() and append() operations exist. As will be shown latter in the paper, by relying on different functions to select the blockchain to be returned by a read() operation and to select the chain of the blocktree to which the new block is appended, this allows us to present the first algorithm that constructs a blockchain satisfying deterministic eventual finality in an asynchronous system in presence of an unbounded number of Byzantine processes.

2.2.1 Sequential Specification. A blocktree data structure is a directed rooted tree $bt = (V_{bt}, E_{bt})$ where V_{bt} represents a set of blocks and E_{bt} a set of edges such that each block has a single path towards the root of the tree b_0 called the genesis block. Let $\mathcal{BT}$ be the set of blocktrees, $\mathcal{B}$ be the countable and non empty set of uniquely identified blocks and let $\mathcal{BC}$ be the countable non empty set of blockchains, where a blockchain is a path from a leaf of bt to b_0. A blockchain is denoted by bc. The structure is equipped with two operations append() and read(). Operation append(b) adds the block $b \notin bt$ to V_{bt} and adds the edge (b, b') to E_{bt} where $b' \in V_{bt}$ is returned by the append selection function $f_a()$ applied to bt. Operation read() returns the chain bc selected by the read selection function $f_r()$ applied to bt. In the following the chain bc returned by a read() operation r is denoted by r/bc. Only blocks satisfying some validity

predicate P can be appended to the tree. The definition of the Blocktree ADT (BT-ADT) amended with the two types of selection functions is as follows.

Definition 2. *(Blocktree ADT) The Blocktree Abstract Data Type is the* 6*-tuple* $\mathrm{BT-ADT}=\{A=\{append(b), read()/bc \in \mathcal{BC}\}, B=\mathcal{BC}\cup\{\top,\bot\}, Z=\mathcal{BT}, \xi_0=b_0, \tau, \delta\}$*, where the transition function* $\tau: Z\times A\rightarrow Z$ *is defined by*

$$\tau(bt, read()) = bt$$

$$\tau(bt, append(b)) = \begin{cases} (V_{bt}\cup\{b\}, E_{bt}\cup\{b, last\_block(f_a(bt))\}) \text{ if } P(f_a(bt)^\frown b) \\ bt \text{ otherwise,} \end{cases}$$

and where the output function $\delta: Z\times A\rightarrow B$ *is defined by*

$$\delta(bt, read()) = f_r(bt)$$

$$\delta(bt, append(b)) = \begin{cases} \top \text{ if } P(f_a(bt)^\frown b) \\ \bot \text{ otherwise.} \end{cases}$$

The read selection $f_r()$ takes as argument the blocktree and returns a chain of blocks, that is a sequence of blocks starting from the genesis block to a leaf block of the blocktree. The chain returned by the read() operation is called the blockchain. The append selection function $f_a()$ takes as argument the blocktree and returns a chain of blocks. Function *last_block*() takes as argument a chain of blocks and returns the last appended block of the chain.

Predicate P is an application-dependent predicate used to verify the validity of the chain obtained by appending the new block b to the chain returned by $f_a()$ (denoted by $f_a(bt)^\frown b$). In Bitcoin for instance this predicate embeds the logic to verify that the obtained chain does not contain double spending or overspending transactions.

Note that we do not need to add the validity check during the read operation in the sequential specification, because in absence of concurrency the validity check during the append operation is enough.

2.2.2 Concurrent Specification and Consistency Criteria. The concurrent specification of a blocktree abstract data type is the set of concurrent histories. A blocktree consistency criterion is a function that returns the set of concurrent histories admissible for a blocktree abstract data type. For ease of readability, we provide the notations that have been introduced for most of them in [2]:

- $E(a^*, r^*)$ is an infinite set containing an infinite number of append() and read() invocation and response events;
- $E(a, r^*)$ is an infinite set containing *(i)* a finite number of append() invocation and response events and *(ii)* an infinite number of read() invocation and response events;
- o_{inv} and o_{rsp} indicate respectively the invocation and response event of an operation o; and in particular for the read() operation, r_{rsp}/bc denotes the returned blockchain bc associated with the response event r_{rsp} and for the append() operation $a_{inv}((b))$ denotes the invocation of the append operation having b as input parameter;
- length : $\mathcal{BC}\rightarrow\mathbb{N}$ denotes a monotonic increasing deterministic function that takes as input a blockchain bc and returns a natural number as length of bc. Increasing monotonicity means that $\mathsf{length}(bc^\frown\{b\}) > \mathsf{length}(bc)$;
- $bc\sqsubseteq bc'$ iff bc prefixes bc'.
- $bc[i]$ refers to the i-th block of blockchain bc.

We define three consistency criteria for the blocktree: the *BT eventual finality*, the *BT immediate finality* and *BT eventual immediate finality*. As previously indicated in the introduction, the eventual finality extends the eventual prefix

defined in [2] to an infinite number of append operations, while the BT eventual immediate finality, introduced in the present paper, will be shown not to be weaker than the eventual consensus abstraction [9].

