



HAL
open science

On finality in blockchains

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

► **To cite this version:**

Emmanuelle Anceaume, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni. On finality in blockchains. 2021. cea-03080029v3

HAL Id: cea-03080029

<https://cea.hal.science/cea-03080029v3>

Preprint submitted on 17 May 2021 (v3), last revised 22 Nov 2021 (v5)

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

1 On Finality in Blockchains

2 **Emmanuelle Anceaume** @

3 CNRS, Univ Rennes, Inria, IRISA, France

4 **Antonella Del Pozzo** @

5 Paris-Saclay, CEA, List, France

6 **Thibault Rieutord** @

7 Paris-Saclay, CEA, List, France

8 **Sara Tucci-Piergiovanni** @

9 Paris-Saclay, CEA, List, France

10 — Abstract —

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

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

30 2012 ACM Subject Classification

31 **Keywords and phrases** Blockchain specification, Finality

32 **Digital Object Identifier** 10.4230/LIPIcs...

33 1 Introduction

34 This paper focuses on blockchain finality, which refers to the time when it becomes impossible
35 to remove a block that has previously been appended to the blockchain. Blockchain finality
36 can be deterministic or probabilistic, immediate or eventual.

37 Informally, immediate finality guarantees, as its name suggests, that when a block is
38 appended to a local copy, it is immediately finalized and thus will never be revoked in the
39 future. Designing blockchains with immediate finality favors consistency against availability
40 in presence of transient partitions of the system. It leverages the properties of Consensus (i.e
41 a decision value is unique and agreed by everyone), at the cost of synchronization constraints.
42 Assuming partially synchronous environments, most of the permissioned blockchains satisfy
43 the deterministic form of immediate consistency, as for example Red Belly blockchain [9]
44 and Hyperledger Fabric blockchain [2]. The probabilistic form of immediate consistency is
45 typically achieved by permissionless pure proof-of-stake blockchains such as Algorand [8].



© Emmanuelle Anceaume and Antonella Del Pozzo and Thibault Rieutord and Sara Tucci-Piergiovanni;

licensed under Creative Commons License CC-BY 4.0



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

XX:2 On Finality in Blockchains

46 Unlike immediate finality, eventual finality only ensures that all local copies of the
47 blockchain share a common increasing prefix, and thus finality of their blocks increases as
48 more blocks are appended to the blockchain. The majority of cryptoassets blockchains, with
49 Bitcoin [21] and Ethereum [27] as celebrated examples, guarantee eventual finality with
50 some probability: blocks may be revoked after being appended to the blockchain, yet with
51 decreasing probability as they sink deeper into the chain. More recently, a huge effort has
52 been devoted to propose alternatives to the energy-wasting proof-of-work method of Bitcoin
53 and Ethereum. These proof-of-stake blockchains (e.g. [17, 22, 13, 16]) offer as well a form
54 of eventual finality. More broadly, all these solutions favor availability (or progress) under
55 adversarial conditions, therefore they do not rely on Byzantine Consensus. This implies that
56 it is admitted that a blockchain may lose consistency by incurring a fork, which is the presence
57 of multiple chains at different processes. The heart of these solutions is then a reconciliation
58 mechanism, which is always available to recover from a fork. Reconciliation typically consists
59 in a local deterministic rule selecting a chain among the different alternatives available. In
60 Bitcoin for instance any participant is able to reconcile the state following the "longest"
61 chain rule. Once a winner chain is chosen, the other alternatives are revoked, as such all
62 the blocks belonging to them. It is important to stress that the design of these blockchains
63 aims at being resistant to adversarial participants creating alternative chains on purpose in
64 synchronous environments. This means that during reconciliation all the candidate chains
65 available at one honest participant are also available at all other honest participants. This
66 allows us to compute finalisation probabilistic guarantees, such as the one in Bitcoin that
67 says that it is computationally hard to revoke a block followed by six other blocks in presence
68 of no more than 10% of Byzantine participants (selfish attack). Real large-scale distributed
69 systems, however, can hardly be synchronous. Network effects make the moment at which all
70 honest processes observe the same set of candidate chains unknown. The asynchrony effect
71 might render the reconciliation rule and finalisation guarantees unsure, or simply extremely
72 inefficient, for example by considering a block as finalised after one or more days. To solve
73 this problem a number of permissionless blockchain projects are investigating how to add
74 "finality gadgets" (e.g., [5, 26]) to proof-of-work or proof-of-stake blockchains, which means
75 seeking additional mechanisms or protocols to reach "better" finality properties in network
76 adversarial settings. The hope is to find ways to get deterministic finality by periodically
77 running finality gadgets. For the time being, the only way that has been concretely pursued is
78 to add Byzantine Consensus – e.g. Tenderbake [4] adds Byzantine Consensus to the existing
79 proof-of-stake method assuring deterministic finality to each block followed by other two
80 blocks. How to add mechanisms that do not resort to Consensus, however, is an intriguing
81 and open question, related to the finality properties one would like to guarantee.

82 The quest for "good" deterministic finality properties for blockchains is still in its infancy,
83 though. Our motivation is to provide a thorough study of several possible finality properties
84 and explore their solvability. In doing so, we are particularly intrigued by answering the
85 question, what lies between eventual finality and immediate finality?

86 To this aim we introduce the notion of bounded revocation, which informally says that
87 the number of blocks that can be revoked from the current blockchain is bounded. Providing
88 solutions that guarantee deterministic bounded revocation reveals to be an important crux
89 in the construction of blockchains. We thus provide rigorous definitions for the weakest form
90 of eventual finality \mathcal{EF}^* , which does not guarantee any bound on the number of blocks that
91 can be revoked from the blockchain at any given fork, and two stronger forms: $\mathcal{EF}^{\diamond c}$, in
92 which revocation is bounded but unknown, and \mathcal{EF}^c in which revocation is bounded and
93 known. Intuitively, if \mathcal{EF}^c holds, processes revoke at most a constant number c of blocks

94 from the current blockchain at each reconciliation, while if $\mathcal{EF}^{\diamond c}$ holds, processes revoke at
 95 most a constant number of c blocks from the current chain only eventually, i.e., after a finite
 96 but unknown number of reconciliations.

