



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. OPODIS 2021 - 25th Conference on Principles of Distributed Systems, Dec 2021, Strasbourg, France. cea-03080029v5

HAL Id: cea-03080029

<https://cea.hal.science/cea-03080029v5>

Submitted on 22 Nov 2021

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

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



Distributed under a Creative Commons Attribution 4.0 International License

1 On Finality in Blockchains*

2 **Emmanuelle Anceaume** ✉ 

3 CNRS, Univ Rennes, Inria, IRISA, [Rennes, France]

4 **Antonella Del Pozzo** ✉ 

5 Université Paris-Saclay, CEA, List, [F-91120, Palaiseau, France]

6 **Thibault Rieutord** ✉

7 Université Paris-Saclay, CEA, List, [F-91120, Palaiseau, France]

8 **Sara Tucci Piergiovanni** ✉ 

9 Université Paris-Saclay, CEA, List, [F-91120, Palaiseau, 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** Theory of computation

31 **Keywords and phrases** Blockchain, consistency properties, Byzantine tolerant implementations

32 **Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.52

33 **Category** Regular paper

* 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

81 notion of finality in a protocol-agnostic way. At the heart of the proposed formalisation lies
 82 the notion of *revocation number*. Informally, given a system run and a blockchain bc read by
 83 a user at some time t , we call the revocation number the natural number n such that by
 84 pruning the last n blocks from bc , we obtain a prefix of any blockchain bc' read after t .

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

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

94 Between \mathcal{F} and \mathcal{F}^c we then define three other forms of deferred finality: \mathcal{F}^n , where the
 95 revocation number is bounded but not known, $\mathcal{F}^{\diamond,c}$, where the revocation number is constant
 96 but holds only eventually, and finally $\mathcal{F}^{\diamond,n}$, where the bound on the revocation number is
 97 not known and holds only eventually. \mathcal{F}^n guarantees that finality of each block is deferred
 98 by a constant c in each system run, but this constant can vary from one run to another. For
 99 $\mathcal{F}^{\diamond,c}$ and $\mathcal{F}^{\diamond,n}$ we have that $\mathcal{F}^{\diamond,c}$ guarantees that eventually finality of each block is deferred
 100 by c in all system runs, while for $\mathcal{F}^{\diamond,n}$, eventually finality of each block is deferred by c in
 101 each system run with c varying from one run to another. Nicely, we obtain each consistency
 102 criterion by adding a proper bounded revocation property to \mathcal{F} and we prove that \mathcal{F}^n , $\mathcal{F}^{\diamond,c}$,
 103 $\mathcal{F}^{\diamond,n}$ are all equivalent.

104 The rigorous formalisation of these consistency criteria enables us to easily show that
 105 solutions that guarantee \mathcal{F}^c are equivalent to Consensus, while solutions that guarantee \mathcal{F}^n
 106 (or equivalently $\mathcal{F}^{\diamond,n}$ and $\mathcal{F}^{\diamond,c}$) are not weaker than Eventual Consensus, an abstraction
 107 that captures eventual agreement among all participants. From these reductions, we show
 108 some related impossibility results in presence of Byzantine processes. Beside reductions and
 109 related impossibilities, we propose the following non-trivial results:

- 110 ■ \mathcal{F} cannot be achieved in an asynchronous system if the reconciliation rule follows the
 111 "longest" chain rule (Theorem 20). This implies that the reconciliation rule, used in current
 112 blockchains to provide probabilistic finality in synchronous settings, cannot guarantee
 113 that participants will eventually converge to a stable prefix of the chain in asynchronous
 114 settings.
- 115 ■ A solution that guarantees \mathcal{F} in an asynchronous system with a possibly infinite set of
 116 processes which can append infinitely many blocks. This novel solution is simple and
 117 tolerant to an unbounded number of Byzantine processes (Theorem 21).
- 118 ■ A solution that solves \mathcal{F}^n in an eventually synchronous environment in presence of less
 119 than half of Byzantine processes (Theorem 22). The central point of our solution is to let
 120 correct processes blame each fork on a particular Byzantine process, which can then be
 121 excluded from the computation. Weakening the classic requirement of $< 1/3$ to $< 1/2$
 122 Byzantine processes makes such a solution well adapted to large scale adversarial systems.
 123 As for the previous one, we are not aware of any such solution in the literature.

124 We hope that these results will better guide blockchain designs and link them to clear
 125 formal properties of finality. Hence, in the remainder of this article, Section 2 situates our
 126 work with respect to similar ones. Section 3 formally presents the sequential specification of
 127 a blockchain and the formalisation of the different finality properties we may expect from a
 128 blockchain when concurrently accessed. Section 4 presents reductions between different forms

129 of finality, Consensus and Eventual Consensus. Section 5 first shows why \mathcal{F} is not solvable
 130 in an asynchronous environment when the "longest" chain rule is used, and then presents
 131 two original and simple algorithms that respectively solve \mathcal{F} and \mathcal{F}^n . Finally, Section 6
 132 concludes the paper.

133 **2** Related Work

134 Formalization of blockchains in the lens of distributed computing has been recognized as
 135 an extremely important topic [14]. Garay et al. [10] have been the first to analyze the
 136 Bitcoin backbone protocol and to define invariants this protocol has to satisfy to verify with
 137 high probability an eventual consistent prefix. The authors have analyzed the protocol in a
 138 synchronous system, while others, as for example Pass et al. [21], have extended this line
 139 of work considering a more adversarial network. In those works the specification of the
 140 consistency properties are protocol dependent and thus provide an abstraction level that
 141 does not allow us to model the blockchain as a shared object being agnostic of the way it is
 142 implemented. The objective we pursue throughout this work is to formalize the semantic
 143 of the interface between the blockchain and the users. To do so we consider the blockchain
 144 as a shared object, and thus the consistency properties are specified independently of the
 145 synchrony assumptions of underlying distributed system and the type of failures that may
 146 occur. By doing this, we offer a higher level of abstraction than well-known properties do.

