HAL archives-ouvertes.fr

On finality in blockchains

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

To cite this version:

Emmanuelle Anceaume, Antonella Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni. On finality in blockchains. 2021. cea-03080029v4

HAL Id: cea-03080029

https://hal-cea.archives-ouvertes.fr/cea-03080029v4

Preprint submitted on 9 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

On Finality in Blockchains

Emmanuelle Anceaume @ H ORCID
CNRS, Inria, Université Rennes, IRISA, Campus de beaulieu, 35042 Rennes, France

Antonella Del Pozzo @
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Thibault Rieutord @
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Sara Tucci-Piergiovanni @
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

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.

The quest for "good" deterministic finality properties for blockchains is still in its infancy, though. Our motivation is to provide a thorough study of several possible deterministic finality properties and explore their solvability. This is achieved by introducing the notion of bounded revocation, which informally says that the number of blocks that can be revoked from the current blockchain is bounded. Based on the requirements we impose on this revocation number, we provide reductions between different forms of eventual finality, Consensus and Eventual Consensus. From these reductions, we show some related impossibility results in presence of Byzantine processes, and provide non-trivial results. In particular, we provide an algorithm that solves a weak form of eventual finality in an asynchronous system in presence of an unbounded number of Byzantine processes. We also provide an algorithm that solves eventual finality with a bounded revocation number in an eventually synchronous environment in presence of less than half of Byzantine processes. The simplicity of the arguments should better guide blockchain designs and link them to clear formal properties of finality.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Blockchain, consistency properties, Byzantine tolerant implementations

1 Introduction

This paper focuses on blockchain finality, which refers to the time when it becomes impossible to remove a block previously 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. 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. Assuming partially synchronous environments, most of the permissioned blockchains satisfy the deterministic form of immediate consistency, as for example Red Belly blockchain [8] and Hyperledger Fabric blockchain [2]. The probabilistic form of immediate finality is typically achieved by permissionless pure proof-of-stake blockchains such as Algorand [7].

Unlike immediate finality, eventual finality only ensures that eventually 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 [25] 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. In an effort to replace the energy-wasting pow method of Bitcoin and Ethereum, recent proof-of-stake blockchains such as e.g. [16, 12, 15] emerged. These blockchains offer as well a form of eventual finality. More broadly, all these permission-less solutions favor availability (or progress) only relying on a broadcast primitive to diffuse blocks and a local reconciliation mechanism to select a unique chain. It is indeed admitted that a blockchain may lose consistency by incurring a fork, which is the presence of multiple chains at different processes. The reconciliation mechanism, available to recover from a fork, consists in a local deterministic rule selecting a chain among the different possible alternatives. In Bitcoin for instance any participant reconciles the state following the "longest" chain rule (the term "longest" chain rule is commonly employed, but this is actually the one that required the most work to be built). Once a winner chain is chosen, the other alternatives are revoked, as such all the blocks belonging to them. In these designs, however, network effects make the moment at which all honest processes observe the same set of candidate chains unknown. Reconciliation and finalisation guarantees are then unsure, or simply extremely inefficient, for example by considering a block as finalised after one or more days. To solve this problem a number of permissionless blockchain projects are investigating how to add "finality gadgets" (e.g., [5, 24]) to proof-of-work or proof-of-stake blockchains, which means seeking additional mechanisms or protocols to reach "better" finality properties in network adversarial settings. The hope is to find ways to get deterministic finality by periodically running finality gadgets. For the time being, the only way that has been concretely pursued is to resort to Byzantine Consensus – e.g. Tenderbake [4] adds Byzantine Consensus to the existing proof-of-stake method assuring deterministic finality to each block followed by other two blocks. How to add mechanisms that do not resort to Consensus, however, is an intriguing and open question, related to the finality properties one would like to guarantee.

The quest for "good" deterministic finality properties for blockchains is still in its infancy, though. Our motivation is to provide a protocol-independent abstraction of several possible finality properties to study their solvability. To this aim we formalise, for the first time, the notion of finality in a protocol-agnostic way. At the heart of the proposed formalisation lies the notion of *bounded revocation*. Bounded revocation informally says that the number of blocks that can be revoked from the current blockchain is bounded. Providing solutions that guarantee deterministic bounded revocation reveals to be an important crux in the construction of blockchains. We thus provide rigorous definitions for the weakest form of eventual finality $\mathcal{EF}^\star$, which does not guarantee any bound on the number of blocks that can be revoked from the blockchain at any given fork, and two stronger forms: $\mathcal{EF}^{\Diamond c}$, in which revocation is bounded but unknown, and $\mathcal{EF}^c$, in which revocation is bounded and known. Intuitively, if $\mathcal{EF}^c$ holds, processes revoke at most a constant number c of blocks from the current blockchain at each reconciliation, while if $\mathcal{EF}^{\Diamond c}$ holds, processes revoke at most a constant number of c blocks from the current chain only eventually, i.e., after a finite but unknown number of reconciliations. The rigorous formalisation of these properties enable us to easily show that solutions that guarantee $\mathcal{EF}^c$ are equivalent to Consensus, while solutions that guarantee $\mathcal{EF}^{\Diamond c}$ are not weaker than Eventual Consensus, an abstraction that captures eventual agreement among all participants. From these reductions, we show some related impossibility results in presence of Byzantine processes. Beside reductions and related impossibilities, we propose the following non-trivial results:

- $\mathcal{EF}^\star$ cannot be achieved in an asynchronous system if the reconciliation rule follows the "longest" chain rule (Theorem 13). This implies that the reconciliation rule, used in current blockchains to provide probabilistic finality in synchronous settings, cannot guarantee that participants will eventually converge to a stable prefix of the chain in asynchronous settings.
- A solution that guarantees $\mathcal{EF}^\star$ in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks. This novel solution is strikingly simple and tolerant to an unbounded number of Byzantine processes (Theorem 14).
- A solution that solves $\mathcal{EF}^{\Diamond c}$ in an eventually synchronous environment in presence of less than half of Byzantine processes (Theorem 15). The central point of our solution is to let correct processes blame each fork on a particular Byzantine process, which can then be excluded from the computation. Weakening the classic requirement of $< 1/3$ to $< 1/2$ Byzantine processes makes such a solution well adapted to large scale adversarial systems. As for the previous one, we are not aware of any such solution in the literature.

We hope that these results will better guide blockchain designs and link them to clear formal properties of finality. Hence, in the remainder of this article, Section 2 situates our work with respect to similar ones. Section 3 formally presents the sequential specification of a blockchain and the formalisation of the different finality properties we may expect from a blockchain when concurrently accessed. Section 4 presents reductions between different forms of finality, Consensus and Eventual Consensus. Section 5 first shows why $\mathcal{EF}^\star$ is not solvable in an asynchronous environment when the "heaviest" chain rule is used, and then presents two original and surprisingly simple algorithms that respectively solve $\mathcal{EF}^\star$ and $\mathcal{EF}^{\Diamond c}$. Finally, Section 6 concludes.

2 Related Work

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 analyze 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 have analyzed the protocol in a synchronous system, while others, as for example Pass et al. [21], have extended this line of work considering a more adversarial network. In those works the specification of the consistency properties are protocol dependent and thus provide an abstraction level that does not allow us to model the blockchain as a shared object being agnostic of the way it is implemented. The objective we pursue throughout this work is to formalize the semantic of the interface between the blockchain and the users. To do so we consider the blockchain as a shared object, and thus the consistency properties are specified independently of the synchrony assumptions of underlying distributed system and the type of failures that may occur. By doing this, we offer a higher level of abstraction than well-known properties do.

This approach has been recently followed in particular by Anta et al. [3], Anceaume et al. [1] and Guerraoui et al. [13] [1]. In Anta et al. [3], the authors propose 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. Anceaume

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

et al. [1] have captured the convergence process of two distinct classes of blockchain systems: the class providing strong prefix as [3] (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. The authors of [1] show that to solve strong prefix the Consensus abstraction is needed, however they do not address solvability of eventual prefix and do not formalise finality. Interestingly, our notion of finality and bounded revocation is able to encompass the strong and the eventual prefix consistency properties of [1].

3 Definitions

3.1 Preliminary Definitions

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 [22]. Prior to presenting the blockchain abstract data type we first recall the formalization used to describe an abstract data type (ADT).

3.1.1 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 respectively called *input alphabet* and *output alphabet*. Z is a countable set of abstract object *states* and $z_0 \in Z$ is the initial abstract state. The map $\tau : Z \times A \to Z$ is the *transition function*, specifying the effect of an input on the object state and the map $\delta : Z \times A \to B$ 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.

3.1.2 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)** *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;*
- $\Lambda : E \to \Sigma$ *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* $e \nearrow e'$.

3.2 The blocktree ADT

We represent a blockchain as a tree of blocks. The same representation has been adopted in [1]. Indeed, while consensus-based blockchains prevent forks or branching in the tree of blocks, 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).

3.2.1 Sequential Specification of the Blocktree ADT (BT-ADT)

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. A branching in the tree is called a *fork*. 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 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 (note that in [1], the read() and append() operations are defined with a unique selection function). 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 bc returned by a read() operation r is called the blockchain, and is denoted by r/bc. 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. Only blocks satisfying some validity predicate P can be appended to the tree. 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. Formally,

▶ **Definition 2.** *(Sequential specification of the 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* $\tau : Z \times A \to Z$ *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 $\delta : Z \times A \to B$ *is defined by*

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

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

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

3.2.2 Concurrent Specification and Consistency Criteria of the BlockTree ADT

The concurrent specification of the blocktree abstract data type is the set of its concurrent histories. A blocktree consistency criterion is a function that returns the set of concurrent histories admissible for the blocktree abstract data type. In this paper, we define three consistency criteria for the blocktree, i.e. the *BT eventual finality (EF)*, the *BT immediate finality (IF)* and *BT eventual immediate finality (EIF)*, and the notion of block revocation. This family of consistency criteria combined with the revocation notion provides a comprehensive characterization of what we may expect from blockchains.

▶ **Notation 1.**
- $E(a^*, r^*)$ *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;*
- $\mathsf{length} : \mathcal{BC} \to \mathbb{N}$ *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* $\mathsf{length}(bc^\frown\{b\}) > \mathsf{length}(bc)$*;*
- *We represent chain bc as an infinite list* $b_0 b^* \bot^+$ *of blocks, where the first block* $bc[0] = b_0$*, the genesis block, followed by block values b, and an infinite number of* $\bot$ *values. Notation* $bc[i]$ *refers to the i-th block of blockchain bc. Note that the special "*$\bot$*" symbol counts for zero for the* length *function.*
- $bc \sqsubseteq bc'$ *if and only if bc prefixes bc′. The operator* $\sqsubseteq$ *ignores all the records set to* $\bot$*.*

▶ **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 validility:**
  $\forall r_{rsp} \in E, P(r_{rsp}/bc)$.
  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 at 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 (i.e., tree built as a star with infinite branches of bounded length). Specifically the "Ever growing tree" property imposes that in presence of an infinite number of* read *and* append *operations, for any natural number i, there will always exist a* read *operation that will return a chain of at least length i. Note that the well known "Chain Growth Property" [10, 21] states that each (honest) chain grows proportionally with the number of rounds of the protocol, which in contrast to our specification, makes it protocol dependent.*

▶ **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:** $\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, r'_{rsp}/bc', \forall i \in \mathbb{N} : bc[i] \neq \bot,$
  - if $r_{rsp} \mapsto r'_{rsp}$ *then* $\forall j \leq i, bc'[j] = bc[j] \neq \bot$
  - *otherwise*
    - *either* $bc'[i] \neq \bot$ *and* $\forall j \leq i, bc'[j] = bc[j] \neq \bot$
    - *or* $bc'[i] = \bot$ *and* $|\{r''/bc'' : bc''[i] = \bot\}| < \infty$

  In all the histories in which the number of read *invocations is infinite, and for any couple of* read *operations r and r' that return respectively bc and bc', such that for any position i in bc, $bc[i]$ contains a value then, if both r and r' occur at the same process p and r occurs before r', then bc is a prefix of bc'. Otherwise (r and r' do not occur on the same process) either bc' contains the same value as the one contained in bc or contains $\bot$. In that case, the overall number of* read *operations that return $\bot$ at position i is finite.*

▶ **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:** $\forall E \in E(a, r^*) \cup E(a^*, r^*), \exists \ell \in \mathbb{N}, \forall r_{rsp}/bc, r'_{rsp}/bc', \forall i \geq \ell \in \mathbb{N} : bc[i] \neq \bot,$
  - if $r_{rsp} \mapsto r'_{rsp}$ *then* $\forall j \leq i, bc'[j] = bc[j] \neq \bot$
  - *otherwise*
    - *either* $bc'[i] \neq \bot$ *and* $\forall j \leq i, bc'[j] = bc[j] \neq \bot$
    - *or* $bc'[i] = \bot$ *and* $|\{r''/bc'' : bc''[i] = \bot\}| < \infty$

  There exists a natural number ℓ, such that once the length of the chain is equal to ℓ, then the strong prefix property holds. Notice that, when the length of the chain is smaller than ℓ then values it contains may change from one read to another.

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 c blocks from bc the obtained chain is a prefix of bc'. Note that constant c can be initially known or not.

▶ **Definition 6** (c-Bounded Revocation).
$\exists c \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq (\mathsf{length}(r_{rsp}/bc) - c)$ then $(r_{rsp}/bc)[i] = (r'_{rsp}/bc')[i]$.

Note that the c-Bounded Revocation is not protocol dependent in contrast to the well-known "Common-Prefix Property" [10, 21], which states that for any two rounds r and r' of the protocol with $r < r'$, the (honest) chain read at round r from which the last c blocks have been pruned is a prefix of (resp. is equal to with high probability) the one read at round r'.

▶ **Notation 2.** 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 can now define the $c-$Bounded Eventual Finality criteria by augmenting the previous consistency criteria with the Bounded revocation property:

▶ **Definition 7.** *$c-$Bounded Eventual Finality criteria*
- $\mathcal{EF}^{\star} = EF$, *in this case the revocation is unbounded.*
- $\mathcal{EF}^{c} = EIF$ *combined with $c-$Bounded revocation, such that c is known a priori.*
- $\mathcal{EF}^{\Diamond c} = EIF$ *combined with $c-$Bounded revocation where c is unknown but bounded.*

4 (Eventual) Consensus Reductions

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

4.1 Known Bounded Revocation and Consensus

▶ **Theorem 8.** *$\mathcal{EF}^{c}$ is equivalent to Consensus.*

Proof. We first show how to solve immediate finality (IF) given a solution $\mathcal{P}$ for $\mathcal{EF}^{c}$ 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 $\mathcal{P}$. To do so, we consider the following transformation from the protocol $\mathcal{P}$. To make an append() operation, processes simply use the append() operation provided by $\mathcal{P}$. But, for the the read() operation, processes use the read() operation provided by $\mathcal{P}$ to obtain a chain and prune the last c blocks from it before returning the remaining chain. Note that if there are less than c 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 c blocks.
- **Chain integrity:** The chain integrity property is still satisfied by pruning the last c blocks.
- **Strong prefix:** The strong prefix property follows from the known bounded revocation property and the removal of the last c blocks. Indeed, if we remove the last c 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 c blocks.

For the other direction, we can build a solution to $\mathcal{EF}^{c}$ using a solution for immediate finality. This trivially solves $\mathcal{EF}^{c}$ with $c = 0$. ◀

354 ▶ **Corollary 9.** *There does not exist any solution that solves $\mathcal{EF}^c$ in an eventual synchronous*
355 *system with more than $n/3$ Byzantine processes, where n is the number of processes partici-*
356 *pating to the algorithm.*

357 **Proof.** The proof follows from the equivalence between $\mathcal{EF}^c$ and Consensus (cf. Theorem 8),
358 which is unsolvable in a synchronous (and thus also in an eventually synchronous) system
359 with more than one third of Byzantine processes [17]. ◀

360 ## 4.2 Unknown Bounded Revocation and Eventual Consensus

361 In this section we show that $\mathcal{EF}^{\Diamond c}$ is not weaker than Eventual Consensus. We first show its
362 equivalence with eventual immediate finality (EIF). Later we recall the Eventual Consensus
363 problem with a small modification of the validity property to make it suitable to the blockchain
364 context and show that EIF is not weaker than Eventual Consensus.

365 ▶ **Theorem 10.** *$\mathcal{EF}^{\Diamond c}$ is equivalent to eventual immediate finality.*

366 **Proof.** Let $\mathcal{P}_1$ be a protocol solving $\mathcal{EF}^{\Diamond c}$, then by definition of $\mathcal{EF}^{\Diamond c}$, $\mathcal{P}_1$ trivially solves
367 eventual immediate finality.
368 For the other direction, let us consider a protocol $\mathcal{P}_2$ solving eventual immediate finality
369 and let us show that it solves $\mathcal{EF}^{\Diamond c}$. Eventual strong prefix property clearly implies the
370 eventual prefix property. Let $\mathsf{revocation}(b_1, b_2)$ be the function that takes two blockchains b_1
371 and b_2 and returns the number of blocks needed to prune b_1 in order to obtain a chain b'_1 such
372 that $b'_1 \sqsubseteq b_2$. Let us show that $\exists c \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E^2, r \nearrow r', \mathsf{revocation}(r_{rsp}/bc, r'_{rsp}/bc) <$
373 c. Assume by contradiction that this inequality is not satisfied. This implies that for any
374 c, there exists a couple of reads with a greater revocation than c, and thus the eventual
375 strong prefix property does not hold. A contradiction with the assumption. Hence eventual
376 immediate finality implies $\mathcal{EF}^{\Diamond c}$. Putting all together, we have shown that eventual immediate
377 finality is equivalent to $\mathcal{EF}^{\Diamond c}$. ◀

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

- 384 **EC-Termination.** Every correct process eventually returns a response to $\mathsf{proposeEC_j}$
385 for all $j \in \mathbb{N}$.
- 386 **EC-Integrity.** No process responds twice to $\mathsf{proposeEC_j}$ for all $j \in \mathbb{N}$.
- 387 **EC-Validity.** Every value returned to $\mathsf{proposeEC_j}$ is valid with respect to predicate P_{EC}.
- 388 **EC-Agreement.** No two correct processes return different values to $\mathsf{proposeEC_j}$ for all
389 $j \geq k$.

390 ▶ **Theorem 11.** *Eventual immediate finality is not weaker than Eventual Consensus.*

391 **Proof.** We show that there exists a protocol $\mathcal{P}_1$ that solves Eventual Consensus assuming the
392 existence of a protocol $\mathcal{P}_2$ that solves eventual immediate finality. We do the transformation
393 as follows. Every correct process p invokes $\mathsf{proposeEC_j}$ for all $j \in \mathbb{N}$. We impose that the
394 validity predicate P of the blocktree ADT (see Section 3) be equal to predicate P_{EC}. When
395 a correct process p invokes the $\mathsf{proposeEC_j}(v)$ operation of $\mathcal{P}_1$, for any $j \in \mathbb{N}$, then p executes
396 the following sequence of three steps: *(i)* p invokes the $\mathsf{append}(v)$ operation of $\mathcal{P}_2$, then *(ii)* p

invokes a sequence of read() operations up to the moment the read() returns a chain bc such that $bc[j] \neq \perp$, and finally *(iii)* p decides chain bc (i.e., it returns chain bc) and triggers the next operation $\mathsf{proposeEC}_{j+1}(v')$. We now show that protocol $\mathcal{P}_1$ 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 by the chain validity property since predicate P equals to predicate 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 $\mathsf{proposeEC}_j$ with $j \geq k$ return the same value to all correct processes.

◀

▶ **Theorem 12.** *There does not exist any solution that solves $\mathcal{EF}^{\Diamond c}$ in an asynchronous system with at least one Byzantine process.*

Proof. The proof follows from the relationship between the $\mathcal{EF}^{\Diamond c}$ and eventual immediate finality (EIF). EIF is not weaker than the Eventual Consensus problem (cf. Theorem 11), which is equivalent to the leader election problem [9], which cannot be solved in an asynchronous system with at least one Byzantine process [23]. ◀

5 Eventual Finality Solutions

In this section we first show the impossibility of solving $\mathcal{EF}^{\star}$ when the append operation, in case of forks, selects the "longest" chain. We then provide the first solution to $\mathcal{EF}^{\star}$ with an unbounded number of Byzantine processes using an alternative selection rule.

5.1 Impossibility of Eventual Finality with the Longest Chain Rule

In the following we prove that, in an asynchronous environment, we cannot provide $\mathcal{EF}^{\star}$ if, in case of forks, the append selection function $f_a()$ follows the longest chain rule, i.e., returns the longest chain of the blockchain tree. Obviously we assume that blocks are not created using the Consensus abstraction: With Consensus, immediate finality is easily ensured, and thus no fork will ever occur. Thus, when the Consensus abstraction cannot be implemented (d to the adversity of the environment), many blockchain systems adopt a selection function f_a based on the longest chain. For instance, in proof-of-work systems such as Bitcoin, selected chains are the ones that have required the most amount of work, which is equivalent to the longest chains when the difficulty is constant. In Ethereum, while the selection rule is based on heaviest sub-tree of the blockchain tree, or in proof-of-stake systems like EOS [12] or Tezos [11], the same argument applies.

To show this impossibility result, 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 forks deterministically selects the longest chain through the length function (see Section 3.2.2), and in case of a tie selects the chain following any deterministic rule (for instance the chain whose last block hast the smallest digest). We show that it is impossible to guarantee $\mathcal{EF}^{\star}$ for such append selection function f_a.

Intuitively, the impossibility follows from the fact that with the longest chain selection rule, 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.

To capture the synchronisation power of the system, we abstract the deterministic creation of new blocks and their addition to the blockchain within an oracle. This oracle is the only generator of valid blocks, and regulates the number of appended children from a same parent. The same approach has been proposed in [1]. The branching factor of an oracle is the maximal number of children that can be appended to a block. The oracle owns a synchronization power equal to Consensus if its branching factor is equal to 1. 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 $\perp$ otherwise. The following theorem shows that, even with this strong oracle (that allows to have a bounded branching factor in contrast to PoW approaches), we cannot reach eventual finality if we rely on the longest chain rule to resolve forks.

▶ **Theorem 13.** *It is impossible to guarantee $\mathcal{EF}^\star$ if the* append *operation is based on the longest chain rule in an asynchronous environment.*

Proof. The interested reader is invited to read the proof in the Appendix of this paper. ◀

5.2 Asynchronous Solution to $\mathcal{EF}^\star$ 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, since it has been proven that reliable communications are necessary to ensure eventual finality [1], we assume that each process is equipped with an Eventually Reliable Broadcast primitive that satisfies the following two properties: If a correct process p broadcasts a message m then p eventually delivers m and if a correct process p delivers m then all correct processes eventually deliver m. Such a primitive can be implemented by organizing the infinite set of processes in a topology in which for each pair of correct processes, there exists a path composed by only correct processes [19]. Thus, we do not require any assumptions on the proportion between Byzantine and correct processes in the system but on the way those processes are arranged on the network topology.

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 parents and producer signatures.

To perform an append operation of a block b, correct 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 added to the block of bt whose hash is equal to the one contained in b_i's header. A read()

Algorithm 1 $\mathcal{EF}^\star$ 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
```

487 operation triggered by a correct process p returns the chain selected by f_r on the current
488 blocktree bt of p. Given a blocktree bt, the append selection function f_a selects a chain in bt
489 by going from the root (i.e., genesis block) to a leaf, choosing at each fork b_i the edge to the
490 child with the lowest identifier. If more than one child have the same identifier (i.e., they
491 have been created by the same process), then all of them are ignored. If all the children have
492 the same identifier, then f_a returns the chain from the genesis block to b_i. Blocks are ranked
493 by the creator identifier, in the domain of the natural number and thus lower bounded by 0.
494 Then even though, an infinite number of blocks is added continuously to a fork, there is not,
495 for a given block, an infinite number of blocks with a smaller identifier. Thus eventually the
496 selection function f_a will always select the same prefix. Finally, since blocks are diffused by
497 an eventually reliable broadcast primitive, eventually all correct processes will have the same
498 view of the blocktree. When a process invokes the read() operation, it returns the blockchain
499 selected by the read selection function f_r applied to its current view of the blocktree. By
500 imposing that $f_r = f_a$, then eventually all the processes, when reading, will select the same
501 prefix.

502 ▶ **Theorem 14.** *Algorithm 1 is a solution for $\mathcal{EF}^\star$ in an asynchronous system with a possibly*
503 *infinite set of processes which can append infinitely many blocks, and suffer from an unbounded*
504 *number of Byzantine failures.*

505 **Proof.** We show by construction that Algorithm 1 solves $\mathcal{EF}^\star$ in an asynchronous system
506 with a possibly infinite set of processes which can append infinitely many blocks, and can
507 suffer an unbounded number of Byzantine failures. Intuitively, despite the unbounded number
508 of blocks in each fork, by the eventually reliable broadcast, eventually for each fork all correct
509 processes have the same block with the smallest identifier. Hence, by the read selection
510 function f_r that at each fork selects the block with the smallest identifier in order to select
511 the chain to return, eventually, at all correct processes, function f_r returns the blockchain
512 having a common increasing prefix. Let $p_1, p_2, \ldots,$ be a possibly infinite set of processes,
513 such that each one maintains its local view bt_i of blocktree bt by running Algorithm 1. Then
514 for any correct process p_i the following properties hold.

515 - **Chain validity:** it is satisfied by function bt.addIfValid(bc) that merges blockchain bc to
516 bt_i only if the former is valid.
517 - **Chain integrity:** The read() operation returns the chain of blocks selected by function
518 f_r applied to bt_i. By Line 2 of Algorithm 1, only valid blocks are locally added to bt_i
519 once they have been reliably delivered. By Algorithm 1, the only place at which blocks
520 are reliably broadcast is in the append() operation.
521 - **Eventual prefix:** This property follows from the definition of f_a and the eventually

reliable broadcast primitive. Thanks to the latter, for any b in the bt of a correct process p, eventually all correct processes deliver b. Let t_b be the time after which no process can append further blocks b_{child} to b such that b_{child} is part of the chain returned by f_a. This time t_b always exists, as for each block b having potentially infinitely many children we have, by definition of function f_a, that $f_a(bt)$ selects a chain in bt by going from the root to a leaf, choosing at each fork b the edge to the child with the lowest identifier. Since identifiers are lower bounded by 0, eventually function f_a will always select the same child b' of b. The same argument applies for b' and its children. Hence, if any two correct processes invoke the read operation infinitely many times, then as $f_r = f_a$, eventually they return chains that satisfy the eventual prefix property.

- **Ever growing tree:** This 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 tree growth. ◀

5.3 Eventually Synchronous Solution to $\mathcal{EF}^{\Diamond c}$ with less than half of Byzantine Processes

In this section we prove that $\mathcal{EF}^{\Diamond c}$ is solvable in an eventually synchronous message-passing system with less than $n/2$ Byzantine processes, where n is the number of processes.

We propose an algorithm, called $\mathcal{AF}$ for Accountable Forking. This algorithm is inspired by the Streamlet [6] algorithm. Streamlet [6] assumes the presence of less than a third of Byzantine processes and an eventually synchronous system with a known message delay Δ after GST. Algorithm $\mathcal{AF}$ relies on weaker assumptions: we assume the presence of only a majority of correct processes and we do not explicitly use bound Δ. We suppose that processes have access to the eventually reliable broadcast presented in Section 5.2. Prior to presenting our algorithm, let us first describe the original Streamlet.

The Streamlet Algorithm. The Streamlet algorithm works in an 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 units. Each epoch, composed of 2Δ time units, has a designated leader chosen at random by a publicly known hash function. Each block b is labelled with the epoch ($b.epoch$) at which it has been created. This allows processes to determine whether block b has been created by a legitimate leader. The protocol works as follows:

- **Propose-Vote.** In every epoch:
  - The epoch's designated leader proposes a new block and reliably broadcasts it, extending the longest notarized chain (defined below) it has seen, or breaking ties arbitrarily if they have the same height.
  - Each process votes (rb-broadcasts a vote) for the first proposal it sees from the epoch's leader, as long as the proposed block extends (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 prefixes must be final too.

569 **The Accountable Forking ($\mathcal{AF}$) Algorithm.** The $\mathcal{AF}$ algorithm extends Streamlet by
570 guaranteeing that for any given fork, correct processes can blame the process that originates
571 it, i.e, a Byzantine process creating a fork is accountable for it. This is achieved as follows:
572 First, we only require that a block gains votes from a majority of distinct processes to
573 become notarized, which means that forks can occur. The second modification we propose
574 goes deeper: if a fork occurs, any correct processes can detect the Byzantine process that
575 originated it, and excludes it from the voters. Specifically, when two conflicting chains are
576 finalized (i.e., two finalized chains that are not the prefix of one another) then processes look
577 for inconsistent blocks. By definition, two notarized blocks b, b' are inconsistent with one
578 another if one of the following two conditions holds:

579 - **Condition 1.** b and b' share the same epoch, i.e, $b.epoch = b'.epoch$;
580 - **Condition 2.** either $((b.epoch < b'.epoch)$ and $(b.height > b'.height))$ or $((b'.epoch <$
581 $b.epoch)$ and $(b'.height > b.height))$. Function height counts the number of blocks from
582 the genesis block.

583 If a process votes for blocks inconsistent with one another then it is detected as Byzantine.
584 Once a correct process p detects a Byzantine process q, p ignores all messages coming from
585 q. Since all messages received by a correct process q are eventually received by any correct
586 process, then all of them do the same with respect to q.

587 ▶ **Theorem 15.** *There exists a solution that solves eventual immediate finality in an eventually*
588 *synchronous system with less than $n/2$ Byzantine processes, where n is the number of processes*
589 *participating to the algorithm.*

590 **Proof.** We show that algorithm $\mathcal{AF}$ is such a solution.
591 Let us first demonstrate that voting for two inconsistent blocks b and b' is a Byzantine
592 failure. We have two cases to consider. If both b and b' are inconsistent because Condition 1
593 holds, then the intersecting voters are Byzantine as correct processes vote only once per epoch.
594 Hence if process q votes for b and b' then q is Byzantine. If both b and b' are inconsistent
595 because Condition 2 is met, then the intersecting voters are Byzantine, as correct processes
596 vote only for blocks extending one of the longest notarized chains. That is, if correct process
597 p votes for b it means that b is extending a notarized block b_{pred} that is of height $b.height - 1$,
598 therefore p cannot vote afterwards for a block b' whose height is strictly smaller than $b.height$
599 because p must extend one of the longest notarized chain. It follows that if process q votes
600 for both b and b' then q is Byzantine.
601 Let us now show that a fork occurs because of two inconsistent blocks. If there is a
602 fork then this gives rise to two sequences of three adjacent blocks with consecutive epochs,
603 b_1, b_2, b_3 and b'_1, b'_2, b'_3 (by construction given the finalization rule). If no blocks share the
604 same epoch number then we can assume w.l.o.g. that $b_3.epoch < b'_1.epoch$. Let block b'
605 belonging to the prefix of b'_3 such that $b'.epoch > b_1.epoch$ and $b'.height$ is the smallest in the
606 prefix of b'_3. Such block always exists as b'_1 satisfies those two conditions. We have two cases:
607 Either $b'.height < b_3.height$ or $b'.height \geq b_3.height$. In the former case, b' is inconsistent
608 with b_3 since by assumption $b'.epoch > b_3.epoch$. In the latter case, the predecessor of b'
609 is inconsistent with b_3. Indeed, the predecessor of b' has a strictly smaller height than b_1
610 and by assumption has a larger epoch number than b_3. Figure 1 illustrates the presence
611 of inconsistent blocks in presence of a fork at some block b_c. From b_c two chains are built,
612 the first one consisting of the sequence of three blocks b_1, b_2 and b_3, and the second chain
613 consisting of four consecutive blocks b_d, b'_1, b'_2, b'_3 (to illustrate the first case) and of five
614 consecutive blocks $b_d, b_e, b'_1, b'_2, b'_3$ (to illustrate the second case). In both cases block b'_1 plays
615 the role of block b'. In the first case (figure in the top), $b_3.height = 6$ and $b'.height = 5$ while

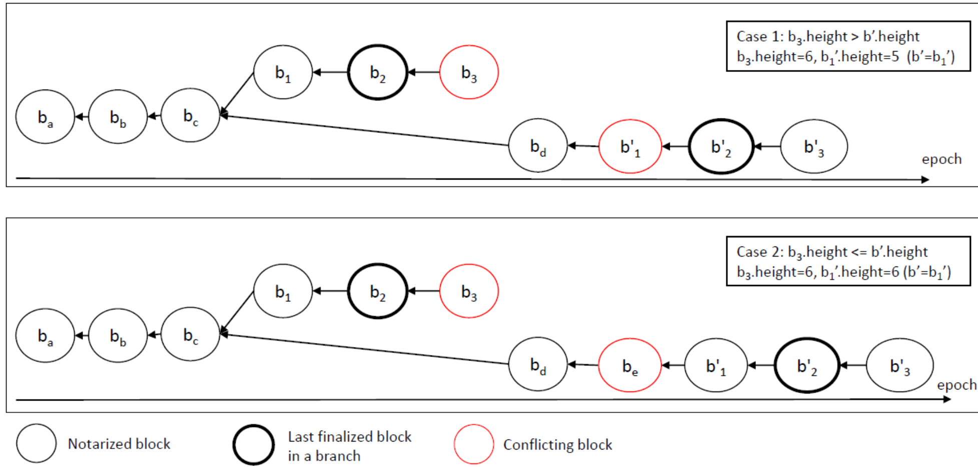

Figure 1 Illustration of block inconsistencies due to the occurrence of a fork when the finalized blocks are not labelled with the same epoch. Epochs are on the x axis, and all consecutive blocks have consecutive epochs, e.g., b_c and b_d have four epochs of difference, 4 and 7 respectively, while b_1 and b_2 are labelled with consecutive epochs.

616 $b_3.epoch = 6$ and $b'.height = 5$. Thus Condition 2 applies. In the second case (figure in
617 the bottom), since $b'.height \geq b3.height$ then there must exist some block b_e in the b' prefix.
618 Thus $b_e.height < b'.height$. Given that by assumption $b_e.epoch > b_3.epoch$, then Condition 2
619 holds for b_e and b_3. Hence there is always a couple of inconsistent blocks in a fork.

620 Let us now conclude our proof that protocol $\mathcal{AF}$ solves eventual immediate finality. If a
621 fork occurs, then each correct process eventually detects at least one Byzantine process and
622 ignores its votes. Thus, the number of forks is finite since we have a finite number of Byzantine
623 processes. As a consequence, there is always a single chain that is eventually finalized. As
624 there is a majority of correct processes, algorithm $\mathcal{AF}$ remains live as the original Streamlet
625 one. Algorithm $\mathcal{AF}$ also inherits the properties of the original Streamlet algorithm regarding
626 the eventual finalization of blocks when the system becomes synchronous. ◀

627 6 Conclusion

628 In this work we have focused on the formalisation of eventual finality, which ensures that
629 selected main chains at different processes share a common increasing prefix. We have
630 formalised different forms of eventual finality in terms of the maximal number of blocks
631 that can be revoked at each reconciliation, which is a crux in current blockchain designs.
632 We have formally shown that in an asynchronous system is not possible to reach a bound
633 on the number of blocks that can be revoked. On the other hand, we proposed for the
634 first time a solution in an eventually synchronous system with less than half of Byzantine
635 processes guaranteeing that such bound is reached eventually. We have also shown that in
636 an asynchronous system eventual finality with no bound on the number of revocable blocks
637 cannot be solved using the reconciliation rule of Bitcoin. Still we provide an asynchronous
638 solution with an unlimited number of Byzantine processes. We hope that this work will
639 better guide blockchain designs.

640 ## References

641 **1** E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci Piergiovanni.
642 Blockchain abstract data type. In *Proceedings of the ACM Symposium on Parallelism in*
643 *Algorithms and Architectures (SPAA)*, 2019.
644 **2** E. Androulaki and et al. Hyperledger fabric: a distributed operating system for permissioned
645 blockchains. In *Proceedings of the European Conference on Computer Systems (EuroSys)*,
646 2018.
647 **3** A. Anta Fernández, K. Konwar, C. Georgiou, and N. Nicolaou. Formalizing and implementing
648 distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.
649 **4** L. Aştefanoaei, P. Chambart, A. D. Pozzo, T. Rieutord, S. Tucci-Piergiovanni, and E. Zălinescu.
650 Tenderbake - a solution to dynamic repeated consensus for blockchains. In *Proceedings of the*
651 *Fourth International Symposium of Foundations and Applications of Blockchain*, 2021.
652 **5** V. Buterin and V. Griffith. Casper the friendly finality gadget. *CoRR*, 2017.
653 **6** B. Y. Chan and E. Shi. Streamlet: Textbook streamlined blockchains. `https://eprint.iacr.`
654 `org/2020/088.pdf`, 2020.
655 **7** J. Chen and S. Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput.*
656 *Sci.*, 2019.
657 **8** T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (leader/randomization/signature)-free
658 byzantine consensus for consortium blockchains. *CoRR*, abs/1702.03068, 2017.
659 **9** S. Dubois, R. Guerraoui, P. Kuznetsov, F. Petit, and P. Sens. The weakest failure detector
660 for eventual consistency. In *Proceedings of the ACM Symposium on Principles of Distributed*
661 *Computing (PODC)*, 2015.
662 **10** J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and
663 applications. In *Proc. EUROCRYPT International Conference*, 2015.
664 **11** L. Goodman. Tezos – a self-amending crypto-ledger, 2014.
665 **12** I. Grigg. EOS: An introduction. `https://whitepaperdatabase.com/eos-whitepaper/`.
666 **13** R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovič, and D.-A. Seredinschi. The consensus
667 number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of*
668 *Distributed Computing (PODC)*, 2019.
669 **14** M. Herlihy. Blockchains and the future of distributed computing. In *Proceedings of the ACM*
670 *Symposium on Principles of Distributed Computing (PODC)*, 2017.
671 **15** A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-
672 stake blockchain protocol. In *Proceedings of the Advances in Cryptology*, 2017.
673 **16** A. Koltsov, V. Cheremensky, and S. Kapulkin. Casper White Paper.
674 **17** L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions*
675 *on Programming Languages and Systems*, 1982.
676 **18** B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGLAN Notices*, 9(4),
677 1974.
678 **19** A. Maurer and S. Tixeuil. On byzantine broadcast in loosely connected networks. In *Proceedings*
679 *of the 26th International Symposium on Distributed Computing (DISC)*, 2012.
680 **20** S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *www.bitcoin.org*, 2008.
681 **21** R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous
682 networks. In *Proceedings of the EUROCRYPT International Conference*, 2017.
683 **22** M. Perrin. *Distributed Systems, Concurrency and Consistency.* ISTE Press, Elsevier, 2017.
684 **23** M. Raynal. Eventual leader service in unreliable asynchronous systems: Why? how? In
685 *Proceedings of the IEEE International Symposium on Network Computing and Applications*
686 *(NCA)*, 2007.
687 **24** A. Stewart. Poster: Grandpa finality gadget. In *Proceedings of the 2019 ACM SIGSAC*
688 *Conference on Computer and Communications Security*, CCS '19, page 2649–2651, 2019.
689 **25** G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.`
690 `com/Paper.pdf`.

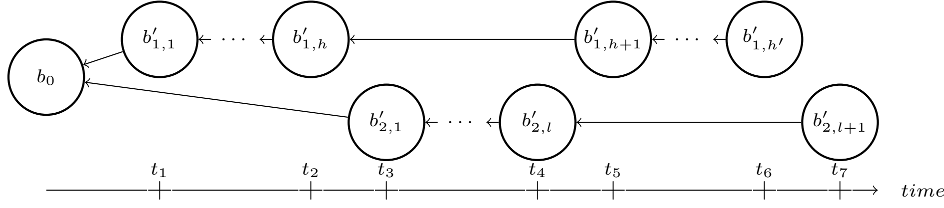

Figure 2 A blocktree generated by two processes. On the x-axis the longest chain value of each chain at different time instants (from the root to the current leaf) and the relationships between those values.

Appendix

Theorem 13 It is impossible to guarantee $\mathcal{EF}^\star$ if the append operation is based on the longest chain rule in an asynchronous environment.

Proof. In the proof we consider the stronger oracle allowing the occurrence of one fork, 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. If the oracle receives new requests to append additional blocks to this parent, it shall return $\bot$ to all such requests.

Let p_1 and p_2 be two processes trying to append infinitely many blocks. Without loss of generality, we carry out this proof with a length function that counts the number of blocks from the genesis block.

We illustrate our proof with Figure 2. 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$(b'_{2,c}, b'_{2,c+1})$ with $c = 2$ such that the length of the chain $b_0, \ldots, b'_{2,c}$ 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 $b_0, \ldots, b'_{2,c+1}$ 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

727 infinite branches sharing only the root by alternatively letting p_1 and p_2 extend their own
728 branch while stopping one and resuming the execution of the other each time its length
729 would overcome the length of the other branch extended with the pending block (and the
730 appropriate lexicographical orderings in case of equal length). This way we construct a tree
731 composed of two infinite branches sharing only the root b_0 as common prefix. It is easy to
732 see that we can integrate read operations that may return the current chain from any branch
733 as both branches are temporarily the longest one. Thus, the common prefix never increases,
734 and so, the eventual finality consistency criterion is not satisfied.

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