On Finality in Blockchains

Emmanuelle Anceaume
CNRS, Univ Rennes, Inria, IRISA, Rennes, France

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

Thibault Rieutord
CEA-List, Université Paris-Saclay, Palaiseau, France

Sara Tucci-Piergiovanni
CEA-List, Université Paris-Saclay, 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

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.6

Funding This work was partially supported by the French ANR project ByBloS (ANR-20-CE25-0002) devoted to the modular design of building blocks for large-scale fault-tolerant multi-users applications.

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 permissionless 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 proof-of-work (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 permissionless solutions favor availability (or progress) relying on a Nakamoto-style consensus: 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 designs using Nakamoto-style consensus, 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 uncertain, or simply extremely inefficient, for example by considering a block as finalised after one or more days. To solve this problem a number of projects are investigating how to add “finality gadgets” (e.g., [5, 24]) to Nakamoto-style 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 on top of Nakamoto-style consensus. 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 *revocation number*. Informally, given a system run and a blockchain bc read by a user at some time t, we call the revocation number the natural number n such that by pruning the last n blocks from bc, we obtain a prefix of any blockchain bc' read after t.

By leaving the revocation number unbounded in all the runs of the system, we formalise our weakest form of finality, the eventual finality consistency criterion $\mathcal{F}$: In each run, the revocation number can be infinite when the run goes to infinity, still each block will be eventually finalised.

By introducing restrictions on the revocation number, we then introduce stronger criteria. The strongest criterion, called $\mathcal{F}^c$, is obtained by restricting the revocation number to be a constant c in all the runs of the system. Informally, $\mathcal{F}^c$ guarantees that finality of each block is deferred by at most c blocks in all system runs, i.e., any block followed by at least c blocks in the blockchain cannot be revoked.

Between $\mathcal{F}$ and $\mathcal{F}^c$ we then define three other forms of deferred finality: $\mathcal{F}^n$, where the revocation number is bounded but not known, $\mathcal{F}^{\diamond,c}$, where the revocation number is constant but holds only eventually, and finally $\mathcal{F}^{\diamond,n}$, where the bound on the revocation number is not known and holds only eventually. $\mathcal{F}^n$ guarantees that finality of each block is deferred by a constant c in each system run, but this constant can vary from one run to another. For $\mathcal{F}^{\diamond,c}$ and $\mathcal{F}^{\diamond,n}$ we have that $\mathcal{F}^{\diamond,c}$ guarantees that eventually finality of each block is deferred by c in all system runs, while for $\mathcal{F}^{\diamond,n}$, eventually finality of each block is deferred by c in each system run with c varying from one run to another. Nicely, we obtain each consistency criterion by adding a proper bounded revocation property to $\mathcal{F}$ and we prove that $\mathcal{F}^n$, $\mathcal{F}^{\diamond,c}$, $\mathcal{F}^{\diamond,n}$ are all equivalent.

The rigorous formalisation of these consistency criteria enables us to easily show that solutions that guarantee $\mathcal{F}^c$ are equivalent to Consensus, while solutions that guarantee $\mathcal{F}^n$ (or equivalently $\mathcal{F}^{\diamond,n}$ and $\mathcal{F}^{\diamond,c}$) 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:

- $\mathcal{F}$ cannot be achieved in an asynchronous system if the reconciliation rule follows the "longest" chain rule (Theorem 22). 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 $\mathcal{F}$ in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks. This novel solution is simple and tolerant to an unbounded number of Byzantine processes (Theorem 23).
- A solution that solves $\mathcal{F}^n$ in an eventually synchronous environment in presence of less than half of Byzantine processes (Theorem 24). 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 $\mathcal{F}$ is not solvable in an asynchronous environment when the "longest" chain rule is used, and then presents two original and simple algorithms that respectively solve $\mathcal{F}$ and $\mathcal{F}^n$. Finally, Section 6 concludes the paper.

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. 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 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

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

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, 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* $\mathrm{BT}-\mathrm{ADT}{=}\{A = \{append(b), read()/bc \in \mathcal{BC}\}, B = \mathcal{BC} \cup \{\top, \bot\}, Z = \mathcal{BT}, \xi_0 = b_0, \tau, \delta\}$*, 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*

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

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 different consistency criteria for the blocktree. We first define eventual finality, which is the weakest consistency criterion that we may expect from blockchains, along with the notion of block revocation. We then combine eventual finality with different forms of revocation to provide stronger consistency criteria. The presented family of consistency criteria is a comprehensive characterization of what we may expect from blockchains.