97 The rigorous formalisation of these properties enable us to easily show that solutions that
 98 guarantee \mathcal{EF}^c are equivalent to Consensus, while solutions that guarantee $\mathcal{EF}^{\diamond c}$ are not
 99 weaker than Eventual Consensus, an abstraction that captures eventual agreement among all
 100 participants. From these reductions, we show some related impossibility results in presence of
 101 Byzantine processes. Beside reductions and related impossibilities, we propose the following
 102 non-trivial results:

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

117 We hope that these results will better guide blockchain designs and link them to clear
 118 formal properties of finality.

119 1.0.0.1 Related Work

120 Formalization of blockchains in the lens of distributed computing has been recognized as an
 121 extremely important topic [15]. Garay et al. [11] have been the first to analyze the Bitcoin
 122 backbone protocol and to define invariants this protocol has to satisfy to verify with high
 123 probability an eventual consistent prefix, i.e. probabilistic eventual finality. The authors
 124 have analyzed the protocol in a synchronous system, while Pass et al. [23] have extended
 125 this line of work considering a more adversarial network. Anta et al. [3] have proposed
 126 a formalization of distributed ledgers modeled as an ordered list of records along with
 127 implementations for sequential consistency and linearizability using a total order broadcast
 128 abstraction. Not related to the blockchain data structure, authors of [14] have formalized
 129 the notion of cryptocurrency showing that Consensus is not needed.

130 While probabilistic eventual finality has been widely studied in the context of Bit-
 131 coin [11, 6, 23], only a few studies have started to lay the foundations of the computation
 132 power of blockchains with deterministic eventual finality consistency. Anceaume et al. [1]
 133 have been the first to capture the convergence process of two distinct classes of blockchain
 134 systems: the class providing strong prefix (for each pair of chains returned at two different
 135 processes, one is the prefix of the other) and the class providing eventual prefix, in which
 136 multiple chains can co-exist but the common prefix eventually converges. Interestingly, the
 137 authors of [1] show that to solve strong prefix the Consensus abstraction is needed, however
 138 they do not address solvability of eventual prefix, which is the focus of this paper.

XX:4 On Finality in Blockchains

140 The paper is organised as follows: Section 2 formally presents the sequential specification
141 of a blockchain and the formalisation of the different finality properties we may expect from
142 a blockchain when concurrently accessed. Section 3 presents reductions between different
143 forms of finality, Consensus and Eventual Consensus. Section 4 first shows why \mathcal{EF}^* is not
144 solvable in an asynchronous environment when the "longest" chain rule is used, and then
145 presents the algorithms to solve \mathcal{EF}^* and $\mathcal{EF}^{\diamond c}$. These algorithms are particularly simple.
146 Finally, Section 5 concludes.

147 2 Definitions

148 2.1 Preliminary Definitions

149 We describe a blockchain object as an abstract data type which allows us to completely
150 characterize a blockchain by the operations it exports [19]. The basic idea underlying the
151 use of abstract data types is to specify shared objects using two complementary facets: a
152 sequential specification that describes the semantics of the object, and a consistency criterion
153 over concurrent histories, i.e. the set of admissible executions in a concurrent environment [24].
154 Prior to presenting the blockchain abstract data type we first recall the formalization used
155 to describe an abstract data type (ADT).

156 2.1.0.1 Abstract data types.

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

166 2.1.0.2 Concurrent histories of an ADT

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

169 ► **Definition 1. (Concurrent history H)** *The execution of a program that uses an abstract*
170 *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*

- 171 ■ $\Sigma = A \cup (A \times B)$ is a countable set of operations;
- 172 ■ E is a countable set of events that contains all the ADT operations invocations and all
173 ADT operation response events;
- 174 ■ $\Lambda : E \rightarrow \Sigma$ is a function which associates events to the operations in Σ ;
- 175 ■ \mapsto : is the process order, irreflexive order over the events of E . Two events $(e, e') \in E^2$
176 are ordered by \mapsto if they are produced by the same process, $e \neq e'$ and e happens before e' ,
177 that is denoted as $e \mapsto e'$.
- 178 ■ \prec : is the operation order, irreflexive order over the events of E . For each couple
179 $(e, e') \in E^2$ if e' is the invocation of an operation occurred at time t' and e is the response
180 of another operation occurred at time t with $t < t'$ then $e \prec e'$;

181 ■ \nearrow : is the program order, irreflexive order over E , for each couple $(e, e') \in E^2$ with $e \neq e'$
 182 if $e \mapsto e'$ or $e \prec e'$ then $e \nearrow e'$.

183 2.2 The blocktree ADT

184 We represent a blockchain as a tree of blocks. Indeed, while consensus-based blockchains
 185 prevent forks or branching in the tree of blocks, blockchain systems based on proof-of-work
 186 allow the occurrence of forks to happen hence presenting blocks under a tree structure. The
 187 blockchain object is thus defined as a blocktree abstract data type (Blocktree ADT).

188 2.2.1 Sequential Specification of the Blocktree ADT (BT-ADT)

189 A blocktree data structure is a directed rooted tree $bt = (V_{bt}, E_{bt})$ where V_{bt} represents a set
 190 of blocks and E_{bt} a set of edges such that each block has a single path towards the root of
 191 the tree b_0 called the genesis block. Let \mathcal{BT} be the set of blocktrees, \mathcal{B} be the countable and
 192 non empty set of uniquely identified blocks and let \mathcal{BC} be the countable non empty set of
 193 blockchains, where a blockchain is a path from a leaf of bt to b_0 . A blockchain is denoted by
 194 bc . The structure is equipped with two operations `append()` and `read()`. Operation `append(b)`
 195 adds the block $b \notin bt$ to V_{bt} and adds the edge (b, b') to E_{bt} where $b' \in V_{bt}$ is returned by the
 196 append selection function $f_a()$ applied to bt . Operation `read()` returns the chain bc selected
 197 by the read selection function $f_r()$ applied to bt (note that in [1], the `read()` and `append()`
 198 operations are defined with a unique selection function). The read selection $f_r()$ takes as
 199 argument the blocktree and returns a chain of blocks, that is a sequence of blocks starting
 200 from the genesis block to a leaf block of the blocktree. The chain b_c returned by a `read()`
 201 operation r is called the blockchain, and is denoted by r/bc . The append selection function
 202 $f_a()$ takes as argument the blocktree and returns a chain of blocks. Function `last_block()`
 203 takes as argument a chain of blocks and returns the last appended block of the chain. Only
 204 blocks satisfying some validity predicate P can be appended to the tree. Predicate P is
 205 an application-dependent predicate used to verify the validity of the chain obtained by
 206 appending the new block b to the chain returned by $f_a()$ (denoted by $f_a(bt) \frown b$). In Bitcoin
 207 for instance this predicate embeds the logic to verify that the obtained chain does not contain
 208 double spending or overspending transactions. Formally,