Definition 3 (BT Eventual Finality Consistency criterion (EF)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$, of a system that uses a BT-ADT, verifies the BT eventual finality consistency criterion if the following four properties hold:*

- **Chain validity:**
  $\forall r_{rsp} \in E, P(r_{rsp}/bc)$.
  Each returned chain is valid.
- **Chain integrity:**
  $\forall r_{rsp} \in E, \forall b \in r_{rsp}/bc : b \neq b_0, \exists a_{inv}(b) \in E, a_{inv}(b) \nearrow r_{rsp}$.
  If a block different from the genesis block is returned, then an append *operation has been invoked with this block as parameter. This property is to avoid the situation in which* reads *return blocks never* appended.
- **Eventual prefix:**
  $\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, \forall i \in \mathbb{N} : bc[i] \neq \bot, \exists r'_{rsp}, \forall r''_{rsp} : r'_{rsp} \nearrow r''_{rsp}, ((r'_{rsp}/bc)[i] = (r''_{rsp}/bc)[i])$.
  In all the histories in which the number of read *invocations is infinite, then for any non empty read chain position i, there exists a* read r'/bc' *from which all the subsequent* reads r''/bc'' *will return the same block at position i, i.e.* $bc'[i] = bc''[i]$.
- **Ever growing tree:**
  $\forall E \in E(a^*, r^*), \forall k \in \mathbb{N}, \exists r \in E : \mathsf{length}(r_{rsp}/bc) > k$.
  In all the histories in which the number of append *and* read *invocations is infinite, for each length k, there exists a* read *that returns a chain with length greater than k. This property avoids the trivial scenario in which the length of the chain remains unchanged despite the occurrence of an infinite number of* append *operations. This can happen for instance if the tree is built as a star with infinite branches of bounded length.*

Definition 4 (BT Immediate Finality Consistency criterion (IF)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT immediate finality consistency criterion if chain validity, chain integrity, ever growing tree (as defined for EF) and the following property hold:*

- **Strong prefix:**
  $\forall r_{rsp}, r'_{rsp} \in E^2, (r'_{rsp}/bc' \sqsubseteq r_{rsp}/bc) \vee (r_{rsp}/bc \sqsubseteq r'_{rsp}/bc')$.
  For each pair of returned blockchains, one blockchain is the prefix of the other.

Definition 5 (BT Eventual Immediate Finality Consistency criterion (EIF)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT eventual immediate finality consistency criterion if chain validity, chain integrity, ever growing tree (as defined for EF) and the following property hold:*

- **Eventual strong prefix:**
  $\forall E \in E(a, r^*) \cup E(a^*, r^*), \exists r_{rsp} \in E, \forall r'_{rsp}, r''_{rsp} \in E^2 : r_{rsp} \nearrow r'_{rsp} \wedge r_{rsp} \nearrow r''_{rsp}, (r''_{rsp}/bc' \sqsubseteq r'_{rsp}/bc) \vee (r'_{rsp}/bc \sqsubseteq r''_{rsp}/bc')$.
  In all histories with an infinite number of reads*, there exists a* read *r from which for each pair of returned blockchains, one blockchain is the prefix of the other.*

Bounded revocation. Informally, bounded revocation says that for any two reads r/bc and r'/bc' such that r precedes r', then by pruning the last ℓ blocks from bc the obtained chain is a prefix of bc'. Note that constant ℓ can be initially known or not.

Definition 6. *Bounded revocation*

- $\exists \ell \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq \mathsf{length}(r_{rsp}/bc) - \ell, (r_{rsp}/bc)[i] = (r'_{rsp}/bc')[i]$.

Notation 1. For readability reasons, in the following we will simply say *finality* instead of *finality consistency criterion*, i.e., eventual finality consistency criterion will be replaced by eventual finality, and (eventual) immediate finality consistency criterion will be replaced by (eventual) immediate finality.

We will show in the following that satisfying eventual finality plus the bounded revocation property with known ℓ boils down to getting immediate finality. This is because from any algorithm $\mathcal{P}_{\mathcal{EP}}$ implementing eventual finality plus the known and bounded revocation property, a read that returns the blockchain that is returned by a read provided by $\mathcal{P}_{\mathcal{EP}}$ except for the last ℓ blocks guarantees the strong prefix property. Eventual finality plus known bounded revocation is designated in the sequel by *known bounded revocation eventual finality*. Furthermore, eventual finality plus the bounded revocation property with unknown ℓ boils down to get eventual immediate finality. Indeed as shown later, take an algorithm $\mathcal{P}_{\mathcal{EP}}$ implementing eventual finality plus the unknown but bounded revocation property. A read that returns half of the blockchain returned by a read provided by $\mathcal{P}_{\mathcal{EP}}$ guarantees eventual immediate finality since chains are always growing, and thus the number of removed blocks increases up to reaching ℓ. In the sequel, eventual finality plus the unknown bounded revocation is designated by *unknown bounded revocation eventual finality*.

In the following section we prove the above-mentioned equivalences more formally and study relationships to known problems such as consensus and eventual consensus to determine the assumptions on the system needed to implement the three blocktree consistency models.

3 (EVENTUAL) CONSENSUS REDUCTIONS

In this section we investigate the impact of the bounded revocation property on the construction of a bt-tree satisfying eventual finality. In particular, we show that when the bound ℓ is known, this problem is equivalent to the consensus abstraction, while when unknown, this problem is not weaker than the eventual consensus abstraction [9].