147 This approach has been recently followed in particular by Anta et al. [3], Anceaume et
 148 al. [1] and Guerraoui et al. [13]¹. In Anta et al. [3], the authors propose a formalization of
 149 distributed ledgers, modeled as an ordered list of records along with implementations for
 150 sequential consistency and linearizability using a total order broadcast abstraction. Anceaume
 151 et al. [1] have captured the convergence process of two distinct classes of blockchain systems:
 152 the class providing strong prefix as [3] (for each pair of chains returned at two different
 153 processes, one is the prefix of the other) and the class providing eventual prefix, in which
 154 multiple chains can co-exist but the common prefix eventually converges. The authors of [1]
 155 show that to solve strong prefix the Consensus abstraction is needed, however they do not
 156 address solvability of eventual prefix and do not formalise finality. Interestingly, our notion
 157 of finality and bounded revocation is able to encompass the strong and the eventual prefix
 158 consistency properties of [1].

159 **3** Definitions

160 **3.1** Preliminary Definitions

161 We describe a blockchain object as an abstract data type which allows us to completely
 162 characterize a blockchain by the operations it exports [18]. The basic idea underlying the
 163 use of abstract data types is to specify shared objects using two complementary facets: a
 164 sequential specification that describes the semantics of the object, and a consistency criterion
 165 over concurrent histories, i.e. the set of admissible executions in a concurrent environment [22].
 166 Prior to presenting the blockchain abstract data type we first recall the formalization used
 167 to describe an abstract data type (ADT).

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

168 3.1.1 Abstract data types.

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

178 3.1.2 Concurrent histories of an ADT

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

181 ► **Definition 1. (Concurrent history H)** *The execution of a program that uses an abstract*
 182 *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*

- 183 ■ $\Sigma = A \cup (A \times B)$ is a countable set of operations;
- 184 ■ E is a countable set of events that contains all the ADT operations invocations and all
 185 ADT operation response events;
- 186 ■ $\Lambda : E \rightarrow \Sigma$ is a function which associates events to the operations in Σ ;
- 187 ■ \mapsto : is the process order, irreflexive order over the events of E . Two events $(e, e') \in E^2$
 188 are ordered by \mapsto if they are produced by the same process, $e \neq e'$ and e happens before e' ,
 189 that is denoted as $e \mapsto e'$.
- 190 ■ \prec : is the operation order, irreflexive order over the events of E . For each couple
 191 $(e, e') \in E^2$ if e' is the invocation of an operation occurred at time t' and e is the response
 192 of another operation occurred at time t with $t < t'$ then $e \prec e'$;
- 193 ■ \nearrow : is the program order, irreflexive order over E , for each couple $(e, e') \in E^2$ with $e \neq e'$
 194 if $e \mapsto e'$ or $e \prec e'$ then $e \nearrow e'$.

195 3.2 The blocktree ADT

196 We represent a blockchain as a tree of blocks. The same representation has been adopted
 197 in [1]. Indeed, while consensus-based blockchains prevent forks or branching in the tree of
 198 blocks, blockchain systems based on proof-of-work allow the occurrence of forks to happen
 199 hence presenting blocks under a tree structure. The blockchain object is thus defined as a
 200 blocktree abstract data type (Blocktree ADT).

201 3.2.1 Sequential Specification of the Blocktree ADT (BT-ADT)

202 A blocktree data structure is a directed rooted tree $bt = (V_{bt}, E_{bt})$ where V_{bt} represents a
 203 set of blocks and E_{bt} a set of edges such that each block has a single path towards the root
 204 of the tree b_0 called the genesis block. A branching in the tree is called a *fork*. Let \mathcal{BT} be
 205 the set of blocktrees, B be the countable and non empty set of uniquely identified blocks
 206 and let \mathcal{BC} be the countable non empty set of blockchains, where a blockchain is a path
 207 from a leaf of bt to b_0 . A blockchain is denoted by bc . The structure is equipped with two
 208 operations `append()` and `read()`. Operation `append(b)` adds block $b \notin bt$ to V_{bt} and adds the
 209 edge (b, b') to E_{bt} where $b' \in V_{bt}$ is returned by the append selection function $f_a()$ applied to

210 bt . Operation $read()$ returns the chain bc selected by the read selection function $f_r()$ applied
 211 to bt (note that in [1], the $read()$ and $append()$ operations are defined with a unique selection
 212 function). The read selection $f_r()$ takes as argument the blocktree and returns a chain of
 213 blocks, that is a sequence of blocks starting from the genesis block to a leaf block of the
 214 blocktree. The chain bc returned by a $read()$ operation r is called the blockchain, and is
 215 denoted by r/bc . The append selection function $f_a()$ takes as argument the blocktree and
 216 returns a chain of blocks. Function $last_block()$ takes as argument a chain of blocks and
 217 returns the last appended block of the chain. Only blocks satisfying some validity predicate
 218 P can be appended to the tree. Predicate P is an application-dependent predicate used to
 219 verify the validity of the chain obtained by appending the new block b to the chain returned
 220 by $f_a()$ (denoted by $f_a(bt) \frown b$). In Bitcoin for instance this predicate embeds the logic to
 221 verify that the obtained chain does not contain double spending or overspending transactions.
 222 Formally,

223 ► **Definition 2.** (Sequential specification of the Blocktree ADT) The Blocktree Abstract Data
 224 Type is the 6-tuple $BT - ADT = \{A = \{append(b), read()/bc \in \mathcal{BC}\}, B = \mathcal{BC} \cup \{\top, \perp\}, Z =$
 225 $\mathcal{BT}, \xi_0 = b_0, \tau, \delta\}$, where the transition function $\tau : Z \times A \rightarrow Z$ is defined by

$$226 \quad \tau(bt, read()) = bt$$

$$227 \quad \tau(bt, append(b)) = \begin{cases} (V_{bt} \cup \{b\}, E_{bt} \cup \{b, last_block(f_a(bt))\}) & \text{if } P(f_a(bt) \frown b) \\ bt & \text{otherwise,} \end{cases}$$
 228

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

$$230 \quad \delta(bt, read()) = f_r(bt)$$

$$231 \quad \delta(bt, append(b)) = \begin{cases} \top & \text{if } P(f_a(bt) \frown b) \\ \perp & \text{otherwise.} \end{cases}$$
 232

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

236 3.2.2 Concurrent Specification and Consistency Criteria of the 237 BlockTree ADT

238 The concurrent specification of the blocktree abstract data type is the set of its concurrent
 239 histories. A blocktree consistency criterion is a function that returns the set of concurrent
 240 histories admissible for the blocktree abstract data type. In this paper, we define different
 241 consistency criteria for the blocktree. We first define eventual finality, which is the weakest
 242 consistency criterion that we may expect from blockchains, along with the notion of block
 243 revocation. We then combine eventual finality with different forms of revocation to provide
 244 stronger consistency criteria. The presented family of consistency criteria is a comprehensive
 245 characterization of what we may expect from blockchains.