209 ► **Definition 2.** (*Sequential specification of the Blocktree ADT*) The Blocktree Abstract Data
 210 Type is the 6-tuple $\text{BT-ADT} = \{A = \{\text{append}(b), \text{read}()/bc \in \mathcal{BC}\}, B = \mathcal{BC} \cup \{\top, \perp\}, Z =$
 211 $\mathcal{BT}, \xi_0 = b_0, \tau, \delta\}$, where the transition function $\tau : Z \times A \rightarrow Z$ is defined by

$$212 \quad \tau(bt, \text{read}()) = bt$$

$$213 \quad \tau(bt, \text{append}(b)) = \begin{cases} (V_{bt} \cup \{b\}, E_{bt} \cup \{b, \text{last_block}(f_a(bt))\}) & \text{if } P(f_a(bt) \frown b) \\ bt & \text{otherwise,} \end{cases}$$

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

$$216 \quad \delta(bt, \text{read}()) = f_r(bt)$$

$$217 \quad \delta(bt, \text{append}(b)) = \begin{cases} \top & \text{if } P(f_a(bt) \frown b) \\ \perp & \text{otherwise.} \end{cases}$$

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

2.2.2 Concurrent Specification and Consistency Criteria of the BlockTree ADT

The concurrent specification of the blocktree abstract data type is the set of concurrent histories. A blocktree consistency criterion is a function that returns the set of concurrent histories admissible for the blocktree abstract data type.

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 provide 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;
- $\text{length} : \mathcal{BC} \rightarrow \mathbb{N}$ 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 $\text{length}(bc \frown \{b\}) > \text{length}(bc)$;
- $bc \sqsubseteq bc'$ iff bc prefixes bc' .
- $bc[i]$ refers to the i -th block of blockchain bc .

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

■ Chain validity:

- $\forall r_{rsp} \in E, P(r_{rsp}/bc)$.
- Each returned chain is valid.

■ Chain integrity:

- $\forall r_{rsp} \in E, \forall b \in r_{rsp}/bc : b \neq b_0, \exists a_{inv}(b) \in E, a_{inv}(b) \nearrow r_{rsp}$.
- If a block different from the genesis block is returned, then an `append` operation has been invoked with this block as parameter. This property is to avoid the situation in which reads return blocks never appended.

■ Eventual prefix:

- $\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, \forall i \in \mathbb{N} : bc[i] \neq \perp, \exists r'_{rsp}, \forall r''_{rsp} : r'_{rsp} \nearrow r''_{rsp}, ((r'_{rsp}/bc)[i] = (r''_{rsp}/bc)[i])$.
- In all the histories in which the number of `read` invocations is infinite, then for any non empty read chain position i , there exists a `read` r'/bc' from which all the subsequent reads r''/bc'' will return the same block at position i , i.e. $bc'[i] = bc''[i]$.

■ Ever growing tree:

- $\forall E \in E(a^*, r^*), \forall k \in \mathbb{N}, \exists r \in E : \text{length}(r_{rsp}/bc) > k$.
- In all the histories in which the number of `append` and `read` invocations is infinite, for each length k , there exists a `read` that returns a chain with length greater than k . This property avoids the trivial scenario in which the length of the chain remains unchanged despite the occurrence of an infinite number of `append` operations. This can happen for instance if the tree is built as a star with infinite branches of bounded length.

269 ► **Definition 4** (BT Immediate Finality Consistency criterion (IF)). *A concurrent history*
 270 $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ *of the system that uses a BT-ADT verifies the BT immediate finality*
 271 *consistency criterion if chain validity, chain integrity, ever growing tree (as defined for EF)*
 272 *and the following property hold:*

273 ■ **Strong prefix:**

274 $\forall r_{rsp}, r'_{rsp} \in E^2, (r'_{rsp}/bc' \sqsubseteq r_{rsp}/bc) \vee (r_{rsp}/bc \sqsubseteq r'_{rsp}/bc').$

275 *For each pair of returned blockchains, one blockchain is the prefix of the other.*

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

280 ■ **Eventual strong prefix:**

281 $\forall E \in E(a, r^*) \cup E(a^*, r^*), \exists r_{rsp} \in E, \forall r'_{rsp}, r''_{rsp} \in E^2 : r_{rsp} \nearrow r'_{rsp} \wedge r_{rsp} \nearrow r''_{rsp}, (r''_{rsp}/bc' \sqsubseteq$
 282 $r'_{rsp}/bc) \vee (r'_{rsp}/bc \sqsubseteq r''_{rsp}/bc').$

283 *In all histories with an infinite number of reads, there exists a read r from which for each*
 284 *pair of returned blockchains, one blockchain is the prefix of the other.*

285 Bounded revocation

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

289 ► **Definition 6.** *c-Bounded revocation*

290 ■ $\exists c \in \mathbb{N}, \forall r_{rsp}, r'_{rsp} \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq \text{length}(r_{rsp}/bc) - c, (r_{rsp}/bc)[i] =$
 291 $(r'_{rsp}/bc')[i].$

292 ► **Notation 2.** For readability reasons, in the following we will simply say *finality* instead of
 293 *finality consistency criterion*, i.e., eventual finality consistency criterion will be replaced by
 294 eventual finality, and (eventual) immediate finality consistency criterion will be replaced by
 295 (eventual) immediate finality.

296 We can now define the c -Bounded Eventual Finality criteria by augmenting the previous
 297 consistency criteria with the Bounded revocation property:

298 ► **Definition 7.** *c-Bounded Eventual Finality criteria*

299 ■ $\mathcal{EF}^* = EF$, *in this case the revocation is unbounded.*

300 ■ $\mathcal{EF}^c = EIF$ *combined with c-Bounded revocation, such that c is known a priori.*

301 ■ $\mathcal{EF}^{\diamond c} = EIF$ *combined with c-Bounded revocation where c is unknown but bounded.*

302 We will show in the following that satisfying \mathcal{EF}^c is equivalent to immediate finality (IF).
 303 This is because from any algorithm \mathcal{P} implementing \mathcal{EF}^c , if we take the blockchain that is
 304 returned by a read provided by \mathcal{P} except for the last c blocks, this guarantees the strong
 305 prefix property of IF. Furthermore, $\mathcal{EF}^{\diamond c}$ boils down to eventual immediate finality (EIF).
 306 Indeed as shown later, if we take half of the blockchain returned by a read provided by an
 307 algorithm \mathcal{P} implementing $\mathcal{EF}^{\diamond c}$, this guarantees eventual immediate finality since chains
 308 are always growing, and thus the number of removed blocks increases up to reaching c .