3.1 Known Bounded Revocation and Consensus

Theorem 1. *Known bounded revocation eventual finality is equivalent to Consensus.*

Proof. We first show how to solve immediate finality given a solution $\mathcal{P}_{\mathcal{EF}}$ for known bounded revocation eventual finality and then the reciprocal direction. Indeed, the equivalence between immediate finality and consensus is known from [1]. So let us show that we can solve immediate finality using $\mathcal{P}_{\mathcal{EF}}$. To do so, we consider the following transformation from the protocol $\mathcal{P}_{\mathcal{EF}}$. To make an append() operation, processes simply use the append() operation provided by $\mathcal{P}_{\mathcal{EF}}$. But, for the the read() operation, processes use the read() operation provided by $\mathcal{P}_{\mathcal{EF}}$ to obtain a chain and prune the last ℓ blocks from it before returning the remaining chain. Note that if there are less than ℓ blocks, processes then return the genesis block.

Let us show that this modified protocol solves immediate finality. For this, we need to show that the following properties are satisfied:

- **Chain validity:** The chain validity property is still satisfied by pruning the last ℓ blocks.
- **Chain integrity:** The chain integrity property is still satisfied by pruning the last ℓ blocks.

- **Strong prefix:** The strong prefix property follows from the known bounded revocation property and the removal of the last ℓ blocks. Indeed, if we remove the last ℓ blocks, then for any two read() operations, then the first read() returns a prefix of the second read() operation.
- **Ever growing tree:** The ever growing tree property is still satisfied by pruning the last ℓ blocks.

For the other direction, we can build a solution to known bounded revocation eventual finality using a solution for immediate finality. This trivially solves the known bounded revocation eventual finality with $\ell = 0$. □

If we consider known bounded revocation eventual finality then we have immediate finality (we can read a chain and remove the last ℓ blocks to be sure that what we read is final). Thus, since immediate finality requires consensus and consensus is impossible to deterministically solve in an asynchronous system (with a single crash failure) then, known bounded revocation eventual finality is impossible to deterministically solve in an asynchronous system.

3.2 Unknown Bounded Revocation and Eventual Consensus

If we have the unknown bounded revocation property then eventual finality combined with this property is not weaker than eventual consensus. To show that, we first show its equivalence with eventual immediate finality. Later we recall the eventual consensus problem (with a small modification of the validity property to make it suitable to the blockchain context) and show that eventual immediate finality is not weaker than eventual consensus.

Theorem 2. *Unknown bounded revocation eventual finality is equivalent to eventual immediate finality.*

Proof. Let $\mathcal{P}_{\mathcal{EF}}$ be a protocol solving the unknown bounded revocation eventual finality and let us show that we can solve eventual immediate finality. To do so, we consider the following modification to the protocol $\mathcal{P}_{\mathcal{EF}}$. To make an append() operation, processes simply use the append() operation provided by $\mathcal{P}_{\mathcal{EF}}$. But, for a read() operation, processes use the read() operation provided by $\mathcal{P}_{\mathcal{EF}}$ to obtain a chain and prune the second half of the returned chain before returning the remaining half of the chain.

Let us show that this modified protocol solves eventual immediate finality. For this, we need to show that the following properties are satisfied:

- **Chain validity:** The chain validity property is still satisfied by pruning half of the chain.
- **Chain integrity:** The chain integrity property is still satisfied by pruning half of the chain.
- **Eventual strong prefix:** The eventual strong prefix property follows from the unknown bounded revocation property and the removal of the second half of the chain. Indeed, if we remove the second half of the chain, then eventually for any two read() operations, then the first read() returns a prefix of the second read() operation. Indeed, since we remove a growing number of blocks, eventually we remove at least ℓ blocks and obtain chains such that one is the prefix of the other.
- **Ever growing tree:** The ever growing tree property is still satisfied by pruning half of the chain.

For the other direction, let us consider a protocol $\mathcal{P}_{\mathcal{EIF}}$ solving the eventual immediate finality and let us show that it solves the unknown bounded revocation eventual finality. The property of eventual strong prefix property clearly implies the eventual prefix property. Let $\text{revocation}(b_1, b_2)$ be the function that takes two blockchains b_1 and b_2 and returns the number of blocks needed to prune b_1 to obtain a chain b'_1 such that $b'_1 \sqsubseteq b_2$. Let us show that $\exists \ell \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E^2, r \nearrow r', \text{revocation}(r_{rsp}/bc, r'_{rsp}/bc) < \ell$. Assume by contradiction that this inequality is not satisfied, then it implies that for any ℓ, there exists a couple of reads with a greater revocation than ℓ. This implies that the eventual strong prefix property is not satisfied, which leads to a contradiction. Hence eventual immediate finality

implies unknown bounded revocation eventual finality. Putting all together, we have shown that eventual immediate finality is equivalent to unknown bounded revocation eventual finality. □