246 ► **Notation 1.**

- 247 ■ $E(a^*, r^*)$ is an infinite set containing an infinite number of $append()$ and $read()$ invocation
 248 and response events;
- 249 ■ $E(a, r^*)$ is an infinite set containing (i) a finite number of $append()$ invocation and
 250 response events and (ii) an infinite number of $read()$ invocation and response events;

251 ■ o_{inv} and o_{rsp} indicate respectively the invocation and response event of an operation o ;
 252 and in particular for the `read()` operation, r_{rsp}/bc denotes the returned blockchain bc
 253 associated with the response event r_{rsp} and for the `append()` operation $a_{inv}(b)$ denotes the
 254 invocation of the append operation having b as input parameter;

255 ■ $\text{length} : \mathcal{BC} \rightarrow \mathbb{N}$ denotes a monotonic increasing deterministic function that takes as input
 256 a blockchain bc and returns a natural number as length of bc . Increasing monotonicity
 257 means that $\text{length}(bc \frown \{b\}) > \text{length}(bc)$;

258 ■ We represent chain bc as an infinite list $b_0 b^* \perp^+$ of blocks, where the first block $bc[0] = b_0$,
 259 the genesis block, followed by block values b , and an infinite number of \perp values. Notation
 260 $bc[i]$ refers to the i -th block of blockchain bc . Note that the special “ \perp ” symbol counts for
 261 zero for the length function.

262 ■ $bc \sqsubseteq bc'$ if and only if bc prefixes bc' . The operator \sqsubseteq ignores all the records set to \perp .

263 ► **Definition 3** (BT Eventual Finality Consistency criterion (\mathcal{F})). A concurrent history
 264 $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ of a system that uses a BT-ADT verifies the BT eventual finality
 265 consistency criterion if the following four properties hold:

266 ■ **Chain validity:**

$$\forall r_{rsp} \in E, P(r_{rsp}/bc).$$

268 Each returned chain is valid.

269 ■ **Chain integrity:**

$$\forall r_{rsp} \in E, \forall b \in r_{rsp}/bc : b \neq b_0, \exists a_{inv}(b) \in E, a_{inv}(b) \nearrow r_{rsp}.$$

271 If a block different from the genesis block is returned, then an `append` operation has been
 272 invoked with this block as parameter. This property is to avoid the situation in which
 273 reads return blocks never appended.

274 ■ **Eventual prefix:**

$$\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, \forall i \in \mathbb{N} : bc[i] \neq \perp, \exists r'_{rsp} : r_{rsp} \nearrow r'_{rsp}, \forall r''_{rsp} : r'_{rsp} \nearrow r''_{rsp}, (r'_{rsp}/bc')[i] = (r''_{rsp}/bc'')[i] \neq \perp.$$

277 In all the histories in which the number of read invocations is infinite, then for any read
 278 operation such that the returned chain has a block at position i , there exists a read r'/bc'
 279 from which all the subsequent reads r''/bc'' will return the same block at position i , i.e.
 280 $bc'[i] = bc''[i] \neq \perp$.

281 ■ **Ever growing tree:**

$$\forall E \in E(a^*, r^*), \forall k \in \mathbb{N}, \exists r \in E : \text{length}(r_{rsp}/bc) > k.$$

283 In all the histories in which the number of `append` and `read` invocations is infinite, for
 284 each length k , there exists a `read` that returns a chain with length greater than k . This
 285 property avoids the trivial scenario in which the length of the chain remains unchanged
 286 despite the occurrence of an infinite number of `append` operations (i.e., tree built as a star
 287 with infinite branches of bounded length). Specifically the “Ever growing tree” property
 288 imposes that in presence of an infinite number of `read` and `append` operations, for any
 289 natural number k , there will always exist a `read` operation that will return a chain of at
 290 least length k . Note that the well known “Chain Growth Property” [10, 21] states that
 291 each (honest) chain grows proportionally with the number of rounds of the protocol, which
 292 in contrast to our specification, makes it protocol dependent.

293 Bounded revocation

294 As previously said, the bounded revocation properties are at the heart of our formalisation
 295 of blockchain finality. Informally, given a history, we call the revocation number the natural

296 number n such that for any two reads r/bc and r'/bc' , where r precedes r' , by pruning the
 297 last n blocks from bc we obtain a chain that is a prefix of bc' .

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

300 To obtain stronger consistency criteria, we then introduce restrictions to the revocation
 301 number. To this aim, we define the c -bounded revocation property, which states that the
 302 revocation number n is bounded by a constant c in all histories. We also define the *bounded*
 303 *revocation property*, which states that the revocation number n is bounded by a constant
 304 c in each history, but may be unbounded when we consider the union of all the histories,
 305 i.e., the bound can vary from a history to another. Eventual forms of c -bounded revocation
 306 and bounded revocation state that the revocation number will be equal to a constant c only
 307 eventually. More formally:

308 ► **Definition 4** (c -Bounded Revocation). $\exists c \in \mathbb{N}, \forall E, \forall r_{rsp}/bc, r'_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in$
 309 $\mathbb{N} : i \leq (\text{length}(bc) - c), bc[i] = bc'[i] \neq \perp$.

310 ► **Definition 5** (Bounded Revocation). $\forall E, \exists c \in \mathbb{N}, \forall r_{rsp}/bc, r'_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in$
 311 $\mathbb{N} : i \leq (\text{length}(bc) - c), bc[i] = bc'[i] \neq \perp$.