309 In the following section we prove the above-mentioned equivalences more formally and
 310 study relationships to known problems such as consensus and eventual consensus.

311 **3 (Eventual) Consensus Reductions**

312 In this section we investigate the impact of the bounded revocation property on the construc-
 313 tion of a blocktree satisfying eventual finality. In particular, we show that when the bound c
 314 is known, this problem is equivalent to the consensus abstraction, while when unknown, this
 315 problem is not weaker than the eventual consensus abstraction [10].

316 **3.1 Known Bounded Revocation and Consensus**

317 ► **Theorem 8.** \mathcal{EF}^c is equivalent to *Consensus*.

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

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

- 328 ■ **Chain validity:** The chain validity property is still satisfied by pruning the last c blocks.
- 329 ■ **Chain integrity:** The chain integrity property is still satisfied by pruning the last c
 330 blocks.
- 331 ■ **Strong prefix:** The strong prefix property follows from the known bounded revocation
 332 property and the removal of the last c blocks. Indeed, if we remove the last c blocks, then
 333 for any two `read()` operations, then the first `read()` returns a prefix of the second `read()`
 334 operation.
- 335 ■ **Ever growing tree:** The ever growing tree property is still satisfied by pruning the last
 336 c blocks.

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

339 From Theorem 8 immediately follows the following impossibility result:

340 ► **Theorem 9.** *There does not exist any solution that solves \mathcal{EF}^c in an eventual synchronous*
 341 *system with more than $n/3$ Byzantine processes, where n is the number of processes partici-*
 342 *parting to the algorithm.*

343 **Proof.** The proof follows from the equivalence between \mathcal{EF}^c and *Consensus* (cf. Theorem 8),
 344 which is unsolvable in a synchronous (and thus also in an eventually synchronous) system
 345 with more than one third of Byzantine processes [18]. ◀

346 **3.2 Unknown Bounded Revocation and Eventual Consensus**

347 In this section we show that $\mathcal{EF}^{\diamond c}$ is not weaker than eventual consensus. We first show its
 348 equivalence with eventual immediate finality (EIF). Later we recall the eventual consensus
 349 problem with a small modification of the validity property to make it suitable to the blockchain
 350 context and show that eventual immediate finality is not weaker than eventual consensus.

351 ► **Theorem 10.** $\mathcal{EF}^{\diamond c}$ is equivalent to *eventual immediate finality*.

352 **Proof.** Let \mathcal{P}_1 be a protocol solving $\mathcal{EF}^{\diamond c}$ and let us show that we can solve eventual
 353 immediate finality. To do so, we consider the following modification to the protocol \mathcal{P}_1 . To
 354 make an `append()` operation, processes simply use the `append()` operation provided by \mathcal{P}_1 .
 355 But, for a `read()` operation, processes use the `read()` operation provided by \mathcal{P}_1 to obtain a
 356 chain and prune the second half of the returned chain before returning the remaining half of
 357 the chain.

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

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

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

381 The eventual consensus (EC) abstraction [10] captures eventual agreement among all
 382 participants. It exports, to every process p_i , operations `proposeEC`₁, `proposeEC`₂, ... that
 383 take multi-valued arguments (correct processes propose valid values) and return multi-valued
 384 responses. Assuming that, for all $j \in \mathbb{N}$, every process invokes `proposeEC` _{j} as soon as it
 385 returns a response to `proposeEC` _{$j-1$} , the abstraction guarantees that, in every admissible run,
 386 there exists $k \in \mathbb{N}$ and a predicate P_{EC} , such that the following properties are satisfied:

- 387 ■ **EC-Termination.** Every correct process eventually returns a response to `proposeEC` _{j}
 388 for all $j \in \mathbb{N}$.
- 389 ■ **EC-Integrity.** No process responds twice to `proposeEC` _{j} for all $j \in \mathbb{N}$.
- 390 ■ **EC-Validity.** Every value returned to `proposeEC` _{j} is valid with respect to predicate P_{EC} .
- 391 ■ **EC-Agreement.** No two correct processes return different values to `proposeEC` _{j} for all
 392 $j \geq k$.

393 ▶ **Theorem 11.** *Eventual immediate finality is not weaker than eventual consensus.*

394 **Proof.** We show that there exists a protocol \mathcal{P}_1 to solve eventual consensus starting from a
 395 protocol \mathcal{P}_2 that solves eventual immediate finality. We do the transformation as follows.
 396 Every correct process p invokes `proposeEC` _{j} for all $j \in \mathbb{N}$. We impose that the validity
 397 predicate P of the blocktree ADT (see Section 2) be equal to predicate P_1 . When a correct

XX:10 On Finality in Blockchains

398 process p invokes the $\text{proposeEC}_j(v)$ operation of \mathcal{P}_1 , for any $j \in \mathbb{N}$, then p executes the
399 following sequence of three steps: (i) it invokes the $\text{append}(v)$ operation of \mathcal{P}_2 , then (ii) it
400 invokes a sequence of $\text{read}()$ operations up to the moment the $\text{read}()$ returns a chain bc such
401 that $bc[j] \neq \perp$, and finally (iii) p returns chain bc as decision for $\text{proposeEC}_j(v)$ and triggers
402 the next operation $\text{proposeEC}_{j+1}(v')$.

403 Let us show that protocol \mathcal{P}_1 solves eventual consensus.

- 404 ■ **EC-Termination** This property is guaranteed by the ever growing tree property.
- 405 ■ **EC-Integrity** This property follows directly from the transformation.
- 406 ■ **EC-Validity** This property follows by construction and the chain validity property, since
407 predicate P equals to predicate P_1 .
- 408 ■ **EC-Agreement** This property follows by the eventual strong prefix property, which
409 guarantees that there exists a $\text{read}()$ operation r such that, all the subsequent ones return
410 blockchains that are each prefix of the following one. In other words, eventually there is
411 agreement on the value contained in $bc[j]$. This implies that there exists k for which all
412 proposeEC_j with $j \geq k$ return the same value to all correct processes.