The eventual consensus (EC) abstraction [9] captures eventual agreement among all participants. It exports, to every process p_i, operations proposeEC$_1$, proposeEC$_2$, ... that take multi-valued arguments (correct processes propose valid values) and return multi-valued responses. Assuming that, for all $j \in \mathbb{N}$, every process invokes proposeEC$_j$ as soon as it returns a response to proposeEC$_{j-1}$, the abstraction guarantees that, in every admissible run, there exists $k \in \mathbb{N}$ and a predicate P_{EC}, such that the following properties are satisfied:

- **EC-Termination.** Every correct process eventually returns a response to proposeEC$_j$ for all $j \in \mathbb{N}$.
- **EC-Integrity.** No process responds twice to proposeEC$_j$ for all $j \in \mathbb{N}$.
- **EC-Validity.** Every value returned to proposeEC$_j$ is valid with respect to predicate P_{EC}.
- **EC-Agreement.** No two correct processes return different values to proposeEC$_j$ for all $j \geq k$.

Theorem 3. *Eventual immediate finality is not weaker than eventual consensus.*

Proof. We show that it exists a protocol $\mathcal{P}_{\mathcal{EC}}$ to solve eventual consensus starting from a protocol $\mathcal{P}_{\mathcal{EIF}}$ that solves eventual immediate finality. We do the transformation as follows. Every correct process p invokes proposeEC$_j$ for all $j \in \mathbb{N}$. We impose that the validity predicate P of the blocktree ADT (see Section 2) be equal to predicate P_{EC}. When a correct process p invokes the proposeEC$_j(v)$ operation of $\mathcal{P}_{\mathcal{EC}}$, for any $j \in \mathbb{N}$, then p executes the following sequence of three steps: *(i)* it invokes the append(v) operation of $\mathcal{P}_{\mathcal{EIF}}$, then *(ii)* it invokes a sequence of read() operations up to the moment the read() returns a chain bc such that $bc[j] \neq \bot$, and finally *(iii)* p returns chain bc as decision for proposeEC$_j(v)$ and triggers the next operation proposeEC$_{j+1}(v')$.

Let us show that protocol $\mathcal{P}_{\mathcal{EC}}$ solves eventual consensus.

- **EC-Termination** This property is guaranteed by the ever growing tree property.
- **EC-Integrity** This property follows directly from the transformation.
- **EC-Validity** This property follows by construction and the chain validity property, since $P = P_{EC}$.
- **EC-Agreement** This property follows by the eventual strong prefix property, which guarantees that there exists a read() operation r such that, all the subsequent ones return blockchains that are each prefix of the following one. In other words, eventually there is agreement on the value contained in $bc[j]$. This implies that there exists k for which all proposeEC$_j$ with $j \geq k$ return the same value to all correct processes.

□

4 EVENTUAL FINALITY SOLUTIONS

In this section we first show the impossibility of solving eventual finality when the append operation, in case of forks, selects the longest chain. We then provide the first solution to eventual finality with an unbounded number of Byzantine processes using an alternative selection rule.

4.1 Impossibility of Eventual Finality with the Longest Chain Rule

In the following we prove that we cannot provide eventual finality if, in case of forks, the append selection function $f_a()$ follows the longest chain rule, i.e., returns the longest chain of the bt-tree. To show this impossibility, we consider a scenario in which the occurrence of any fork produces at most two alternative chains (this is often referred to as a branching factor of 2). We consider a finite number of processes and an append selection function f_a that in case of

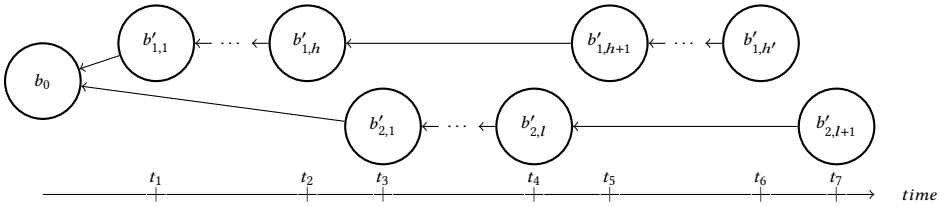

Fig. 1. A blocktree generated by two processes. On the x-axis the longest chain value of each chain at different time instants (from the root to the current leaf) and the relationships between those values.

forks deterministically selects the longest chain, i.e., the chain with the largest number of blocks (the length is thus a monotonically increasing function on prefixes), and in case of a tie selects the chain whose last block is the smallest (in the lexicographical order). We show that it is impossible to guarantee eventual finality for this append selection function f_a. Note that such a selection function is used by many blockchain systems. In proof-of-work systems such as Bitcoin, chains are selected as the chain with the greater number of blocks while in Ethereum chains are selected using the chain with greatest weight, both captured by the selection of chains according to the longest chain. In proof-of-stake systems like EOS [12] or Tezos [11] the same rule is also applied.

Intuitively, the impossibility follows from the fact that with the longest chain selection, races can occur between different branches in the tree. We show that as forks may occur, we can create two infinite branches sharing only the root. One or the other branch constitutes alternatively the longest chain and append operations select chains from each branch alternatively. This is enough to show that the only common prefix that is returned is the root hence, violating eventual finality.