312 ► **Definition 6** (Eventual c -Bounded Revocation). $\exists c \in \mathbb{N}, \forall E, \exists r \in E : \forall r'_{rsp}/bc, r''_{rsp}/bc' \in$
 313 $E : r_{rsp} \nearrow r'_{rsp}, r'_{rsp} \nearrow r''_{rsp}, \forall i \in \mathbb{N} : i \leq (\text{length}(bc') - c), bc'[i] = bc''[i] \neq \perp$

314 ► **Definition 7** (Eventual Bounded Revocation). $\forall E, \exists c \in \mathbb{N}, \exists r \in E : \forall r'_{rsp}/bc, r''_{rsp}/bc' \in E :$
 315 $r_{rsp} \nearrow r'_{rsp}, r'_{rsp} \nearrow r''_{rsp}, \forall i \in \mathbb{N} : i \leq (\text{length}(bc') - c), bc'[i] = bc''[i] \neq \perp$

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

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

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

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

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

336 In the following we formally introduce \mathcal{F}^c , \mathcal{F}^n , $\mathcal{F}^{\diamond, c}$, $\mathcal{F}^{\diamond, n}$, and show equivalences between
 337 $\mathcal{F}^{\diamond, c}$, $\mathcal{F}^{\diamond, n}$ and \mathcal{F}^n .

338 ► **Definition 8** (BT c -Deferred Finality Consistency criterion (\mathcal{F}^c)). *A concurrent history*
 339 $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$ *of the system that uses a BT-ADT verifies the BT c -deferred finality*
 340 *consistency criterion if chain validity, chain integrity, eventual prefix, ever growing tree, and*
 341 *the c -bounded revocation properties hold.*

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

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

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

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

358 ▶ **Notation 2.** For readability reasons, in the following we will simply say *finality* instead of
 359 *finality consistency criterion*.

360 ▶ **Theorem 12.** \mathcal{F}^n and $\mathcal{F}^{\diamond, n}$ are equivalent.

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

367 We now show that $\mathcal{F}^{\diamond, n}$ and $\mathcal{F}^{\diamond, c}$ are equivalent. This equivalence is shown by first
 368 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.

369 ▶ **Theorem 13.** $\mathcal{F}^{\diamond, c=0}$ and $\mathcal{F}^{\diamond, n}$ are equivalent.

370 **Proof.** Let \mathcal{P}_1 be a protocol guaranteeing $\mathcal{F}^{\diamond, n}$. We build protocol \mathcal{P}_2 as follows: to make
 371 an `append()` operation, processes simply use the `append()` operation of \mathcal{P}_1 . For the `read()`
 372 operation, processes use the `read()` operation provided by \mathcal{P}_1 to obtain the blockchain and
 373 prune the second half of it before returning the first half of the blockchain. Let us show that
 374 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
 375 satisfied:

- 376 ■ **Chain validity:** The chain validity property is still satisfied by pruning half of the chain.
- 377 ■ **Chain integrity:** The chain integrity property is still satisfied by pruning half of the
 378 chain.
- 379 ■ **Eventual prefix:** The eventual prefix property is still satisfied by pruning half of the
 380 chain.
- 381 ■ **Ever growing tree:** The ever growing tree property is still satisfied by pruning half of
 382 the chain.
- 383 ■ **($c = 0$)-eventual bounded revocation:** This property follows from the removal of the
 384 second half of the chain. Indeed, if we remove the second half of the chain, then eventually
 385 for any two `read()` operations, then the first `read()` returns a prefix of the second `read()`
 386 operation.

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

388 ▶ **Theorem 14.** $\mathcal{F}^{\diamond,c=0}$ and $\mathcal{F}^{\diamond,c}$ are equivalent.

389 **Proof.** Trivially, $\mathcal{F}^{\diamond,c=0}$ implies $\mathcal{F}^{\diamond,c}$. For the other direction, we apply a construction close
 390 to the one used in the proof of Theorem 13. Specifically, given a protocol \mathcal{P}_1 that guarantees
 391 $\mathcal{F}^{\diamond,c}$, we build a protocol \mathcal{P}_2 by using \mathcal{P}_1 as follows. To make an `append()` operation,
 392 processes simply use the `append()` operation of \mathcal{P}_1 . For the `read()` operation, processes use
 393 the `read()` operation provided by \mathcal{P}_1 to obtain the blockchain and prune its last c blocks
 394 before returning it. Note that if there are less than c blocks, processes then return the genesis
 395 block. The properties of $\mathcal{F}^{\diamond,c=0}$ trivially follow from the properties of $\mathcal{F}^{\diamond,c}$ and the proposed
 396 transformation. ◀

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

398 **Proof.** Straightforward from Theorems 12, 13 and 14. ◀

399 4 (Eventual) Consensus Reductions

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

403 4.1 c -Bounded Deferred Finality and Consensus

404 ▶ **Theorem 16.** *Guaranteeing \mathcal{F}^c is equivalent to solving Consensus.*

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

407 To prove the theorem it is then sufficient to give a protocol \mathcal{P}_2 that guarantees $\mathcal{F}^{c=0}$ given
 408 a solution \mathcal{P}_1 that satisfies \mathcal{F}^c , the other direction being trivial. We build \mathcal{P}_2 by applying
 409 the same transformation of \mathcal{P}_1 described in the proof of Theorem 14. The properties of $\mathcal{F}^{c=0}$
 410 trivially follow from the properties of \mathcal{F}^c and the proposed transformation. ◀

411 ▶ **Corollary 17.** *There does not exist any solution that solves \mathcal{F}^c in an eventual synchronous
 412 system with more than 1/3 of Byzantine processes.*

413 **Proof.** The proof follows from the equivalence between \mathcal{F}^c and Consensus (cf. Theorem 16),
 414 which is unsolvable in a synchronous (and thus also in an eventually synchronous) system
 415 with more than one third of Byzantine processes [17]. ◀

416 4.2 Bounded Deferred Finality and Eventual Consensus

417 In this section we show that guaranteeing bounded deferred finality is not weaker than
 418 Eventual Consensus. To this aim we first recall the Eventual Consensus problem with a
 419 small modification of the validity property to make it suitable to the blockchain context and
 420 then we show that $\mathcal{F}^{\diamond,c=0}$ (which is equivalent to $\mathcal{F}^{\diamond,n}$ by Corollary 15) is not weaker than
 421 Eventual Consensus.