413 ◀

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

416 **Proof.** The proof follows from the relationship between the $\mathcal{EF}^{\diamond c}$ and eventual immediate
417 finality (EIF). EIF is not weaker than the eventual consensus problem (cf. Theorem 11), which
418 is equivalent to the leader election problem [10] which cannot be solved in an asynchronous
419 system with at least one Byzantine process [25]. ◀

420 4 Eventual Finality Solutions

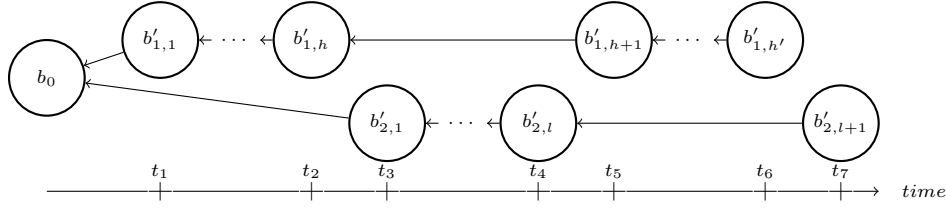
421 In this section we first show the impossibility of solving \mathcal{EF}^* when the append operation, in
422 case of forks, selects the "longest" chain. We then provide the first solution to \mathcal{EF}^* with an
423 unbounded number of Byzantine processes using an alternative selection rule.

424 4.1 Impossibility of Eventual Finality with the Longest Chain Rule

425 In the following we prove that we cannot provide \mathcal{EF}^* if, in case of forks, the append selection
426 function $f_a()$ follows the longest chain rule, i.e., returns the longest chain of the blockchain
427 tree. To show this impossibility, we consider a scenario in which the occurrence of any fork
428 produces at most two alternative chains (this is often referred to as a branching factor of 2).
429 We consider a finite number of processes and an append selection function f_a that in case of
430 forks deterministically selects the longest chain, i.e., the chain with the largest number of
431 blocks (the length is thus a monotonically increasing function on prefixes), and in case of a
432 tie selects the chain whose last block is the smallest (in the lexicographical order). We show
433 that it is impossible to guarantee \mathcal{EF}^* for such append selection function f_a .

434 Note that such a selection function is used by many blockchain systems. In proof-of-work
435 systems such as Bitcoin, chains are selected as the chain with the greater number of blocks
436 (actually this corresponds to the heaviest one by considering the difficulty) while in Ethereum
437 chains are selected using the chain with greatest weight, both captured by the selection of
438 chains according to the longest chain. In proof-of-stake systems like EOS [13] or Tezos [12]
439 the same rule is also applied.

440 Intuitively, the impossibility follows from the fact that with the longest chain selection,
441 races can occur between different branches in the tree. We show that as forks may occur, we



■ **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.

442 can create two infinite branches sharing only the root. One or the other branch constitutes
 443 alternatively the longest chain and **append** operations select chains from each branch alter-
 444 natively. This is enough to show that the only common prefix that is returned is the root
 445 hence, violating eventual finality.

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

450 To capture the synchronisation power of the system, we introduce an oracle that regulates
 451 the number of appended children from a same parent. The same approach has been proposed
 452 in [1]. The branching factor of an oracle is the maximal number of children that can
 453 be appended to a block. The oracle is the only generator of valid blocks. It owns a
 454 synchronization power equal to Consensus if its branching factor is equal to 1.

455 The oracle grants access to the blocktree as a shared object, through the following three
 456 operations: **update_view()** returns the current state of the blocktree; **getValidBlock(b_i, b_j)**
 457 returns a valid block b'_j , constructed from b_j , that can be appended to block b_i , where b_i is
 458 already included in the blocktree; and **setValidBlock(b_i, b'_j)** appends the valid block b'_j to b_i ,
 459 and returns \top when the block is successfully appended and \perp otherwise.

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

462 **Proof.** In the proof we consider the stronger oracle allowing the occurrence of one fork, i.e.,
 463 an oracle with branching factor equal to 2. That is, this oracle allows for two valid blocks to
 464 be appended to the same parent, afterwards, it shall return \perp to all requests.

465 Let p_1 and p_2 be two processes trying to **append** infinitely many blocks. W.l.o.g., we
 466 carry out this proof with a length function equal to the number of blocks.

467 At time t_0 , for both p_1 and p_2 , the **update_view()** of bt equals b_0 , thus when both apply
 468 the **append** selection function f_a on it to select the leaf where to **append** the new block, they
 469 both get b_0 . Then they both call **getValidBlock($b_0, b_{i,1}$)** = b'_i , where $i = 1$ for p_1 and $i = 2$
 470 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
 471 call **setValidBlock($b_0, b'_{1,1}$)**, which must return \top and hence $b'_{1,1}$ is appended to b_0 . Process p_1
 472 then proceeds to **append** a new block $b_{1,2}$, i.e., after having updated its bt 's view, through the
 473 **update_view()** function, p_1 applies the **append** selection function f_a on it to select the leaf
 474 where to **append** its new block, in this case the only leaf is $b'_{1,1}$. It calls **getValidBlock($b'_{1,1}, b_{1,2}$)**
 475 function which returns $\{b'_{1,2}\}$ and it is poised to call **setValidBlock($b'_{1,1}, b'_{1,2}$)**.

476 We let p_1 continue to **append** new blocks until some time t_2 at which it is poised to
 477 call **setValidBlock($b'_{1,h}, b'_{1,h+1}$)**, with $h = 1$, such that the length of the chain $b_0, \dots, b'_{1,h+1}$

XX:12 On Finality in Blockchains

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

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

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

504 4.2 Asynchronous Solution to \mathcal{EF}^* with an Unbounded Number of 505 Byzantine Processes

506 We consider an asynchronous system with a possibly infinite set of processes which can
507 append infinitely many blocks, and processes can be affected by Byzantine failures. Each
508 process has a unique identifier $i \in \mathbb{N}$ and is equipped with signatures that can be used to
509 identify the message sender identifier. Each block is identified with the identifier of the
510 process that created it. Block identifier is inserted in the header of the block. Moreover, each
511 process is equipped with an Eventual BFT-Reliable Broadcast primitive. If a correct process
512 p broadcasts a message m then p eventually delivers m and if a correct process p delivers
513 m then all correct processes eventually deliver m . We assume the system is such that we
514 can implement an eventual reliable broadcast primitive, e.g., we assume that the infinite set
515 of processes are arranged in a topology in which for each pair of correct processes, there
516 exists a path composed by only correct processes [20]. Moreover, as proved in [1] reliable
517 communications are necessary for eventual finality. We show that in that setting it is possible
518 to build a blockchain that satisfies eventual prefix consistency.

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

523 To perform an `append()` operation of a block b , processes extend the chain returned
524 by function f_a applied on their current view of bt with b , i.e., $f_a(bt) \frown b$, and `rb-broadcast`

■ **Algorithm 1** \mathcal{EF}^* 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( $f_a(bt) \frown b$ )
6 end
7 upon read()
8   | return  $f_r(bt)$ 
9 end

```

525 $f_a(bt) \frown b$. When a process **rb-delivers** a blockchain bc , it calls **bt.addIfValid(bc)** that merges bc
526 with bt if the former is valid. By merging bc with bt we mean that for each block b_i of bc
527 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
528 added to the block of bt whose hash is equal to the one contained in b_i 's header. For **read()**
529 operations, processes return the chain selected by f_r on their current bt .

530 Given a blocktree bt , the append selection function f_a selects a chain in bt by going from
531 the root (i.e., genesis block) to a leaf, choosing at each fork b_i the edge to the child with
532 the lowest identifier. If more than one child have the same identifier (i.e., they have been
533 created by the same process), then all of them are ignored. If all the children have the same
534 identifier, then f_a returns the chain from the genesis block to b_i . Blocks are ranked by the
535 creator identifier, in the domain of the natural number and thus lower bounded by 0. Then
536 even though, an infinite number of blocks is added continuously to a fork, there is not, for
537 a given block, an infinite number of blocks with a smaller identifier. Thus eventually the
538 selection function f_a will always select the same prefix. Finally, since blocks are diffused
539 by a **rb-broadcast** primitive, eventually all correct processes will have the same view of the
540 blocktree. When a process invokes the **read()** operation, it returns the blockchain selected by
541 the read selection function f_r applied to its current view of the blocktree. By imposing that
542 $f_r = f_a$, then eventually all the processes, when reading, will select the same prefix.

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

546 **Proof.** We show by construction that Algorithm 1 solves \mathcal{EF}^* in an asynchronous system
547 with a possibly infinite set of processes which can append infinitely many blocks, and can
548 suffer an unbounded number of Byzantine failures. Intuitively, despite the unbounded number
549 of blocks in each fork, by the eventual reliable broadcast, eventually for each fork all correct
550 processes have the same block with the smallest identifier. Hence, by the read selection
551 function that at each fork selects the block with the smallest identifier in order to select the
552 chain to read, eventually, at all correct processes, function f_r returns the blockchain having
553 a common increasing prefix. Let p_1, p_2, \dots , be a possibly infinite set of processes, such that
554 each one maintains its local view bt_i of blocktree bt by running Algorithm 1. Then for any
555 correct process p_i the following properties hold.

- 556 ■ **Chain validity:** it is satisfied by function **bt.addIfValid(bc)** that merges blockchain bc to
557 bt_i only if the former is valid.
- 558 ■ **Chain integrity:** The **read()** operation returns the chain of blocks selected by function
559 f_r applied to bt_i . By Line 2 of Algorithm 1, only valid blocks are locally added to bt_i

560 once they have been reliably delivered. By Algorithm 1, the only place at which blocks
 561 are reliably broadcast is in the `append()` operation.

562 ■ **Eventual prefix:** The eventual prefix property follows from the definition of f_a and
 563 the reliable broadcast. Thanks to the reliable broadcast for any b in the bt of a correct
 564 process p , eventually all correct processes deliver b . Let t_b be the time after which no
 565 process can append further blocks b_{child} to b such that b_{child} is part of the chain returned
 566 by f_a . This time t_b always exists, as for each block b having potentially infinitely many
 567 children we have that, by definition of function f_a , $f_a(bt)$ selects a chain in bt by going
 568 from the root to a leaf, choosing at each fork b the edge to the child with the lowest
 569 identifier. Since identifiers are lower bounded by 0, eventually function f_a will always
 570 select the same child b' of b . The same argument applies for b' and its children. Hence,
 571 if any two correct processes invoke the read operation infinitely many times, then as
 572 $f_r = f_a$, eventually they return chains that satisfy the eventual prefix property.

573 ■ **Ever growing tree:** The ever-growing tree property relies on the fact that each fork
 574 has a finite number of blocks since there are finitely many processes and each (Byzantine
 575 or correct) process can contribute with at most one block per parent as multiple children
 576 created by the same process are ignored by f_a . Thus, eventually, new blocks contribute
 577 to the growth of the tree.

578

579 4.3 Eventually Synchronous Solution to $\mathcal{EF}^{\diamond c}$ with less than half of 580 Byzantine Processes

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

583 We propose an algorithm, called \mathcal{AF} for Accountable Forking. This algorithm is inspired
 584 by the Streamlet [7] algorithm. Streamlet [7] assumes the presence of less than a third of
 585 Byzantine processes and an eventual synchronous system with a known message delay Δ after
 586 GST. We weaken both of these assumptions to provide a solution to $\mathcal{EF}^{\diamond c}$ (or equivalently to
 587 the eventual immediate finality, see Theorem 10). In particular, we assume only a majority
 588 of correct processes, we do not explicitly use Δ and consider a slightly modified version of