Obviously, this impossibility result holds only when blocks are not created by running a consensus algorithm. When consensus is employed, immediate finality can be assured, and no fork will ever occur. In this case the append operation will return the longest chain by default.

Theorem 4. *It is impossible to guarantee eventual finality if the* append *operation is based on the longest chain rule.*

Proof. The proof is constructed by resorting to the computation model proposed in [1], in which the branching factor is regulated by a fictional abstraction, called oracle. The oracle grants access to the blocktree as a shared object, through the following three operations: update_view() returns the current state of the blocktree; getValidBlock(b_i, b_j) returns a valid block b'_j, constructed from b_j, that can be appended to block b_i, where b_i is already included in the blocktree; and setValidBlock(b_i, b'_j) appends the valid block b'_j to b_i, and returns $\top$ when the block is successfully appended and $\bot$ otherwise.[1] Note that, in this shared object model processes are asynchronous.

In this computational model, oracles are the only generators of valid blocks and regulate the number of appended children from a same parent. It follows that oracles own a synchronization power, which is equal to Consensus if the branching factor is equal to 1 [1]. In the proof we consider the stronger oracle allowing the occurrence of forks, i.e., an oracle with branching factor equal to 2. That is, this oracle allows for two valid blocks to be appended to the same parent, afterwards, it shall return $\bot$ to all requests.

[1]In [1] getValidBlock() and setValidBlock() functions were called differently and owned a slightly different semantics, here we opted for a simplified interface introduced in [25].

Let p_1 and p_2 be two processes trying to append infinitely many blocks. Note that we carry out this proof with general indexes for the use of a length function that increases monotonically, but first instantiate them by using a length equal to the number of blocks for illustration purposes.

At time t_0, for both p_1 and p_2, the update_view() of bt equals b_0, thus when both apply the append selection function f_a on it to select the leaf where to append the new block, they both get b_0. Then they both call getValidBlock$(b_0, b_{i,1}) = b'_i$, where $i = 1$ for p_1 and $i = 2$ for p_2. At time $t_1 > t_0$, p_1 and p_2 are poised to call setValidBlock$(b_0, b'_{i,1})$. We then let p_1 call setValidBlock$(b_0, b'_{1,1})$, which must return $\top$ and hence $b'_{1,1}$ is appended to b_0. Process p_1 then proceeds to append a new block $b_{1,2}$, i.e., after having updated its bt's view, through the update_view() function, p_1 applies the append selection function f_a on it to select the leaf where to append its new block, in this case the only leaf is $b'_{1,1}$. It calls getValidBlock$(b'_{1,1}, b_{1,2})$ function which returns $\{b'_{1,2}\}$ and it is poised to call setValidBlock$(b'_{1,1}, b'_{1,2})$.

We let p_1 continue to append new blocks until some time t_2 at which it is poised to call setValidBlock$(b'_{1,h}, b'_{1,h+1})$, with $h = 1$, such that the length of the chain $b_0, \ldots, b'_{1,h+1}$ would be greater than or would have the same length but a smaller lexicographical order than the chain $b_0, b'_{2,1}$ if $b'_{2,1}$ were already appended to block b_0. Afterwards, at time $t_3 \geq t_2$, we let p_2 resume and complete its call to setValidBlock$(b_0, b'_{2,1})$ which must also succeed and return $\top$ as our oracle has a branching factor of 2. By construction, p_2 sees the two branches in its following update_view() of bt (i.e., chain $b_0, b'_{1,h}$ with $h = 1$, and chain $b_0, b'_{2,1}$) of the same length thus the selection function f_a selects the branch $b_0, b'_{2,1}$ for where to append the next block as block $b'_{2,1}$ is smaller than $b'_{1,h}$ in the lexicographical order. We let p_2 append blocks to this branch until some time t_4 at which it becomes poised to call setValidBlock$(b'_{2,\ell}, b'_{2,\ell+1})$ with $\ell = 2$ such that the length of the chain $b_0, \ldots, b'_{2,\ell}$ is smaller than the chain $b_0, \ldots, b'_{1,h+1}$, or in case of equal length has a higher lexicographical order, and such that the length of the chain $b_0, \ldots, b'_{2,\ell+1}$ is greater than the chain $b_0, \ldots, b'_{1,h+1}$, or in case of equal length has a smaller lexicographical order.

As before, it is time to stop the execution of p_2 and resume the execution of p_1 and to let it complete its call to setValidBlock$(b'_{1,h}, b'_{1,h+1})$. We can continue to create two infinite branches sharing only the root by alternatively letting p_1 and p_2 extend their own branch while stopping one and resuming the execution of the other each time its length would overcome the length of the other branch extended with the pending block (and the appropriate lexicographical orderings in case of equal length). This way we construct a tree composed of two infinite branches sharing only the root b_0 as common prefix. It is easy to see that we can integrate read operations that may return the current chain from any branch as both branches are temporarily the longest one. Thus, the common prefix never increases, and so, the eventual finality consistency criteria is not satisfied.