▶ **Notation 3.**

- $E(a^*, r^*)$ *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* length$(bc^\frown\{b\}) >$ length(bc)*;*

- *We represent chain bc as an infinite list $b_0b^*\bot^+$ 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 4** (BT Eventual Finality Consistency criterion ($\mathcal{F}$))**.** *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:**
  $\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, \forall i \in \mathbb{N} : bc[i] \neq \bot, \exists r'_{rsp} : r_{rsp} \nearrow r'_{rsp}, \forall r''_{rsp} : r'_{rsp} \nearrow r''_{rsp}, (r'_{rsp}/bc')[i] = (r''_{rsp}/bc'')[i] \neq \bot$.
  In all the histories in which the number of read *invocations is infinite, then for any read operation such that the returned chain has a block 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. $bc'[i] = bc''[i] \neq \bot$.*
- **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 k, there will always exist a* read *operation that will return a chain of at least length k. 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.*

Bounded revocation

As previously said, the bounded revocation properties are at the heart of our formalisation of blockchain finality. Informally, given a history, we call the revocation number the natural number n such that for any two reads r/bc and r'/bc', where r precedes r', by pruning the last n blocks from bc we obtain a chain that is a prefix of bc'.

Note that the eventual finality consistency criterion presented so far does not impose any bound on the revocation number, which can be then infinite when the history goes to infinity.

To obtain stronger consistency criteria, we then introduce restrictions to the revocation number. To this aim, we define the *c-bounded revocation* property, which states that the revocation number n is bounded by a constant c in all histories. We also define the *bounded revocation property*, which states that the revocation number n is bounded by a constant

c in each history, but may be unbounded when we consider the union of all the histories, i.e., the bound can vary from a history to another. Eventual forms of c-bounded revocation and bounded revocation state that the revocation number will be equal to a constant c only eventually. More formally:

▶ **Definition 5** (c-Bounded Revocation). $\exists c \in \mathbb{N}, \forall E, \forall r_{rsp}/bc, r'_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq (\mathsf{length}(bc) - c), bc[i] = bc'[i] \neq \perp$.

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

▶ **Definition 7** (Eventual c-Bounded Revocation). $\exists c \in \mathbb{N}, \forall E, \exists r \in E : \forall r'_{rsp}/bc, r''_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, r'_{rsp} \nearrow r''_{rsp}, \forall i \in \mathbb{N} : i \leq (\mathsf{length}(bc') - c), bc'[i] = bc''[i] \neq \perp$

▶ **Definition 8** (Eventual Bounded Revocation). $\forall E, \exists c \in \mathbb{N}, \exists r \in E : \forall r'_{rsp}/bc, r''_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, r'_{rsp} \nearrow r''_{rsp}, \forall i \in \mathbb{N} : i \leq (\mathsf{length}(bc') - c), bc'[i] = bc''[i] \neq \perp$

Note that Bounded Revocation properties are 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'*.

Based on these different forms of bounded revocation, we define four criteria stronger than eventual finality. Nicely, we obtain each consistency criterion by adding the proper bounded revocation property to $\mathcal{F}$.

By adding c-bounded revocation to $\mathcal{F}$, we obtain the *c-deferred finality* form, denoted by $\mathcal{F}^c$. Informally, $\mathcal{F}^c$ guarantees that finality of each block is deferred by at most c blocks in all histories, i.e., any block followed by at least c blocks in the blockchain cannot be revoked.

By adding the bounded revocation property to $\mathcal{F}$, we obtain the *bounded deferred finality* form, denoted by $\mathcal{F}^n$. Informally $\mathcal{F}^n$ guarantees that finality of each block is deferred by a constant c in each history, but this constant can vary from history to history. In other words constant c is unknown.

Finally, by adding respectively, eventual c-bounded finality and eventual bounded finality to $\mathcal{F}$, we obtain other two forms of deferred finality, namely $\mathcal{F}^{\diamond,c}$ $\mathcal{F}^{\diamond,n}$, both equivalent to $\mathcal{F}^n$. Informally, $\mathcal{F}^{\diamond,c}$ guarantees that eventually finality of each block is deferred by c in all histories. For $\mathcal{F}^{\diamond,n}$, eventually finality of each block is deferred by c in each history, with c varying from history to history.

In the following we formally introduce $\mathcal{F}^c$, $\mathcal{F}^n$, $\mathcal{F}^{\diamond,c}$, $\mathcal{F}^{\diamond,n}$, and show equivalences between $\mathcal{F}^{\diamond,c}$, $\mathcal{F}^{\diamond,n}$ and $\mathcal{F}^n$.

▶ **Definition 9** (BT c-Deferred Finality Consistency criterion ($\mathcal{F}^c$)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT c-deferred finality consistency criterion if chain validity, chain integrity, eventual prefix, ever growing tree, and the c-bounded revocation properties hold.*

▶ **Definition 10** (BT Bounded Deferred Finality Consistency criterion ($\mathcal{F}^n$)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT bounded deferred finality consistency criterion if chain validity, chain integrity, eventual prefix, ever growing tree, and the bounded revocation properties hold.*

▶ **Definition 11** (BT Eventual c-Deferred Finality Consistency criterion ($\mathcal{F}^{\diamond,c}$)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT eventual c-deferred finality consistency criterion if chain validity, chain integrity, ever growing tree, eventual prefix and the eventual c-bounded revocation properties hold.*

▶ **Definition 12** (BT Eventual Bounded Deferred Finality Consistency criterion ($\mathcal{F}^{\diamond,n}$)). *A concurrent history $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of the system that uses a BT-ADT verifies the BT eventual bounded deferred finality consistency criterion if chain validity, chain integrity, ever growing tree, eventual prefix and the eventual bounded revocation properties hold.*

Note that in the blockchain literature, $\mathcal{F}^c$, with $c = 0$, is also referred to as *immediate finality*. Immediate finality is equivalent to BT strong consistency defined in [1], which implies that for any two read operations, one of the returned blockchains is the prefix of the other one.

▶ **Notation 13.** For readability reasons, in the following we will simply say *finality* instead of *finality consistency criterion*.

▶ **Theorem 14.** *$\mathcal{F}^n$ and $\mathcal{F}^{\diamond,n}$ are equivalent.*

Proof. Trivially, $\mathcal{F}^n$ implies $\mathcal{F}^{\diamond,n}$. Let us now consider the other direction. From $\mathcal{F}^{\diamond,n}$, we have that given any execution E, there exists $c \in \mathbb{N}$ and a read operation r such that for all reads r', r'' after r, with $r' \nearrow r''$ the blockchain returned by r' pruned of the last c blocks is a prefix of the blockchain returned by r''. Let c' be the maximal length of blockchains returned by read operations occurring before r, and let $c'' = \max(c, c')$. By construction, $\mathcal{F}^n$ is satisfied for E with revocation number $n = c''$. Hence $\mathcal{F}^{\diamond,n}$ implies $\mathcal{F}^n$. ◀

We now show that $\mathcal{F}^{\diamond,n}$ and $\mathcal{F}^{\diamond,c}$ are equivalent. This equivalence is shown by first proving that $\mathcal{F}^{\diamond,n}$ and $\mathcal{F}^{\diamond,c=0}$ are equivalent and then that $\mathcal{F}^{\diamond,c=0}$ and $\mathcal{F}^{\diamond,c}$ are equivalent.

▶ **Theorem 15.** *$\mathcal{F}^{\diamond,c=0}$ and $\mathcal{F}^{\diamond,n}$ are equivalent.*

Proof. Let $\mathcal{P}_1$ be a protocol guaranteeing $\mathcal{F}^{\diamond,n}$. We build protocol $\mathcal{P}_2$ as follows: to make an append() operation, processes simply use the append() operation of $\mathcal{P}_1$. For the read() operation, processes use the read() operation provided by $\mathcal{P}_1$ to obtain the blockchain and prune the second half of it before returning the first half of the blockchain. Let us show that protocol $\mathcal{P}_2$ guarantees $\mathcal{F}^{\diamond,c=0}$. For this, we need to show that the properties of $\mathcal{F}^{\diamond,c=0}$ 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 prefix:** The eventual prefix property is still satisfied by pruning half of the chain.
- **Ever growing tree:** The ever growing tree property is still satisfied by pruning half of the chain.
- ($c = 0$)**-eventual bounded revocation:** This property follows from 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.