 589 the protocol. In the following we first describe Streamlet and then present our protocol in
 590 terms of proposed modifications to Streamlet, before providing the proof.

591 **Streamlet protocol.** The Streamlet protocol works in an eventually synchronous system
 592 with a known message delay Δ and a finite set of n processes. In particular, before the Global
 593 Stabilisation Time (GST), message delays can be arbitrary; however, after GST, messages
 594 sent by correct processes are guaranteed to be received by correct processes within Δ time.¹

595 In Streamlet [7], each epoch, composed of 2Δ , has a designated leader chosen at random
 596 by a publicly known hash function. Each block b is labelled with the epoch ($b.epoch$) at
 597 which it has been created. This allows processes to establish if a block b has been created by
 598 a legitimate leader. The protocol works as follows:

- 599 ■ **Propose-Vote.** In every epoch:
- 600 ■ The epoch's designated leader proposes a new block (rb-broadcast it, rb-broadcast
 - 601 as defined in Section 4.2) extending from the longest notarized chain (defined in a
 - 602 moment) it has seen, if there are multiple then it breaks ties arbitrarily.

¹ Notice that, in Streamlet [7] there is not the notion of time but of round, which denotes a basic unit of time.

- 603 – Every process votes (rb-broadcast a vote) for the first proposal they see from the
604 epoch's leader, as long as the proposed block extends from (one of) the longest notarized
605 chain(s) that the voter has seen. A vote is a signature on the proposed block.
- 606 – When a block gains votes from at least $2n/3$ distinct processes, it becomes notarized.
607 A chain is notarized if its constituent blocks are all notarized.
- 608 ■ **Finalize.** Notarized does not mean final. If in any notarized chain, there are three
609 adjacent blocks with consecutive epoch numbers, the prefix of the chain up to the second
610 of the three blocks is considered final. When a block becomes final, all of its prefix must
611 be final too.

612 Our protocol \mathcal{AF} is such that for any given fork, correct processes can blame the process
613 that originates it, i.e, a Byzantine process creating a fork is accountable for it. \mathcal{AF} makes
614 the following two modifications to Streamlet. First, we only require that a block gains votes
615 from a majority of distinct processes to become notarized, which means that forks can occur.
616 The second modification goes deeper: if a fork occurs, then it is possible to detect Byzantine
617 processes and to exclude them from the voters. This is done as follows. When, two conflicting
618 chains are finalized, that is two finalized chains that are not the prefix of one another, then
619 processes look for inconsistent blocks. Two notarized blocks b, b' are inconsistent with one
620 another if one of the following two conditions hold:

- 621 ■ **Cond. 1.** b and b' share the same epoch, i.e, $b.epoch = b'.epoch$;
- 622 ■ **Cond. 2.** either $((b.epoch < b'.epoch) \text{ and } (b.height > b'.height))$ or $((b'.epoch < b.epoch)$
623 $\text{ and } (b'.height > b.height))$. Function height counts the number of blocks from the genesis
624 block.

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

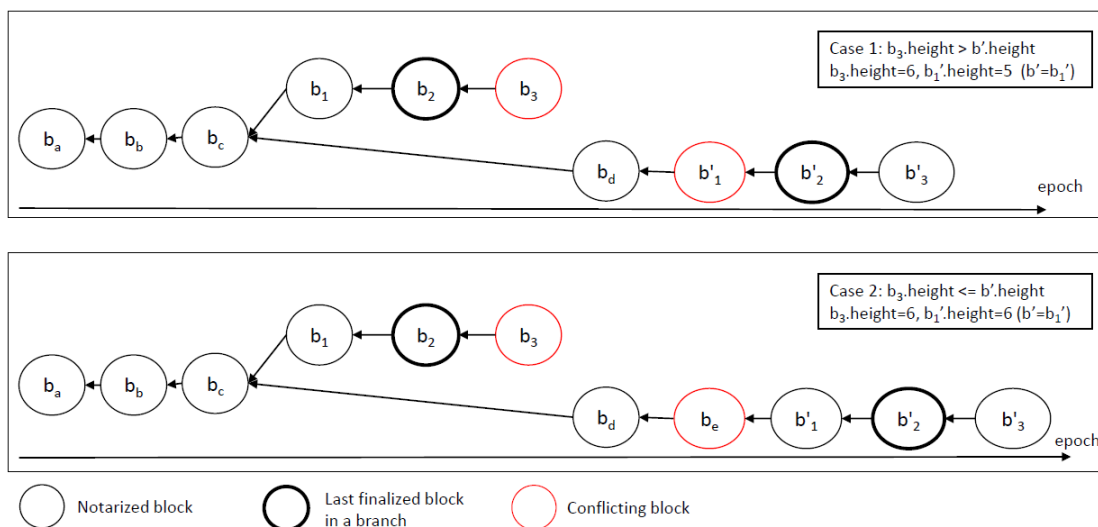
629 ► **Theorem 15.** *There exists a solution that solves unknown bounded revocation eventual*
630 *finality in an eventual synchronous system with less than $n/2$ Byzantine processes, where n*
631 *is the number of processes participating to the algorithm.*

632 **Proof.** Let us show that \mathcal{AF} is a solution for unknown bounded revocation eventual finality.

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

642 Let us now show that when a fork occurs we must have two inconsistent blocks. Indeed, if
643 there is a fork then we have two sequences of three adjacent blocks with consecutive epochs,
644 b_1, b_2, b_3 and b'_1, b'_2, b'_3 (by construction, given the finalization rule). If no blocks share the
645 same epoch number then we can assume w.l.o.g. that $b_3.epoch < b'_1.epoch$. Let block b'
646 belonging to the prefix of b'_3 such that $b'.epoch > b_1.epoch$ and $b'.height$ is the smallest in the
647 prefix of b'_3 . Such block always exists as b'_1 satisfies those two conditions. We have two cases:
648 Either $b'.height < b_3.height$ or $b'.height \geq b_3.height$. In the former case, b' is inconsistent

XX:16 On Finality in Blockchains



■ **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.

649 with b_3 since by assumption $b'.epoch > b_3.epoch$. In the latter case, the predecessor of b'
 650 is inconsistent with b_3 . Indeed, the predecessor of b' has a strictly smaller height than b_1
 651 and by assumption has a larger epoch number than b_3 . Figure 2 illustrates the presence
 652 of inconsistent blocks in presence of a fork at some block b_c . From b_c two chains are built,
 653 the first one consisting of the sequence of three blocks b_1, b_2 and b_3 , and the second chain
 654 consisting of four consecutive blocks b_d, b'_1, b'_2, b'_3 (to illustrate the first case) and of five
 655 consecutive blocks $b_d, b_e, b'_1, b'_2, b'_3$ (to illustrate the second case). In both cases block b'_1
 656 plays the role of block b' . In the first case (figure in the top), $b_3.height = 6$ and $b'.height = 5$
 657 while $b_3.epoch = 6$ and $b'.epoch = 5$. Thus Cond. 2 applies. In the second case (figure in
 658 the bottom), since $b'.height \geq b_3.height$ then there must exist some block b_e in the b' prefix.
 659 Thus $b_e.height < b'.height$. Moreover, given that by assumption $b_e.epoch > b_3.epoch$, then
 660 Cond. 2 holds for b_e and b_3 .

661 Hence there is always a couple of inconsistent blocks in a fork.

662 Let us now conclude our proof that we solve the eventual immediate finality. If a fork
 663 occurs, then each correct process eventually detects at least one Byzantine process and
 664 ignores its votes, hence, we have a finite number of forks as we have a finite number of
 665 Byzantine processes, hence eventually there is always a single chain that is finalized. As there
 666 is a majority of correct processes, our protocol \mathcal{S} remains live as in the original Streamlet
 667 protocol. \mathcal{S} also inherits the properties of the original Streamlet protocol for finalizing blocks
 668 eventually when synchrony is reached. ◀

669 5 Conclusion

670 In this work we have focused on the formalisation of eventual finality, which ensures that
 671 selected main chains at different processes share a common increasing prefix. We have
 672 formalised different forms of eventual finality in terms of the maximal number of blocks
 673 that can be revoked at each reconciliation, which is a crux in current blockchain designs.

674 We have formally shown that in an asynchronous system is not possible to reach a bound
675 on the number of blocks that can be revoked. On the other hand, we proposed for the
676 first time a solution in an eventually synchronous system with less than half of Byzantine
677 processes guaranteeing that such bound is reached eventually. We have also shown that in
678 an asynchronous system eventual finality with no bound on the number of revocable blocks
679 cannot be solved using the reconciliation rule of Bitcoin. Still we provide an asynchronous
680 solution with an unlimited number of Byzantine processes. We hope that these results will
681 better guide blockchain designs and link them to clear formal properties of finality.

682 ——— References ———

- 683 1 ANCEAUME, E., POZZO, A. D., LUDINARD, R., POTOP-BUTUCARU, M., AND TUCCI PIER-
684 GIOVANNI, S. Blockchain abstract data type. In Proceedings of the ACM Symposium on
685 Parallelism in Algorithms and Architectures (SPAA) (2019).
- 686 2 ANDROULAKI, E., AND ET AL. Hyperledger fabric: a distributed operating system for
687 permissioned blockchains. In Proceedings of the European Conference on Computer Systems
688 (EuroSys) (2018).
- 689 3 ANTA, A. F., KONWAR, K., GEORGIU, C., AND NICOLAOU, N. Formalizing and implementing
690 distributed ledger objects. ACM SIGACT News 49, 2 (2018), 58–76.
- 691 4 AȘTEFANOAEI, L., CHAMBART, P., POZZO, A. D., RIEUTORD, T., TUCCI-PIERGIOVANNI, S.,
692 AND ZĂLINESCU, E. Tenderbake - a solution to dynamic repeated consensus for blockchains.
693 In Proceedings of the Fourth International Symposium of Foundations and Applications of
694 Blockchain (2021).
- 695 5 BUTERIN, V., AND GRIFFITH, V. Casper the friendly finality gadget. CoRR (2017).
- 696 6 CACHIN, C. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote
697 Abstract). In Proceedings of the International Conference on Principles of Distributed Systems
698 (OPODIS) (2016).
- 699 7 CHAN, B. Y., AND SHI, E. Streamlet: Textbook streamlined blockchains. [https://eprint.
700 iacr.org/2020/088.pdf](https://eprint.iacr.org/2020/088.pdf), 2020.
- 701 8 CHEN, J., AND MICALI, S. Algorand: A secure and efficient distributed ledger. Theor. Comput.
702 Sci. (2019).
- 703 9 CRAIN, T., GRAMOLI, V., LARREA, M., AND RAYNAL, M. (leader/randomization/signature)-
704 free byzantine consensus for consortium blockchains. CoRR abs/1702.03068 (2017).
- 705 10 DUBOIS, S., GUERRAOU, R., KUZNETSOV, P., PETIT, F., AND SENS, P. The weakest failure
706 detector for eventual consistency. In Proceedings of the ACM Symposium on Principles of
707 Distributed Computing (PODC) (2015).
- 708 11 GARAY, J. A., KIAYIAS, A., AND LEONARDOS, N. The bitcoin backbone protocol: Analysis
709 and applications. In Proc. EUROCRYPT International Conference (2015).
- 710 12 GOODMAN, L. Tezos – a self-amending crypto-ledger, 2014.
- 711 13 GRIGG, I. EOS: An introduction. <https://whitepaperdatabase.com/eos-whitepaper/>.
- 712 14 GUERRAOU, R., KUZNETSOV, P., MONTI, M., PAVLOVIČ, M., AND SEREDINSCHI, D.-A.
713 The consensus number of a cryptocurrency. In Proceedings of the 2019 ACM Symposium on
714 Principles of Distributed Computing (PODC) (2019).
- 715 15 HERLIHY, M. Blockchains and the future of distributed computing. In Proceedings of the
716 ACM Symposium on Principles of Distributed Computing (PODC) (2017).
- 717 16 KIAYIAS, A., RUSSELL, A., DAVID, B., AND OLIYNYKOV, R. Ouroboros: A provably secure
718 proof-of-stake blockchain protocol. In Proceedings of the Advances in Cryptology (2017).
- 719 17 KOLTSOV, A., CHEREMENSKY, V., AND KAPULKIN, S. Casper White Paper.
- 720 18 LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. ACM
721 Transactions on Programming Languages and Systems (1982).
- 722 19 LISKOV, B., AND ZILLES, S. Programming with abstract data types. ACM SIGLAN Notices
723 9, 4 (1974).

XX:18 On Finality in Blockchains

- 724 20 MAURER, A., AND TIXEUIL, S. On byzantine broadcast in loosely connected networks. In
725 Proceedings of the 26th International Symposium on Distributed Computing (DISC) (2012).
- 726 21 NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org (2008).
- 727 22 NOMADIC LABS. Analysis of Emmy⁺. <https://blog.nomadic-labs.com/analysis-of-emmy.html>, 2019.
- 728
- 729 23 PASS, R., SEEMAN, L., AND SHELAT, A. Analysis of the blockchain protocol in asynchronous
730 networks. In Proceedings of the EUROCRYPT International Conference (2017).
- 731 24 PERRIN, M. Distributed Systems, Concurrency and Consistency. ISTE Press, Elsevier, 2017.
- 732 25 RAYNAL, M. Eventual leader service in unreliable asynchronous systems: Why? how? In
733 Proceedings of the IEEE International Symposium on Network Computing and Applications
734 (NCA) (2007).
- 735 26 STEWART, A. Poster: Grandpa finality gadget. In Proceedings of the 2019 ACM SIGSAC
736 Conference on Computer and Communications Security (2019), CCS '19, p. 2649–2651.
- 737 27 WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. <http://gavwood.com/Paper.pdf>.
- 738