It is important to note that with any length function that increases monotonically with prefixes (e.g, the length function could count the total number of transactions that belong to the blocks on the same branch) then this scenario still holds. In that case h and ℓ in the proof could be larger than 1 and 2 respectively. □

4.2 Asynchronous Solution to Eventual Finality with an Unbounded Number of Byzantine Processes

We consider an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and processes can be affected by Byzantine failures. Each process has a unique identifier $i \in \mathbb{N}$ and is equipped with signatures that can be used to identify the message sender identifier. Each block is identified with the identifier of the process that created it. Block identifier is inserted in the header of the block. Moreover, each process is equipped with an Eventual BFT-Reliable Broadcast primitive. We assume the system is such that we can implement an eventual reliable broadcast primitive, e.g., we assume that the infinite set of processes are arranged in a topology in which for each pair of correct processes, there exists a path composed by only correct processes [19]. Moreover, as proved in [1] reliable

communications are necessary for eventual finality. We show that in that setting it is possible to build a blockchain that satisfies eventual prefix consistency.

Algorithm 1: Eventual prefix with an unbounded number of Byzantine processes

```
1 upon rb-delivery(bc)
2 |  bt.addIfValid(bc)
3 end
4 upon append(b)
5 |  rb-broadcast(f_a(bt)⌢b)
6 end
7 upon read()
8 |  return f_r(bt)
9 end
```

The main idea of Algorithm 1 consists in using local selection functions f_a and f_r for append and read operations respectively and characterizing blocks by their parent and the producer signature. Let us first describe the append() and read() operations first and the selection function after.

To perform an append() operation of a block b, processes extend the chain returned by function f_a applied on their current view of bt with b, i.e., $f_a(bt)^\frown b$, and rb-broadcast $f_a(bt)^\frown b$. When a process rb-delivers a blockchain bc, it calls bt.addIfValid(bc) that merges bc with bt if the former is valid. By merging bc with bt we mean that for each block b_i of bc starting from the genesis block b_0, if b_i is not present in bt then b_i is added to bt, i.e., b_i is appended to the block of bt whose hash is equal to the one contained in b_i's header. For read() operations, processes return the chain selected by f_r on their current bt.

Given a blocktree bt, the append selection function f_a selects a chain in bt by going from the root (i.e., genesis block) to a leaf, choosing at each fork b_i the edge to the child with the lowest identifier. If more than one child have the same identifier (i.e., they have been created by the same process), then all of them are ignored. If all the children have the same identifier, then f_a returns the chain from the genesis block to b_i. Blocks are ranked by the creator identifier, in the domain of the natural number and thus lower bounded by 0. Then even though at a fork infinitely many blocks can be continuously being added, then given a block, there does not exist infinitely many blocks with a smaller identifier. Thus eventually the selection function f_a will always select the same prefix. Finally, since blocks are diffused by a rb-broadcast primitive, eventually all correct processes will have the same view of the blocktree. When a process invokes the read() operation, it returns the blockchain selected by the read selection function f_r applied to its current view of the blocktree. By imposing that $f_r = f_a$, then eventually all the processes, when reading, will select the same prefix.

Theorem 5. *Algorithm 1 is a solution for eventual finality in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and suffer from an unbounded number of Byzantine failures.*

Proof. We show by construction that Algorithm 1 solves eventual finality in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and can suffer an unbounded number of Byzantine failures. Intuitively, despite the unbounded number of blocks in each fork, by the eventual reliable broadcast, eventually for each fork all correct processes have the same block with the smallest identifier. Hence, by the read selection function that at each fork selects the block with the smallest identifier in order to select the chain to read, eventually, at all correct processes, function f_r returns the blockchain having a common increasing prefix. Let $p_1, p_2, \ldots,$ be a possibly

infinite set of processes, such that each one maintains its local view bt_i of blocktree bt by running Algorithm 1. Then for any correct process p_i the following properties hold.

- **Chain validity:** it is satisfied by function bt.addIfValid(bc) that merges blockchain bc to bt_i only if the former is valid.
- **Chain integrity:** The read() operation returns the chain of blocks selected by function f_r applied to bt_i. By Line 2 of Algorithm 1, only valid blocks are locally added to bt_i once they have been reliably delivered. By Algorithm 1, the only place at which blocks are reliably broadcast is in the append() operation.
- **Eventual prefix:** The eventual prefix property follows from the definition of f_a. For any b in bt, let t_b be the time after which no process contributes to the same block b. All correct processes trigger the append operation on chains containing the non-ignored child with the longest chain. By induction, it creates an ever-growing prefix selected by f_a. As read chains are selected according to the read selection function, it provides the eventual prefix property.
- **Ever growing tree:** The ever-growing tree property relies on the fact that each fork has a finite number of blocks since there are finitely many processes and each (Byzantine or correct) process can contribute with at most one block per parent as multiple children created by the same process are ignored by f_a. Thus, eventually, new blocks contribute to the growth of the tree.

□

4.3 Impossibilities and Possibilities of Unknown Bounded Revocation Eventual Finality

In this section we discuss the impossibilities and possibilities of unknown bounded revocation eventual prefix consistency. In particular, we show that it is unsolvable in an asynchronous shared-memory or message-passing system if at least one process fails with Byzantine failures. Interestingly, if we consider the previous result with known bounded revocation, the impossibility holds even in an eventually synchronous system model with more than one third of Byzantine processes (with not surprise, given the equivalence with the Consensus problem, see Theorem 1). Finally we prove that unknown bounded revocation eventual prefix consistency is solvable in an eventual synchronous message-passing system with a majority of correct processes.