For the other direction, we can build a solution to $\mathcal{F}^{\diamond,n}$ using a solution to $\mathcal{F}^{\diamond,c=0}$. ◀

▶ **Theorem 16.** *$\mathcal{F}^{\diamond,c=0}$ and $\mathcal{F}^{\diamond,c}$ are equivalent.*

Proof. Trivially, $\mathcal{F}^{\diamond,c=0}$ implies $\mathcal{F}^{\diamond,c}$. For the other direction, we apply a construction close to the one used in the proof of Theorem 15. Specifically, given a protocol $\mathcal{P}_1$ that guarantees $\mathcal{F}^{\diamond,c}$, we build a protocol $\mathcal{P}_2$ by using $\mathcal{P}_1$ as follows. To make an append() operation, processes simply use the append() operation of $\mathcal{P}_1$. For the read() operation, processes use the read() operation provided by $\mathcal{P}_1$ to obtain the blockchain and prune its last c blocks before returning it. Note that if there are less than c blocks, processes then return the genesis block. The properties of $\mathcal{F}^{\diamond,c=0}$ trivially follow from the properties of $\mathcal{F}^{\diamond,c}$ and the proposed transformation. ◀

▶ **Corollary 17.** *$\mathcal{F}^n$, $\mathcal{F}^{\diamond,n}$, $\mathcal{F}^{\diamond,c}$, and $\mathcal{F}^{\diamond,c=0}$ are equivalent.*

Proof. Straightforward from Theorems 14, 15 and 16. ◀

4 (Eventual) Consensus Reductions

In this section, we show that guaranteeing $\mathcal{F}^c$ is equivalent to solving Consensus, while guaranteeing bounded deferred finality (or any of the equivalent forms) is not weaker than solving Eventual Consensus.

4.1 c-Bounded Deferred Finality and Consensus

▶ **Theorem 18.** *Guaranteeing $\mathcal{F}^c$ is equivalent to solving Consensus.*

Proof. Let us first remark that $\mathcal{F}^{c=0}$ is equivalent to BT Strong Consistency [1], which has been shown to be equivalent to Consensus [1].

To prove the theorem it is then sufficient to give a protocol $\mathcal{P}_2$ that guarantees $\mathcal{F}^{c=0}$ given a solution $\mathcal{P}_1$ that satisfies $\mathcal{F}^c$, the other direction being trivial. We build $\mathcal{P}_2$ by applying the same transformation of $\mathcal{P}_1$ described in the proof of Theorem 16. The properties of $\mathcal{F}^{c=0}$ trivially follow from the properties of $\mathcal{F}^c$ and the proposed transformation. ◀

▶ **Corollary 19.** *There does not exist any solution that solves $\mathcal{F}^c$ in an eventual synchronous system with more than $1/3$ of Byzantine processes.*

Proof. The proof follows from the equivalence between $\mathcal{F}^c$ and Consensus (cf. Theorem 18), which is unsolvable in a synchronous (and thus also in an eventually synchronous) system with more than one third of Byzantine processes [17]. ◀

4.2 Bounded Deferred Finality and Eventual Consensus

In this section we show that guaranteeing bounded deferred finality is not weaker than Eventual Consensus. To this aim we first recall the Eventual Consensus problem with a small modification of the validity property to make it suitable to the blockchain context and then we show that $\mathcal{F}^{\diamond,c=0}$ (which is equivalent to $\mathcal{F}^{\diamond,n}$ by Corollary 17) is not weaker than Eventual Consensus.

The Eventual Consensus (EC) abstraction [9] captures eventual agreement among all participants. It exports, to every process p_i, operations $\mathsf{proposeEC}_1, \mathsf{proposeEC}_2, \ldots$ 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 $\mathsf{proposeEC}_\mathsf{j}$ as soon as it returns a response to $\mathsf{proposeEC}_{\mathsf{j}-1}$, 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 $\mathsf{proposeEC}_\mathsf{j}$ for all $j \in \mathbb{N}$.
- **EC-Integrity.** No process responds twice to $\mathsf{proposeEC}_\mathsf{j}$ for all $j \in \mathbb{N}$.
- **EC-Validity.** Every value returned to $\mathsf{proposeEC}_\mathsf{j}$ is valid with respect to predicate P_{EC}.
- **EC-Agreement.** No two correct processes return different values to $\mathsf{proposeEC}_\mathsf{j}$ for all $j \geq k$.

▶ **Theorem 20.** *Guaranteeing $\mathcal{F}^{\diamond,c=0}$ (or any of the equivalent forms) is not weaker than solving Eventual Consensus.*

Proof. We show that there exists a protocol $\mathcal{P}_1$ that solves Eventual Consensus assuming the existence of a protocol $\mathcal{P}_2$ that solves $\mathcal{F}^{\diamond,c=0}$. We do the transformation as follows. Every correct process p invokes $\mathsf{proposeEC_j}$ for all $j \in \mathbb{N}$. We impose that the validity predicate P of the blocktree ADT (see Section 3) be equal to predicate P_{EC}. When a correct process p invokes the $\mathsf{proposeEC_j}(v)$ operation of $\mathcal{P}_1$, for any $j \in \mathbb{N}$, then p executes 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 $\mathsf{read}()$ operations up to the moment the $\mathsf{read()}$ returns a chain bc such that $bc[j] \neq \bot$, 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 prefix property and the 0-eventual revocation property, which guarantees that there exists a $\mathsf{read}()$ 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.

Finally, by Corollary 17, the proof of the Theorem completes. ◀

▶ **Theorem 21.** *There does not exist any solution that solves $\mathcal{F}^n$ (and any of the equivalent forms) in an asynchronous system with at least one Byzantine process.*

Proof. The proof follows from Corollary 17 and the fact that $\mathcal{F}^{\diamond,c=0}$ is not weaker than Eventual Consensus (cf. Theorem 20). Since Eventual Consensus is equivalent to the leader election problem [9], which cannot be solved in an asynchronous system with at least one Byzantine process [23], this completes the proof of the Theorem. ◀

5 Finality Solutions

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

5.1 Impossibility to Satisfy $\mathcal{F}$ with the Longest Chain Rule

In the following we prove that, in an asynchronous environment, we cannot provide $\mathcal{F}$ 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. Note that this result holds even in absence of failures. 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 (due 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 $\mathcal{F}$ 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.

▶ **Theorem 22.** *It is impossible to guarantee $\mathcal{F}$ 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 Satisfying $\mathcal{F}$ 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() operation triggered by a correct process p returns the chain selected by f_r on the current blocktree bt of p. Given a blocktree bt, the append selection function f_a selects a chain in bt

Algorithm 1 Guaranteeing $\mathcal{F}$ with an unbounded number of Byzantine processes.

```
upon rb-delivery($bc$)
    bt.addIfValid($bc$)
end
upon append($b$)
    rb-broadcast($f_a(bt)^\frown b$)
end
upon read()
    return $f_r(bt)$
end
```

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, an infinite number of blocks is added continuously to a fork, there is not, for a given block, an infinite number of blocks with a smaller identifier. Thus eventually the selection function f_a will always select the same prefix. Finally, since blocks are diffused by an eventually reliable 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 23.** *Algorithm 1 is a solution satisfying $\mathcal{F}$ 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 $\mathcal{F}$ 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 eventually reliable broadcast, eventually for each fork all correct processes have the same block with the smallest identifier. Hence, by the read selection function f_r that at each fork selects the block with the smallest identifier in order to select the chain to return, 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:** 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 Satisfying Bounded Deferred Finality with less than half of Byzantine Processes

In this section we prove that the bounded deferred finality 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, we first recall the description of the original Streamlet [6].

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. Algorithm 2 presents Steamlet protocol [6].

The Accountable Forking ($\mathcal{AF}$) Algorithm. We propose $\mathcal{AF}$, an algorithm that extends Streamlet. $\mathcal{AF}$ guarantees that for any given fork, correct processes can blame the process that originates it, i.e, a Byzantine process creating a fork is accountable for it. This is achieved as follows: 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 we propose goes deeper: if a fork occurs, any correct processes can detect the Byzantine process that originated it, and excludes it from the voters. Specifically, when two conflicting chains are finalized (i.e., two finalized chains that are not the prefix of one another) then processes look for inconsistent blocks. By definition, two notarized blocks b, b' are inconsistent with one another if one of the following two conditions holds:

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

Algorithm 2 Streamlet algorithm [6].

- **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.

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

▶ **Theorem 24.** *There exists a solution that satisfies $\mathcal{F}^{\diamond,c=0}$ (and all the equivalent forms) in an eventually synchronous system with less than half Byzantine processes.*

Proof. We show in the Appendix that algorithm $\mathcal{AF}$ is such a solution. ◀

6 Conclusion

In this work we have defined different consistency criteria for blockchains. We have first defined eventual finality, which is the weakest consistency criterion that we may expect from blockchains, along with the notion of block revocation. By combining eventual finality with different forms of revocation we obtained stronger consistency criteria, thus providing a comprehensive characterization of what we may expect from blockchains. We have formally shown that in an asynchronous system it is not possible to provide a known bound on the number of blocks that can be revoked. On the other hand, we have proposed for the first time a solution in an eventually synchronous system with less than half of Byzantine processes guaranteeing that eventually such bound is reached. We have also shown that in an asynchronous system, finality with no bound on the number of revocable blocks cannot be solved using the reconciliation rule of Bitcoin. Still we provide an asynchronous solution with an unlimited number of Byzantine processes. We hope that this work will better guide blockchain designs.

References

1 Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Blockchain abstract data type. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

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

3 Antonio Anta Fernández, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.

4 Lăcrămioara Aştefanoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu. Tenderbake – A solution to dynamic repeated consensus for blockchains. In *Proceedings of the Fourth International Symposium of Foundations and Applications of Blockchain*, 2021.

5 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, 2017. `arXiv:1710.09437`.

6 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. `https://eprint.iacr.org/2020/088.pdf`, 2020.

7 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 2019.

8 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. *CoRR*, abs/1702.03068, 2017. `arXiv:1702.03068`.

9 Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest failure detector for eventual consistency. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

10 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. EUROCRYPT International Conference*, 2015. Updated version 2020: https://eprint.iacr.org/2014/765.pdf.

11 L.M. Goodman. Tezos – A self-amending crypto-ledger, 2014.

12 Ian Grigg. EOS: An introduction. `https://whitepaperdatabase.com/eos-whitepaper/`.

13 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

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

15 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Proceedings of the Advances in Cryptology*, 2017.

16 Artem Koltsov, Vitaly Cheremensky, and Stanislav Kapulkin. Casper White Paper.

17 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

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

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

20 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *www.bitcoin.org*, 2008.

21 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proceedings of the EUROCRYPT International Conference*, 2017.

22 Matthieu Perrin. *Distributed Systems, Concurrency and Consistency.* ISTE Press, Elsevier, 2017.

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

24 Alistair Stewart. Poster: Grandpa finality gadget. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2649–2651, 2019.

25 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/Paper.pdf`.

A Appendix

▶ **Theorem 22.** *It is impossible to guarantee $\mathcal{F}$ if the* append *operation is based on the longest chain rule in an asynchronous environment.*

Proof. 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 proof-of-work (PoW) approaches), we cannot reach eventual finality if we rely on the longest chain rule to resolve forks.

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 $\perp$ 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 1. 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 larger 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.

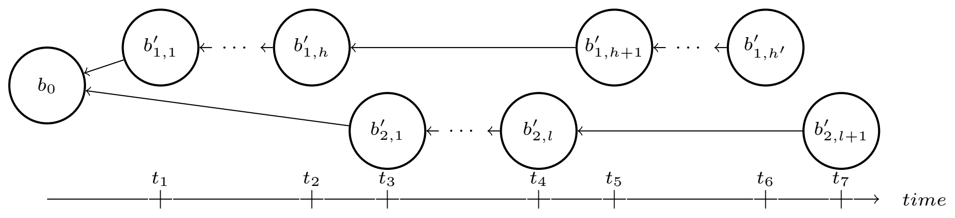

Figure 1 A blocktree generated by two processes. On the x-axis the longest chain value of each chain at different time instants (from the root to the current leaf) and the relationships between those values.

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 criterion 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 c in the proof could be larger than 1 and 2 respectively. ◀

▶ **Theorem 24.** *There exists a solution that satisfies $\mathcal{F}^{\diamond,c=0}$ (and all the equivalent forms) in an eventually synchronous system with less than half Byzantine processes.*

Proof. Let us first demonstrate that voting for two inconsistent blocks b and b' is a Byzantine failure. We have two cases to consider. If both b and b' are inconsistent because Condition 1 holds, then the intersecting voters are Byzantine as correct processes vote only once per epoch. Hence if process q votes for b and b' then q is Byzantine. If both b and b' are inconsistent because Condition 2 is met, then the intersecting voters are Byzantine, as correct processes vote only for blocks extending one of the longest notarized chains. That is, if 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 afterwards for a block b' whose height is strictly smaller than $b.height$ because p must extend one of the longest notarized chain. It follows that if process q votes for both b and b' then q is Byzantine.

Let us now show that a fork occurs because of two inconsistent blocks. If there is a fork then this gives rise to 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' belonging to the prefix of b'_3 such that $b'.epoch > b_1.epoch$ and $b'.height$ is the smallest in the prefix of b'_3. Such block always exists as b'_1 satisfies those two conditions. We have two cases: Either $b'.height < b_3.height$ or $b'.height \geq b_3.height$. In the former case, b' is inconsistent with b_3 since by assumption $b'.epoch > b_3.epoch$. In the latter case, the predecessor of b' is inconsistent with b_3. Indeed, the predecessor of b' has a strictly smaller height than b_1 and by assumption has a larger epoch number than b_3. Figure 2 illustrates the presence

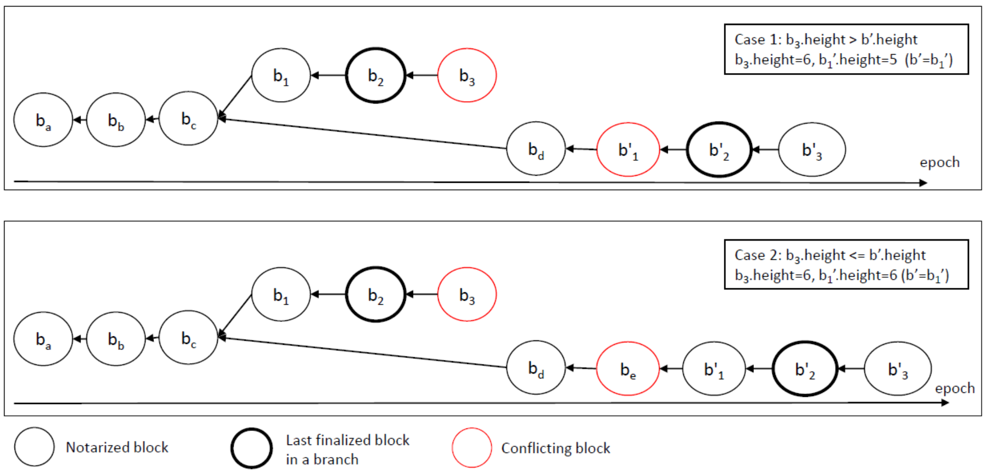
Case 1: b3.height > b'.height
b3.height=6, b1'.height=5 (b'=b1')
ba
bb
bc
b1
b2
b3
bd
b'1
b'2
b'3
epoch
Case 2: b3.height <= b'.height
b3.height=6, b1'.height=6 (b'=b1')
ba
bb
bc
b1
b2
b3
bd
be
b'1
b'2
b'3
epoch
Notarized block
Last finalized block in a branch
Conflicting block

Figure 2 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.

of inconsistent blocks in presence of a fork at some block b_c. From b_c two chains are built, the first one consisting of the sequence of three blocks b_1, b_2 and b_3, and the second chain consisting of four consecutive blocks b_d, b'_1, b'_2, b'_3 (to illustrate the first case) and of five consecutive blocks $b_d, b_e, b'_1, b'_2, b'_3$ (to illustrate the second case). In both cases block b'_1 plays the role of block b'. In the first case (figure in the top), $b_3.height = 6$ and $b'.height = 5$ while $b_3.epoch = 6$ and $b'.height = 5$. Thus Condition 2 applies. In the second case (figure in the bottom), since $b'.height \geq b3.height$ then there must exist some block b_e in the b' prefix. Thus $b_e.height < b'.height$. Given that by assumption $b_e.epoch > b_3.epoch$, then Condition 2 holds for b_e and b_3. Hence there is always a couple of inconsistent blocks in a fork.

Let us now conclude our proof that protocol $\mathcal{AF}$ solves $\mathcal{F}^{\diamond,c=0}$. If a fork occurs, then each correct process eventually detects at least one Byzantine process and ignores its votes. Thus, the number of forks is finite since we have a finite number of Byzantine processes. As a consequence, there is always a single chain that is eventually finalized. As there is a majority of correct processes, algorithm $\mathcal{AF}$ remains live as the original Streamlet one. Algorithm $\mathcal{AF}$ also inherits the properties of the original Streamlet algorithm regarding the eventual finalization of blocks when the system becomes synchronous.

Finally, by applying Corollary 17, we complete the proof of the theorem. ◀