422 The Eventual Consensus (EC) abstraction [9] captures eventual agreement among all
 423 participants. It exports, to every process p_i , operations `proposeEC`₁, `proposeEC`₂, ... that
 424 take multi-valued arguments (correct processes propose valid values) and return multi-valued
 425 responses. Assuming that, for all $j \in \mathbb{N}$, every process invokes `proposeEC`_j as soon as it

426 returns a response to `proposeECj-1`, the abstraction guarantees that, in every admissible run,
 427 there exists $k \in \mathbb{N}$ and a predicate P_{EC} , such that the following properties are satisfied:

- 428 ■ **EC-Termination.** Every correct process eventually returns a response to `proposeECj`
 429 for all $j \in \mathbb{N}$.
- 430 ■ **EC-Integrity.** No process responds twice to `proposeECj` for all $j \in \mathbb{N}$.
- 431 ■ **EC-Validity.** Every value returned to `proposeECj` is valid with respect to predicate P_{EC} .
- 432 ■ **EC-Agreement.** No two correct processes return different values to `proposeECj` for all
 433 $j \geq k$.

434 ► **Theorem 18.** *Guaranteeing $\mathcal{F}^{\diamond, c=0}$ (or any of the equivalent forms) is not weaker than*
 435 *solving Eventual Consensus.*

436 **Proof.** We show that there exists a protocol \mathcal{P}_1 that solves Eventual Consensus assuming
 437 the existence of a protocol \mathcal{P}_2 that solves $\mathcal{F}^{\diamond, c=0}$. We do the transformation as follows. Every
 438 correct process p invokes `proposeECj` for all $j \in \mathbb{N}$. We impose that the validity predicate P
 439 of the blocktree ADT (see Section 3) be equal to predicate P_{EC} . When a correct process
 440 p invokes the `proposeECj(v)` operation of \mathcal{P}_1 , for any $j \in \mathbb{N}$, then p executes the following
 441 sequence of three steps: (i) p invokes the `append(v)` operation of \mathcal{P}_2 , then (ii) p invokes
 442 a sequence of `read()` operations up to the moment the `read()` returns a chain bc such that
 443 $bc[j] \neq \perp$, and finally (iii) p decides chain bc (i.e., it returns chain bc) and triggers the next
 444 operation `proposeECj+1(v')`. We now show that protocol \mathcal{P}_1 solves Eventual Consensus.

- 445 ■ **EC-Termination** This property is guaranteed by the ever growing tree property.
- 446 ■ **EC-Integrity** This property follows directly from the transformation.
- 447 ■ **EC-Validity** This property follows by construction and by the chain validity property
 448 since predicate P equals to predicate P_{EC} .
- 449 ■ **EC-Agreement** This property follows by the eventual prefix property and the 0-eventual
 450 revocation property, which guarantees that there exists a `read()` operation r such that all
 451 the subsequent ones return blockchains that are each prefix of the following one. In other
 452 words, eventually there is agreement on the value contained in $bc[j]$. This implies that
 453 there exists k for which all `proposeECj` with $j \geq k$ return the same value to all correct
 454 processes.

455 Finally, by Corollary 15, the proof of the Theorem completes. ◀

456 ► **Theorem 19.** *There does not exist any solution that solves \mathcal{F}^n (and any of the equivalent*
 457 *forms) in an asynchronous system with at least one Byzantine process.*

458 **Proof.** The proof follows from Corollary 15 and the fact that $\mathcal{F}^{\diamond, c=0}$ is not weaker than
 459 Eventual Consensus (cf. Theorem 18). Since Eventual Consensus is equivalent to the leader
 460 election problem [9], which cannot be solved in an asynchronous system with at least one
 461 Byzantine process [23], this completes the proof of the Theorem. ◀

462 5 Finality Solutions

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

467 5.1 Impossibility to Satisfy \mathcal{F} with the Longest Chain Rule

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

480 To show this impossibility result, we consider a scenario in which the occurrence of any
 481 fork produces at most two alternative chains (this is often referred to as a branching factor
 482 of 2). We consider a finite number of processes and an append selection function f_a that
 483 in case of forks deterministically selects the longest chain through the length function (see
 484 Section 3.2.2), and in case of a tie selects the chain following any deterministic rule (for
 485 instance the chain whose last block has the smallest digest). We show that it is impossible
 486 to guarantee \mathcal{F} for such append selection function f_a .

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

493 ► **Theorem 20.** *It is impossible to guarantee \mathcal{F} if the append operation is based on the*
 494 *longest chain rule in an asynchronous environment.*

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

496 5.2 Asynchronous Solution Satisfying \mathcal{F} with an Unbounded Number 497 of Byzantine Processes

498 We consider an asynchronous system with a possibly infinite set of processes which can
 499 append infinitely many blocks, and processes can be affected by Byzantine failures. Each
 500 process has a unique identifier $i \in \mathbb{N}$ and is equipped with signatures that can be used to
 501 identify the message sender identifier. Each block is identified with the identifier of the
 502 process that created it. Block identifier is inserted in the header of the block. Moreover, since
 503 it has been proven that reliable communications are necessary to ensure eventual finality [1],
 504 we assume that each process is equipped with an Eventually Reliable Broadcast primitive
 505 that satisfies the following two properties: If a correct process p broadcasts a message m
 506 then p eventually delivers m and if a correct process p delivers m then all correct processes
 507 eventually deliver m . Such a primitive can be implemented by organizing the infinite set
 508 of processes in a topology in which for each pair of correct processes, there exists a path
 509 composed by only correct processes [19]. Thus, we do not require any assumptions on the
 510 proportion between Byzantine and correct processes in the system but on the way those
 511 processes are arranged on the network topology.

■ **Algorithm 1** Guaranteeing \mathcal{F} 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

```

512 The main idea of Algorithm 1 consists in using local selection functions f_a and f_r for
513 **append** and **read** operations respectively and characterizing blocks by their parents and
514 producer signatures.

515 To perform an **append** operation of a block b , correct processes extend the chain returned
516 by function f_a applied on their current view of bt with b , i.e., $f_a(bt) \frown b$, and **rb-broadcast**
517 $f_a(bt) \frown b$. When a process **rb-delivers** a blockchain bc , it calls **bt.addIfValid(bc)** that merges bc
518 with bt if the former is valid. By merging bc with bt we mean that for each block b_i of bc
519 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
520 added to the block of bt whose hash is equal to the one contained in b_i 's header. A **read()**
521 operation triggered by a correct process p returns the chain selected by f_r on the current
522 blocktree bt of p . Given a blocktree bt , the **append** selection function f_a selects a chain in bt
523 by going from the root (i.e., genesis block) to a leaf, choosing at each fork b_i the edge to the
524 child with the lowest identifier. If more than one child have the same identifier (i.e., they
525 have been created by the same process), then all of them are ignored. If all the children have
526 the same identifier, then f_a returns the chain from the genesis block to b_i . Blocks are ranked
527 by the creator identifier, in the domain of the natural number and thus lower bounded by 0.
528 Then even though, an infinite number of blocks is added continuously to a fork, there is not,
529 for a given block, an infinite number of blocks with a smaller identifier. Thus eventually the
530 selection function f_a will always select the same prefix. Finally, since blocks are diffused by
531 an eventually reliable broadcast primitive, eventually all correct processes will have the same
532 view of the blocktree. When a process invokes the **read()** operation, it returns the blockchain
533 selected by the read selection function f_r applied to its current view of the blocktree. By
534 imposing that $f_r = f_a$, then eventually all the processes, when reading, will select the same
535 prefix.

536 ► **Theorem 21.** *Algorithm 1 is a solution satisfying \mathcal{F} in an asynchronous system with a*
537 *possibly infinite set of processes which can append infinitely many blocks, and suffer from an*
538 *unbounded number of Byzantine failures.*

539 **Proof.** We show by construction that Algorithm 1 solves \mathcal{F} in an asynchronous system with
540 a possibly infinite set of processes which can append infinitely many blocks, and can suffer
541 an unbounded number of Byzantine failures. Intuitively, despite the unbounded number of
542 blocks in each fork, by the eventually reliable broadcast, eventually for each fork all correct
543 processes have the same block with the smallest identifier. Hence, by the read selection
544 function f_r that at each fork selects the block with the smallest identifier in order to select
545 the chain to return, eventually, at all correct processes, function f_r returns the blockchain
546 having a common increasing prefix. Let p_1, p_2, \dots , be a possibly infinite set of processes,

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

549 ■ **Chain validity:** it is satisfied by function $\text{bt.addIfValid}(bc)$ that merges blockchain bc to
 550 bt_i only if the former is valid.

551 ■ **Chain integrity:** The $\text{read}()$ operation returns the chain of blocks selected by function
 552 f_r applied to bt_i . By Line 2 of Algorithm 1, only valid blocks are locally added to bt_i
 553 once they have been reliably delivered. By Algorithm 1, the only place at which blocks
 554 are reliably broadcast is in the $\text{append}()$ operation.

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

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

570 ◀

571 5.3 Eventually Synchronous Solution Satisfying Bounded Deferred 572 Finality with less than half of Byzantine Processes

573 In this section we prove that the bounded deferred finality is solvable in an eventually
 574 synchronous message-passing system with less than $n/2$ Byzantine processes, where n is the
 575 number of processes.

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

583 **The Streamlet Algorithm.** The Streamlet algorithm works in an eventually synchron-
 584 ous system with a known message delay Δ and a finite set of n processes. In particular,
 585 before the Global Stabilisation Time (GST), message delays can be arbitrary; however, after
 586 GST, messages sent by correct processes are guaranteed to be received by correct processes
 587 within Δ time units. Each epoch, composed of 2Δ time units, has a designated leader chosen
 588 at random by a publicly known hash function. Each block b is labelled with the epoch
 589 ($b.\text{epoch}$) at which it has been created. This allows processes to determine whether block b
 590 has been created by a legitimate leader. Figure 1 presents Steamlet protocol [6].

591 **The Accountable Forking (\mathcal{AF}) Algorithm.** We propose \mathcal{AF} , an algorithm that
 592 extends Streamlet. \mathcal{AF} guarantees that for any given fork, correct processes can blame
 593 the process that originates it, i.e, a Byzantine process creating a fork is accountable for it.

- **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.

■ **Figure 1** Streamlet algorithm [6]

594 This is achieved as follows: First, we only require that a block gains votes from a majority
 595 of distinct processes to become notarized, which means that forks can occur. The second
 596 modification we propose goes deeper: if a fork occurs, any correct processes can detect the
 597 Byzantine process that originated it, and excludes it from the voters. Specifically, when
 598 two conflicting chains are finalized (i.e., two finalized chains that are not the prefix of one
 599 another) then processes look for inconsistent blocks. By definition, two notarized blocks b, b'
 600 are inconsistent with one another if one of the following two conditions holds:

- 601 ■ **Condition 1.** b and b' share the same epoch, i.e. $b.epoch = b'.epoch$;
- 602 ■ **Condition 2.** either $((b.epoch < b'.epoch) \text{ and } (b.height > b'.height))$ or $((b'.epoch <$
 603 $b.epoch) \text{ and } (b'.height > b.height))$. Function height counts the number of blocks from
 604 the genesis block.

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

609 ► **Theorem 22.** *There exists a solution that satisfies $\mathcal{F}^{\diamond, c=0}$ (and all the equivalent forms)*
 610 *in an eventually synchronous system with less than half Byzantine processes.*

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

612 **6 Conclusion**

613 In this work we have defined different consistency criteria for blockchains. We have first
 614 defined eventual finality, which is the weakest consistency criterion that we may expect from
 615 blockchains, along with the notion of block revocation. By combining eventual finality with
 616 different forms of revocation we obtained stronger consistency criteria, thus providing a
 617 comprehensive characterization of what we may expect from blockchains. We have formally
 618 shown that in an asynchronous system it is not possible to provide a known bound on
 619 the number of blocks that can be revoked. On the other hand, we have proposed for the
 620 first time a solution in an eventually synchronous system with less than half of Byzantine
 621 processes guaranteeing that eventually such bound is reached. We have also shown that in
 622 an asynchronous system, finality with no bound on the number of revocable blocks cannot

623 be solved using the reconciliation rule of Bitcoin. Still we provide an asynchronous solution
 624 with an unlimited number of Byzantine processes. We hope that this work will better guide
 625 blockchain designs.

626 — **References** —

- 627 **1** Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru,
 628 and Sara Tucci Piergiovanni. Blockchain abstract data type. In *Proceedings of the ACM*
 629 *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.
- 630 **2** Elli Androulaki and et al. Hyperledger fabric: a distributed operating system for permissioned
 631 blockchains. In *Proceedings of the European Conference on Computer Systems (EuroSys)*,
 632 2018.
- 633 **3** Antonio Anta Fernández, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. Formaliz-
 634 ing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.
- 635 **4** Lăcrămioara Aștefanoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara
 636 Tucci-Piergiovanni, and Eugen Zălinescu. Tenderbake - a solution to dynamic repeated
 637 consensus for blockchains. In *Proceedings of the Fourth International Symposium of Foundations*
 638 *and Applications of Blockchain*, 2021.
- 639 **5** Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, 2017.
- 640 **6** Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. <https://eprint.iacr.org/2020/088.pdf>, 2020.
- 641 **7** Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor.*
 642 *Comput. Sci.*, 2019.
- 643 **8** Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal.
 644 (leader/randomization/signature)-free byzantine consensus for consortium blockchains.
 645 *CoRR*, abs/1702.03068, 2017. URL: <http://arxiv.org/abs/1702.03068>.
- 646 **9** Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest
 647 failure detector for eventual consistency. In *Proceedings of the ACM Symposium on Principles*
 648 *of Distributed Computing (PODC)*, 2015.
- 649 **10** Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis
 650 and applications. In *Proc. EUROCRYPT International Conference*, 2015. Updated version
 651 2020: <https://eprint.iacr.org/2014/765.pdf>.
- 652 **11** L.M. Goodman. Tezos – a self-amending crypto-ledger, 2014.
- 653 **12** Ian Grigg. EOS: An introduction. <https://whitepaperdatabase.com/eos-whitepaper/>.
- 654 **13** Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Sere-
 655 dinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium*
 656 *on Principles of Distributed Computing (PODC)*, 2019.
- 657 **14** Maurice Herlihy. Blockchains and the future of distributed computing. In *Proceedings of the*
 658 *ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.
- 659 **15** Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros:
 660 A provably secure proof-of-stake blockchain protocol. In *Proceedings of the Advances in*
 661 *Cryptology*, 2017.
- 662 **16** Artem Koltsov, Vitaly Chermensky, and Stanislav Kapulkin. Casper White Paper.
- 663 **17** Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM*
 664 *Transactions on Programming Languages and Systems*, 1982.
- 665 **18** B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGLAN Notices*, 9(4),
 666 1974.
- 667 **19** Alexandre Maurer and Sébastien Tixeuil. On byzantine broadcast in loosely connected networks.
 668 In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, 2012.
- 669 **20** Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org, 2008.
- 670 **21** Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous
 671 networks. In *Proceedings of the EUROCRYPT International Conference*, 2017.

- 673 22 Matthieu Perrin. *Distributed Systems, Concurrency and Consistency*. ISTE Press, Elsevier,
674 2017.
- 675 23 Michel Raynal. Eventual leader service in unreliable asynchronous systems: Why? how? In
676 *Proceedings of the IEEE International Symposium on Network Computing and Applications*
677 *(NCA)*, 2007.
- 678 24 Alistair Stewart. Poster: Grandpa finality gadget. In *Proceedings of the 2019 ACM SIGSAC*
679 *Conference on Computer and Communications Security, CCS '19*, page 2649–2651, 2019.
- 680 25 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. [http://](http://gawwood.com/Paper.pdf)
681 gawwood.com/Paper.pdf.

682 **Appendix**

683 **Theorem 20** It is impossible to guarantee \mathcal{F} if the `append` operation is based on the longest
684 chain rule in an asynchronous environment.

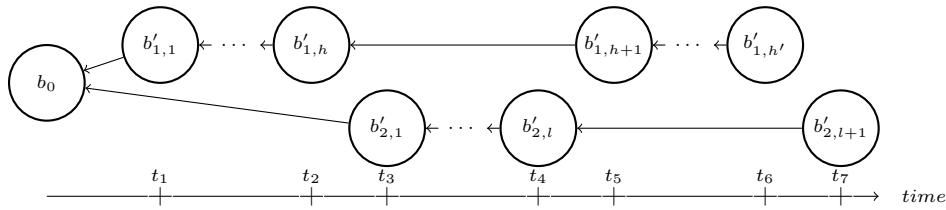
685 **Proof.** To capture the synchronisation power of the system, we abstract the deterministic
686 creation of new blocks and their addition to the blockchain within an oracle. This oracle
687 is the only generator of valid blocks, and regulates the number of appended children from
688 a same parent. The same approach has been proposed in [1]. The branching factor of an
689 oracle is the maximal number of children that can be appended to a block. The oracle owns
690 a synchronization power equal to Consensus if its branching factor is equal to 1. The oracle
691 grants access to the blocktree as a shared object, through the following three operations:
692 `update_view()` returns the current state of the blocktree; `getValidBlock(b_i, b_j)` returns a valid
693 block b'_j , constructed from b_j , that can be appended to block b_i , where b_i is already included
694 in the blocktree; and `setValidBlock(b_i, b'_j)` appends the valid block b'_j to b_i , and returns \top
695 when the block is successfully appended and \perp otherwise. The following theorem shows that,
696 even with this strong oracle (that allows to have a bounded branching factor in contrast to
697 proof-of-work (PoW) approaches), we cannot reach eventual finality if we rely on the longest
698 chain rule to resolve forks.

699 In the proof we consider the stronger oracle allowing the occurrence of one fork, i.e., an
700 oracle with branching factor equal to 2. That is, this oracle allows for two valid blocks to
701 be appended to the same parent. If the oracle receives new requests to append additional
702 blocks to this parent, it shall return \perp to all such requests.

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

706 We illustrate our proof with Figure 2. At time t_0 , for both p_1 and p_2 , the `update_view()`
707 of bt equals b_0 , thus when both apply the append selection function f_a on it to select the leaf
708 where to `append` the new block, they both get b_0 . Then they both call `getValidBlock($b_0, b_{i,1}$) =`
709 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
710 `setValidBlock($b_0, b'_{i,1}$)`. We then let p_1 call `setValidBlock($b_0, b'_{1,1}$)`, which must return \top and
711 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
712 having updated its bt 's view, through the `update_view()` function, p_1 applies the append
713 selection function f_a on it to select the leaf where to `append` its new block, in this case the
714 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
715 to call `setValidBlock($b'_{1,1}, b'_{1,2}$)`.

716 We let p_1 continue to append new blocks until some time t_2 at which it is poised to
717 call `setValidBlock($b'_{1,h}, b'_{1,h+1}$)`, with $h = 1$, such that the length of the chain $b_0, \dots, b'_{1,h+1}$
718 would be greater than or would have the same length but a larger lexicographical order than
719 the chain $b_0, b'_{2,1}$ if $b'_{2,1}$ were already appended to block b_0 . Afterwards, at time $t_3 \geq t_2$,
720 we let p_2 resume and complete its call to `setValidBlock($b_0, b'_{2,1}$)` which must also succeed
721 and return \top as our oracle has a branching factor of 2. By construction, p_2 sees the two
722 branches in its following `update_view()` of bt (i.e., chain $b_0, b'_{1,h}$ with $h = 1$, and chain $b_0, b'_{2,1}$)
723 of the same length thus the selection function f_a selects the branch $b_0, b'_{2,1}$ for where to
724 append the next block as block $b'_{2,1}$ is smaller than $b'_{1,h}$ in the lexicographical order. We
725 let p_2 append blocks to this branch until some time t_4 at which it becomes poised to call
726 `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
727 than the chain $b_0, \dots, b'_{1,h+1}$, or in case of equal length has a higher lexicographical order,
728 and such that the length of the chain $b_0, \dots, b'_{2,c+1}$ is greater than the chain $b_0, \dots, b'_{1,h+1}$,



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

729 or in case of equal length has a smaller lexicographical order.

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

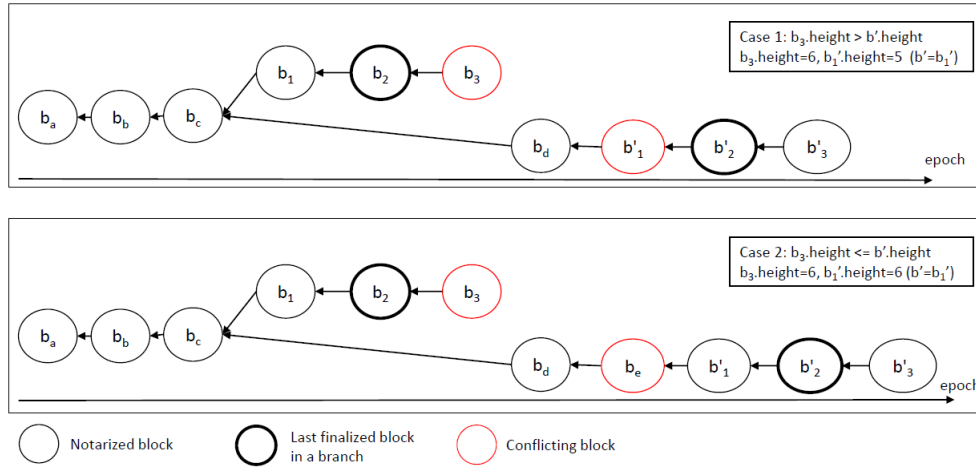
740 It is important to note that with any `length` function that increases monotonically with
 741 prefixes (e.g, the length function could count the total number of transactions that belong to
 742 the blocks on the same branch) then this scenario still holds. In that case h and c in the
 743 proof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

744

745 ► **Theorem 22.** *There exists a solution that satisfies $\mathcal{F}^{\diamond, c=0}$ (and all the equivalent forms)*
 746 *in an eventually synchronous system with less than half Byzantine processes.*

747 **Proof.** Let us first demonstrate that voting for two inconsistent blocks b and b' is a Byzantine
 748 failure. We have two cases to consider. If both b and b' are inconsistent because Condition 1
 749 holds, then the intersecting voters are Byzantine as correct processes vote only once per epoch.
 750 Hence if process q votes for b and b' then q is Byzantine. If both b and b' are inconsistent
 751 because Condition 2 is met, then the intersecting voters are Byzantine, as correct processes
 752 vote only for blocks extending one of the longest notarized chains. That is, if correct process
 753 p votes for b it means that b is extending a notarized block b_{pred} that is of height $b.height - 1$,
 754 therefore p cannot vote afterwards for a block b' whose height is strictly smaller than $b.height$
 755 because p must extend one of the longest notarized chain. It follows that if process q votes
 756 for both b and b' then q is Byzantine.

757 Let us now show that a fork occurs because of two inconsistent blocks. If there is a
 758 fork then this gives rise to two sequences of three adjacent blocks with consecutive epochs,
 759 b_1, b_2, b_3 and b'_1, b'_2, b'_3 (by construction given the finalization rule). If no blocks share the
 760 same epoch number then we can assume w.l.o.g. that $b_3.epoch < b'_1.epoch$. Let block b'
 761 belonging to the prefix of b'_3 such that $b'.epoch > b_1.epoch$ and $b'.height$ is the smallest in the
 762 prefix of b'_3 . Such block always exists as b'_1 satisfies those two conditions. We have two cases:
 763 Either $b'.height < b_3.height$ or $b'.height \geq b_3.height$. In the former case, b' is inconsistent
 764 with b_3 since by assumption $b'.epoch > b_3.epoch$. In the latter case, the predecessor of b'



■ **Figure 3** 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.

765 is inconsistent with b_3 . Indeed, the predecessor of b' has a strictly smaller height than b_1
 766 and by assumption has a larger epoch number than b_3 . Figure 3 illustrates the presence
 767 of inconsistent blocks in presence of a fork at some block b_c . From b_c two chains are built,
 768 the first one consisting of the sequence of three blocks b_1, b_2 and b_3 , and the second chain
 769 consisting of four consecutive blocks b_d, b'_1, b'_2, b'_3 (to illustrate the first case) and of five
 770 consecutive blocks $b_d, b_e, b'_1, b'_2, b'_3$ (to illustrate the second case). In both cases block b'_1 plays
 771 the role of block b' . In the first case (figure in the top), $b_3.height = 6$ and $b'.height = 5$ while
 772 $b_3.epoch = 6$ and $b'.height = 5$. Thus Condition 2 applies. In the second case (figure in the
 773 bottom), since $b'.height \geq b_3.height$ then there must exist some block b_e in the b' prefix.
 774 Thus $b_e.height < b'.height$. Given that by assumption $b_e.epoch > b_3.epoch$, then Condition 2
 775 holds for b_e and b_3 . Hence there is always a couple of inconsistent blocks in a fork.

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

783 Finally, by applying Corollary 15, we complete the proof of the theorem.