Theorem 6. *There does not exist any solution that solves unknown bounded revocation eventual finality in an asynchronous system with at least one Byzantine process.*

Proof. The proof follows from the relationship between the unknown bounded revocation eventual finality which is not weaker than the eventual consensus problem (cf. Theorem 3), which is equivalent to the leader election problem [9] which cannot be solved in an asynchronous system with at least one Byzantine processes [24]. □

Theorem 7. *There does not exist any solution that solves known bounded revocation eventual finality in an eventual synchronous system with more than one third of Byzantine faulty processes.*

Proof. The proof follows from the equivalence between known bounded revocation eventual finality and Consensus (cf. Theorem 1), which is unsolvable in a synchronous (and thus also in an eventually synchronous) system with more than one third of Byzantine faulty process [17]. □

Theorem 8. *There exists a solution that solves unknown bounded revocation eventual finality in an eventual synchronous system with a majority of correct processes.*

In order to prove the existence of a solution, we start from an existing solution, Streamlet [6], that guarantees strong prefix consistency under the assumption of less than a third of byzantine processes and eventual synchrony with a known message delay Δ. We weaken both of these assumptions to provide a solution to the eventual immediate finality. In particular, we assume only a majority of correct processes, we do not explicitly use Δ and consider a slightly modified version of the protocol. In the following we first describe Streamlet and then discuss the proposed modifications before providing the proof.

Streamlet protocol. The Streamlet protocol works in a eventually synchronous system with a known message delay Δ and a finite set of n processes. In particular, before the Global Stabilisation Time (GST), message delays can be arbitrary; however, after GST, messages sent by correct processes are guaranteed to be received by correct processes within Δ time. The following assumption holds: Δ-bounded assumption during periods of synchrony, when an honest node sends a message at time t, an honest recipient is guaranteed to receive it by time $max(GST, t + \Delta)$. [2]

In Streamlet [6], each epoch, composed of 2Δ, has a designated leader chosen at random by a publicly known hash function. The protocol works as follows:

- **Propose-Vote.** In every epoch:
  - The epoch's designated leader proposes a new block extending from the longest notarized chain it has seen (if there are multiple, break ties arbitrarily). The notion "notarized" is defined below.
  - Every process votes for the first proposal they see from the epoch's leader, as long as the proposed block extends from (one of) the longest notarized chain(s) that the voter has seen. A vote is a signature on the proposed block.
  - When a block gains votes from at least 2n/3 distinct processes, it becomes notarized. A chain is notarized if its constituent blocks are all notarized.
- **Finalize.** Notarized does not mean final. If in any notarized chain, there are three adjacent blocks with consecutive epoch numbers, the prefix of the chain up to the second of the three blocks is considered final. When a block becomes final, all of its prefix must be final too.

We propose protocol $\mathcal{S}$ with the following modifications to Streamlet. First, we only require that a block gains votes from a majority of distinct processes to become notarized, which means that forks can occur. The second modification goes deeper: if a fork occurs, then it is possible to detect Byzantine processes and to exclude them from the voters. This is done as follows. When, two conflicting chains are finalized, that is two finalized chains that are not the prefix of one another, then processes look for inconsistent blocks. That is, two notarized blocks b, b' are inconsistent with one another if one of the following two conditions hold:

- Cond. 1 b and b' share the same epoch.
- Cond. 2 b (resp. b') that corresponds to a higher epoch, $b.epoch < b'.epoch$ (resp. $b'.epoch < b.epoch$), with a strictly smaller height, $b'.height < b.height$ (resp. $b.height < b'.height$), than the other block. Function height counts the number of blocks from the genesis block.

If a process votes for blocks inconsistent with one another then it is detected as Byzantine.

Proof. Let us show that $\mathcal{S}$ is a solution for eventual immediate finality. Let us first show that when a fork occurs, then we detect at least a Byzantine process. For this let us show that it implies the existence of two inconsistent blocks

[2] Notice that, in Streamlet [6] there is not the notion of time but of round, which denotes a basic unit of time.

and that the intersection of voters of the two blocks is not empty as two majority intersect. In the following we show that voting for two inconsistent blocks is a Byzantine failure.

Let us first show that voting for two inconsistent blocks b and b' is a Byzantine failure. If the two blocks are inconsistent for Cond. 1, then the intersecting voters are Byzantine as correct processes vote only once per epoch. It follows that if a process q votes for b and b' then q is Byzantine. If the two blocks are inconsistent for Cond. 2, then the intersecting voters are Byzantine as correct processes vote only for blocks extending one of the longest notarized chains. That is, if a correct process p votes for b it means that b is extending a notarized block b_{pred} that is of height $b.height - 1$, therefore p cannot vote for a block b' later on with a height strictly smaller than $b.height$ because it needs to extend one of the longest notarized chain. It follows that if a process q votes for b and b' then q is Byzantine.

Let us now show that when a fork occurs we must have two inconsistent blocks. Indeed, if there is a fork then we have two sequences of three adjacent blocks with consecutive epochs, b_1, b_2, b_3 and b'_1, b'_2, b'_3 (by construction, given the finalization rule). If no blocks share the same epoch number then we can assume w.l.o.g. that $b_3.epoch < b'_1.epoch$. Let block b' be the block with the smallest height that is in the prefix of b'_3 such that it has a greater epoch than b_1 (such block always exists as b'_1 satisfies those conditions). Either $b'.height < b_3.height$ meaning that b' is inconsistent with b_3 or else, $b'.height \geq b_3.height$ meaning that the predecessor of b' is inconsistent with b_1. Indeed, the predecessor of b' has a strictly smaller height than b_1 and by assumption has a smaller epoch number than b_1. Hence there is always a couple of inconsistent blocks in a fork.

Let us now conclude our proof that we solve the eventual immediate finality. If a fork occurs, then each correct process eventually detects at least one Byzantine process and ignore its votes, hence, we have a finite number of forks as we have a finite number of Byzantine processes, hence eventually there is always a single chain that is finalized. As there is a majority of correct processes, we remain live as in the original Streamlet Protocol. We also inherit its properties for finalizing blocks eventually when synchrony is reached. □

5 CONCLUSION

In this work we have focused on necessary and sufficient conditions to implement a blockchain with deterministic eventual finality, which ensures that selected main chains at different processes share a common increasing prefix. We have firstly addressed the question: “can we design an asynchronous deterministic blockchain solution to solve bounded revocation eventual finality ?". We have formally showed that the answer is “no". On the positive side, we have provided for the first time (i) a solution to eventual immediate finality with a majority of correct processes and (ii) an asynchronous solution to eventual finality with an unlimited number of Byzantine processes.

REFERENCES

[1] Anceaume, E., Pozzo, A. D., Ludinard, R., Potop-Butucaru, M., and Tucci Piergiovanni, S. Blockchain abstract data type. CoRR (2018).

[2] Anceaume, E., Pozzo, A. D., Ludinard, R., Potop-Butucaru, M., and Tucci Piergiovanni, S. Blockchain abstract data type. In Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2019).

[3] Androulaki, E., and et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the European Conference on Computer Systems (EuroSys) (2018).

[4] Anta, A. F., Konwar, K., Georgiou, C., and Nicolaou, N. Formalizing and implementing distributed ledger objects. ACM SIGACT News 49, 2 (2018), 58–76.

[5] Cachin, C. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS) (2016).

[6] Chan, B. Y., and Shi, E. Streamlet: Textbook streamlined blockchains. https://eprint.iacr.org/2020/088.pdf, 2020.

[7] Chen, J., and Micali, S. Algorand: A secure and efficient distributed ledger. Theor. Comput. Sci. (2019).

[8] Crain, T., Gramoli, V., Larrea, M., and Raynal, M. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. CoRR abs/1702.03068 (2017).

[9] Dubois, S., Guerraoui, R., Kuznetsov, P., Petit, F., and Sens, P. The weakest failure detector for eventual consistency. In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) (2015).

[10] Garay, J. A., Kiayias, A., and Leonardos, N. The bitcoin backbone protocol: Analysis and applications. In Proc. EUROCRYPT International Conference (2015).

[11] Goodman, L. Tezos – a self-amending crypto-ledger, 2014.

[12] Grigg, I. EOS: An introduction. https://whitepaperdatabase.com/eos-whitepaper/.

[13] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., and Seredinschi, D.-A. The consensus number of a cryptocurrency. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC) (2019).

[14] Herlihy, M. Blockchains and the future of distributed computing. In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) (2017).

[15] Kiayias, A., Russell, A., David, B., and Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Proceedings of the Advances in Cryptology (2017).

[16] Koltsov, A., Cheremensky, V., and Kapulkin, S. Casper White Paper.

[17] Lamport, L., Shostak, R., and Pease, M. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (1982).

[18] Liskov, B., and Zilles, S. Programming with abstract data types. ACM SIGLAN Notices 9, 4 (1974).

[19] Maurer, A., and Tixeuil, S. On byzantine broadcast in loosely connected networks. In Proceedings of the 26th International Symposium on Distributed Computing (DISC) (2012).

[20] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org (2008).

[21] Nomadic Labs. Analysis of Emmy$^+$. https://blog.nomadic-labs.com/analysis-of-emmy.html, 2019.

[22] Pass, R., Seeman, L., and Shelat, A. Analysis of the blockchain protocol in asynchronous networks. In Proceedings of the EUROCRYPT International Conference (2017).

[23] Perrin, M. Distributed Systems, Concurrency and Consistency. ISTE Press, Elsevier, 2017.

[24] Raynal, M. Eventual leader service in unreliable asynchronous systems: Why? how? In Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA) (2007).

[25] Tucci Piergiovanni, S. Invited paper: On the characterization of blockchain consensus under incentives. In Proceedings of the International conference on Stabilization, Safety, and Security of Distributed Systems (SSS) (2019), Springer.

[26] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf.