Polygraph

Accountable Byzantine Agreement

Pierre Civit
University of Sydney
Sydney, Australia
pierrecivit@gmail.com

Seth Gilbert
NUS
Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

Vincent Gramoli
University of Sydney
Sydney, Australia
vincent.gramoli@sydney.edu.au

ABSTRACT

In this paper, we introduce *Polygraph*, the first accountable Byzantine consensus algorithm. If among n users $t < n/3$ are malicious then it ensures consensus; otherwise (if $t \geq n/3$), it eventually detects malicious users that cause disagreement. Polygraph is appealing for blockchain applications as it allows them to totally order blocks in a chain whenever possible, hence avoiding forks and double spending and, otherwise, to punish (e.g., via slashing) at least $n/3$ malicious users when a fork occurs. This problem is more difficult than perhaps it first appears. One could try identifying malicious senders by extending classic Byzantine consensus algorithms to piggyback signed messages. We show however that to achieve accountability the resulting algorithms would then need to exchange $\Omega(\kappa^2 \cdot n^5)$ bits, where κ is the security parameter of the signature scheme. By contrast, Polygraph has communication complexity $O(\kappa \cdot n^4)$. Finally, we implement Polygraph in a blockchain and compare it to the Red Belly Blockchain to show that it commits more than 10,000 Bitcoin-like transactions per second when deployed on 80 geodistributed machines.

1 INTRODUCTION

Over the last several years we have seen a boom in the development of new Byzantine agreement protocols, in large part driven by the excitement over blockchains and cryptocurrencies. Unfortunately, Byzantine agreement protocols have some inherent limitations: it is impossible to ensure correct operation when more than 1/3 of the processing power in the system is controlled by a single malicious party, unless the network can guarantee perfect synchrony in communication. At first, one might hope to relax the liveness guarantees, while always ensuring safety. Alas, in a partially synchronous network, this type of guarantee is impossible. If the adversary controls more than 1/3 of the computing power, it can always force disagreement.

Accountability. What if, instead of *preventing* bad behavior by a party that controls too much power, we guarantee accountability, i.e., we can provide irrefutable evidence of the bad behavior and the identifier of the perpetrator of those illegal actions? Much in the way we prevent crime in the real world, we can prevent bad blockchain behavior: if the attacker has strictly less than 1/3 of the network under their control then consensus is reached, otherwise we record sufficient information to catch the third of the network that is criminal and take remedial actions. Accountability has been increasingly discussed as a desirable property in blockchains to slash stake of cheating peers [9, 32]. The problem is to avoid suspecting correct peers while provably identifying cheating ones.

Why is it a hard problem? As far as we know, there is no generic way of getting definitive evidence of the guilt of processes (or nodes) for all systems. Previous work introduced a method to transform any distributed system into one that is accountable but it only guarantees that faulty processes will be suspected forever if the network is partially synchronous [18]. We thus narrow down the problem to *accountable Byzantine agreement*, specifically reaching agreement when there are fewer than $n/3$ Byzantine participants or detecting at least $n/3$ Byzantine participants in case of a disagreement. Most partially synchronous Byzantine consensus protocols, like PBFT [11], Tendermint [8] or HotStuff [35], already collect forms of cryptographic evidence like signatures or certificates to guarantee agreement upon a decision. So one might think of simply recording the quorum certificates containing honest processes signatures that attest the decision to detect $n/3$ Byzantine processes in case of disagreement. In fact, we show that justifications should contain at least $\Omega(\kappa \cdot n^2)$ bits (where κ is the security parameter of the signature scheme) for a simple piggybacking extension to make any of these algorithms accountable (see Theorem 4.3).

Results. In this paper, we propose *Polygraph*, the first accountable Byzantine agreement solution. The idea is to offer accountability guarantees to the participants of the service. Intuitively, one cannot hold n servers accountable to separate clients (distinct from the servers) that interact with the blockchain when more than $t \geq 2n/3$ of the servers are Byzantine. The reason is that the coalition is sufficiently large to rewrite the blockchain and prevent a client from distinguishing the response of honest servers from the response of malicious servers [13]. This was confirmed by concurrent

Algorithm	Msgs	Bits	Account.
PBFT [11]	$O(n^3)$	$O(\kappa \cdot n^4)$	✗
Tendermint [7]	$O(n^3)$	$O(\kappa \cdot n^3)$	✗
HotStuff [35]	$O(n^2)$	$O(\kappa \cdot n^2)$	✗
HotStuff w/o thres. sig. [21]	$O(n^2)$	$O(\kappa \cdot n^3)$	✗
DBFT binary consensus [14]	$O(n^3)$	$O(n^3)$	✗
DBFT multivalue consensus [14]	$O(n^4)$	$O(n^4)$	✗
Polygraph (Sect. 5)	$O(n^3)$	$O(\kappa \cdot n^4)$	✓
Naive Multiv. Polygraph (App. C)	$O(n^4)$	$O(\kappa \cdot n^5)$	✓
Multivalue Polygraph (App. D)	$O(n^4)$	$O(\kappa \cdot n^4)$	✓

Table 1: Differences in communication complexities after global stabilization time between Polygraph and non-accountable Byzantine consensus algorithms, where n is the number of consensus participants and κ is the security parameter of the corresponding encryption scheme.

research to ours that showed it is impossible to hold servers accountable to separate clients for any number of Byzantine participants [33].

Our solution, called *Polygraph*, ensures that in a symmetric system where all n participants are peers that take part as clients and servers in the accountable Byzantine consensus, then accountability is ensured for **any number** $t \leq n$ **of Byzantine participants**. Note that the problem is trivial when $t > n - 2$ as no disagreement among correct processes is possible, but otherwise Polygraph guarantees all honest participants undeniably detect at least $n/3$ Byzantine participants responsible for disagreement. Because it is resilient to any number of failures, Polygraph is particularly interesting for peer-to-peer blockchain networks. In particular, it allows to hold all peers accountable to other peers, which is appealing for consortium blockchains and shard chains [17].

We also show that Polygraph is optimal in that stronger forms of accountability are impossible. For example, we cannot guarantee agreement when $t > n/3$, even if we are willing to tolerate a failure of liveness (Theorem 4.1); and processes cannot detect even one guilty participant within a fixed time limit (e.g., prior to decision), since (intuitively) that would enable processes to determine guilt before deciding in a way that leads to disagreement. Nor can we guarantee detection of more than $n/3$ malicious users, since it takes only $n/3$ malicious users to cause disagreement and additional malicious users could simply stay mute to not be detected.

Finally, we show that Polygraph is efficient. First, its communication complexity is $O(\kappa \cdot n^4)$ bits, where n is the number of participants and κ is the security parameter of its signature scheme. This complexity is comparable to the communication complexity of state-of-the-art consensus algorithms as depicted in Table 1 because Polygraph simply needs to exchange signed messages received within at most the two latest previous asynchronous rounds. In particular, both the binary and the multivalue versions of Polygraph share the same asymptotic complexity as PBFT, which does not offer accountability. Second, we evaluate the performance of Polygraph in a blockchain application that thus becomes accountable. We deploy this blockchain application on 80 machines across continents and compare its performance to the Red Belly Blockchain [15]. Even though it presents some overheads compared to this non-accountable baseline, our accountable blockchain still exceeds 10,000 TPS at 80 nodes. This high performance can be attributed to the reasonable complexity of the multivalue variant of Polygraph depicted in Table 1.

Roadmap. The background is given in Section 2. The model and the accountable Byzantine consensus problem are presented in Section 3, and impossibility results are given in Section 4. Section 5 describes the Polygraph protocol, which solves the accountable binary Byzantine consensus problem. Section 6 analyses empirically the Polygraph protocol in a geodistributed blockchain. blockchain and Section 7 concludes.

An Appendix is left to the discretion of the reader. Appendix A presents the proof of the impossibility result, Appendix B presents the proof of correctness of the Polygraph protocol, Appendix C presents the multivalue Polygraph protocol that supports arbitrary values. Appendix D presents some optimizations to reduce the complexity of the multivalue Polygraph protocol. Appendix E discusses the applications of the Polygraph protocol to blockchain. Finally, Appendix F shows that a naive extension of classic blockchain consensus protocols, including PBFT, HotStuff and Tendermint, cannot make them accountable.

2 BACKGROUND AND RELATED WORK

In this section, we review existing work on accountability in distributed systems.

2.1 PeerReview

Haeberlen, Kuznetsov, and Druschel [18] pioneered the idea of accountability in distributed systems. They developed a system called *PeerReview* that implemented accountability as an add-on feature for any distributed system. Each process in the system records messages in tamper-evident logs; an authenticator can challenge a process, retrieve its logs, and simulate the original protocol to ensure that the process behaved correctly. They show that in doing so, you can always identify at least one malicious process (if some process acts in a detectably malicious way). Their technique is quite powerful, given its general applicability which can be used in any (deterministic) distributed system!

The issue has to do with (partial) synchrony. The PeerReview approach is challenge-based: to prove misbehavior, an auditor must receive a response from the malicious process. If no response is received, the auditor cannot determine whether the process is malicious, or whether the network has not yet stabilized. It follows that the malicious coalition will only be suspected forever but not proved guilty. There is no fixed point at which the auditor can be *completely certain* that the sender is malicious; the auditor may never have definitive *proof* that the process is malicious; it always might just be poor network performance. The Polygraph Protocol, by contrast, produces a concrete proof of malicious behavior that is completely under the control of the honest processes.

2.2 Accountable blockchains

Recently, accountability has been an important goal in "proof-of-stake" blockchains, where users that violate the protocol can be punished by confiscating their deposited stake.

Buterin and Griffith [9] have proposed a blockchain protocol, Casper, that provides this type of accountability guarantee. Validators try to agree on (or "finalize") a branch of k hundreds of consecutive blocks, by gathering signatures for this branch or "link" from validators jointly owning at least $2n/3$ of the deposited stake. If a validator signs multiple links at the same height, Casper uses its signatures as proofs to slash its deposited stake. This is very similar in intent to Polygraph's notion of identifying $n/3$ malicious users when there is disagreement. Like most blockchain protocols, however, Casper implicitly assumes some synchronous underlying (overlay) network and allows the blockchain to fork into a tree until some branch is finalized. To guarantee "plausible liveness" or that Casper does not block when not enough signatures are collected to finalize a link, validators are always allowed to sign links that overlap but extend links they already signed. However, this does not guarantee that consensus terminates.

The longlasting blockchain [32] builds upon our companion technical report [13] to recover from forks by excluding guilty participants and compensating transient losses with the deposit of guilty participants. It recovers from $f = \lceil 2n/3 \rceil - 1$ failures as long as there are less than $\min(n/3, n - f)$ processes experiencing benign (e.g., crash) failures. Accountability in the context of blockchain fairness was raised by Herlihy and Moir in a keynote address [20], and the idea of "accountable Byzantine fault tolerance" has been discussed [7]. The goal in the latter case is to suggest a broadcast after the consensus in order to detect a fault by matching pre-vote and pre-commit messages of the same validator in Tendermint, but the algorithm is not detailed. Holding n servers accountable to separate clients in the Tendermint consensus algorithm [4, 8] as well as HotStuff [35] was shown possible when $t < 2n/3$ in recent research [33] but could not be achieved when $t \geq 2n/3$, which confirms our observations (Appendix E).

2.3 Earlier work on accountability

Even before PeerReview, others had suggested the idea of accountability in distributed systems as an alternate approach to security (see, e.g., [26, 36, 37]). Yumerefendi and Chase [37] developed an accountable system for network storage, and Michalakis et al. [30] developed an accountable peer-to-peer content distribution network. The idea of accountability appeared less explicit in many earlier systems. For example, Aiyer et al. [3] proposed the BAR model for distributed systems, which relied on incentives to ensure good behavior; one key idea was in detecting and punishing bad behaviors. Finally, Intrusion Detection Systems (e.g., [16, 22, 28] provided heuristics and techniques for detecting malicious behaviors in a variety of different systems.

2.4 Failure detectors

There is a connection between accountability and failure detectors. A failure detector is designed to provide each process in the system with some advice, typically a list of processes that are faulty in some manner. However, failure detectors tend to have a different set of goals. They are used during an execution to help make progress, while accountability is usually about what can be determined *post hoc* after a problem occurs. They provide advice to a process, rather than proofs of culpability that can be shared. Most of the work in this area has focused on detecting crash failures (see, e.g., [12]). There has been some interesting work extending this idea to detecting Byzantine failures [18, 19, 23, 28]. Malkhi and Reiter [28] introduced the concept of an unreliable Byzantine failure detector that could detect *quiet* processes, i.e., those that did not send a message when they were supposed to. They showed that this was sufficient to solve Byzantine Agreement.

Kihlstrom, Moser, and Melliar-Smith [23] continue this direction, considering failures of both omission and commission. Of note, they define the idea of a *mutant message*, i.e., a message that was received by multiple processes and claimed to be identical (e.g., had the same header), but in fact was not. The Polygraph Protocol is designed so that only malicious users sending a mutant message can cause disagreement. In fact, the main task of accountability in this paper is identifying processes that were supposed to broadcast a single message to everyone and instead sent different messages to different processes.

Maziéres and Shasha propose SUNDR [29] that detects Byzantine behaviors in a network file system if all clients are honest and can communicate directly. Polygraph clients request multiple signatures from servers so that they do not need to be honest. Li and Maziéres [27] improves on SUNDR with BFT2F, a weakly consistent protocol when the number of failures is $n/3 \leq t < 2n/3$ and its BFTx variant that copes with more than $2n/3$ failures but does not guarantee liveness even with less than t failures.

3 MODEL AND PROBLEM

We first define the problem in the context of a traditional distributed computing setting. (We later discuss applications to blockchains.)

System. We consider n processes. A subset C of the processes are honest, i.e., always follow the protocol; the remaining $t < n$ are Byzantine, i.e., may maliciously violate the protocol, under the control of a dynamic adversary that fixes the set of Byzantine processes for the duration of each round. We define $t_0 = \max(t \in \mathbb{N}_0 : t < n/3)$, i.e., $t_0 = \lceil \frac{n}{3} \rceil - 1$, a useful threshold on the number of Byzantine behaviors.

Processes execute one step at a time and are asynchronous, proceeding at their own arbitrary, unknown speed. We assume local computation time is zero, as it is negligible with respect to message delays.

We assume that there is an idealized PKI (public-key infrastructure) so that each process has a public/private key pair that it can use to sign messages and to verify signatures.

Partial synchrony. We consider a partially synchronous network. During some intervals of time, messages are delivered in a reliable and timely fashion, while in other intervals of time messages may be arbitrarily delayed. More specifically, we assume that there is some time τ_{GST} known as the *global stabilization time*, unknown to the processes, such that any message sent after time τ_{GST} will be delivered with latency at most d. We say that an event occurs *eventually* if there exists an unknown but finite time when the event occurs. (Note that we tolerate that messages be dropped before τ_{GST} as long as messages are sent infinitely often.) For the sake of simplicity in the presentation, we write "receive k messages" to explain "receive messages from k distinct processes".

Verification algorithm. A verification algorithm V takes as input the state of a process and returns a set G of undeniable guilty processes, that is, every process-id of G is tagged with an unforgeable proof of culpability. (More formally, this means that for every computationally bounded adversary, for every execution in which a process p_j is honest, for every state s generated during the execution or constructed by Byzantine users, the probability that the verification algorithm returns a set containing p_j is negligible. In practice, this will reduce to the non-forgeability of signatures.)

Accountable Byzantine agreement. The problem of Byzantine Agreement, first introduced by Pease, Shostak, and Lamport [25], assumes that each process begins with a binary *input*, i.e., either a 0 or a 1, outputs a *decision*, and requires three properties: agreement, validity, and termination.

We define the Accountable Byzantine Agreement problem in a similar way, with the additional requirement that there exists a verification algorithm that can identify at least $t_0 + 1$ Byzantine users whenever there is disagreement. (Recall that $t_0 = \lceil \frac{n}{3} \rceil - 1$.) More precisely:

Definition 3.1 (Accountable Byzantine Agreement). We say that an algorithm solves Accountable Byzantine Agreement if each process takes an input value, possibly produces a decision, and satisfies the following properties:

- *Agreement:* If $t \leq t_0$, then every honest process that decides outputs the same decision value.
- *Validity:* If all processes are honest and begin with the same value, then that is the only decision value.
- *Termination:* If $t \leq t_0$, every honest process eventually outputs a decision value.
- *Accountability*: There exists a verification algorithm V such that: if two honest processes output disagreeing decision values, then eventually for every honest process p_j, for every state s_j reached by p_j from that point onwards, the verification $V(s_j)$ outputs a guilty set of size at least $t_0 + 1$.

Our validity definition is sometimes called weak validity [31], but Lemma B.8 shows that our accountable binary Byzantine consensus protocol ensures even a stronger validity property (if $t \leq t_0$ and an honest process decides v, then some honest process proposed v).

4 IMPOSSIBILITY RESULTS

It may seem that accountability can be obtained by always guaranteeing agreement but failing to terminate as soon as $t \geq n/3$, by checking evidence before deciding, or by piggybacking a subquadratic number of bits as justifications in classic consensus algorithms. In this section, we show that none of these ideas lead to accountability.

4.1 Avoiding disagreement when $t \geq n/3$ is impossible

A couple of natural questions arise regarding accountable algorithms: Can we design an algorithm that always guarantees agreement, and simply fails to terminate if there are too many Byzantine users? If so, we would trivially get accountability! Can we design an algorithm that provides earlier evidence of Byzantine behavior, even before the decision is possible? If so, we could provide stronger guarantees than are provided in this paper. Alas, neither is possible. The details of the following theorems are deferred to Appendix A (and follow from standard partitioning arguments).

Theorem 4.1. *In a partially synchronous system, no algorithm solves both the Byzantine consensus problem when $t < n/3$ and the agreement and validity of the Byzantine consensus problem when $t > t_0$.*

We say that a verification algorithm V is *swift* if it guarantee the following: assume p_i has already decided some value v, and that p_j is in a state s wherein it will decide $w \neq v$ in its next step; then $V(s) \neq \emptyset$. Notice that a swift verification algorithm may only detect *one* Byzantine process (i.e., it is not sufficient evidence for p_j to decide never to decide).

Theorem 4.2. *Consider an algorithm that solves consensus when $t < n/3$. There is no swift verification algorithm when $t > t_0$.*

4.2 Classic PBFT-like algorithms

It is interesting to see that most partially synchronous consensus algorithms already collect forms of cryptographic evidence like signatures or certificates to guarantee agreement upon decision. While this is a characteristic of PBFT [11], this is the case of modern algorithms that build upon it including Tendermint [8] and HotStuff [35]. One could naturally be tempted to reuse these signatures and certificates to piggyback practical justifications in the existing messages of the original consensus algorithms to turn them into accountable consensus algorithms. Although threshold signatures do not convey enough information to identify which process is guilty, signatures are generally sufficient. We show below that, unfortunately, this transformation cannot work.

The intuition of the proof is split in the four following steps while the full proof is deferred to our technical report. First, we define the class of 'classic' or PBFT-like consensus algorithms and denote it $\mathcal{L}$. Second, we show that HotStuff [35], PBFT [11] and Tendermint [4] belong to this class $\mathcal{L}$. (As we need to remove threshold signatures, we adopt the version of HotStuff without threshold signatures [21].) Third, we define a practical extension of PBFT-like consensus algorithms that piggybacks messages. By 'practical' we mean that piggybacked messages have a bounded staleness to prevent the justification communication to be superlinear (e.g., quadratic) in n. Finally, we prove that there exist executions leading to disagreement with different sets of Byzantine processes that correct processes cannot distinguish.

(1) Class $\mathcal{L}$ of PBFT-like algorithms. $\mathcal{L}$ contains algorithms that all rely on a leader, which rotates in a round-robin fashion across views and proposes a *suggestion* in its view. Two local variables per process, *preparation* and *decision*, have values that relate with each other such that for a process j to have a *decision*, $n - t_0$ processes, including j itself, must have had the same value as a *preparation*. During view changes, if $t < t_0$, then a *preparation* implies a propagation of a value to the leader within messages announcing the new view.

(2) HotStuff, PBFT and Tendermint belong to $\mathcal{L}$. By examination of the code of HotStuff, PBFT and Tendermint, we can see that they all belong to $\mathcal{L}$. HotStuff's leader, PBFT's primary and Tendermint's proposer of an epoch all aim at proposing a *suggestion* within the view they coordinate. A *decision* value always requires the same *preparation* value from $n - t_0$ distinct processes. These two values correspond to HotStuff's commitQC and prepareQC, to Tendermint's decision and validValue, and to PBFT's commit and prepare. In all these three consensus algorithms, a *suggestion* value in view $v + 1$ must be proposed by its leader and correspond to a preparation motivated by $n - t_0$ messages sent in view v.

(3) Extensions of a PBFT-like algorithm. The *t_0-bounded extension* of a PBFT-like consensus algorithm A is an algorithm $\bar{A}$ like A except that it piggybacks a justification within all its new-view and suggestion messages at each view $v_k = k$. This justification consists of a chain of the past alternating sets $Sugg^x$ of suggestion messages sent by the leader ℓ_{v_x} of view $v_x = x$ and sets NV^x of new-view messages sent by processes ϕ^x for view v_x. Each piggybacked chain sent in view v_k has a bounded depth $t_0 - 1 = \Theta(n)$, which means that it contains a (possibly empty) suffix of this sequence of sets: $NV^{k-(t_0-1)}, Sugg^{k-(t_0-2)}, NV^{k-(t_0-2)}, \cdots, Sugg^{k-1}, NV^{k-1}$.

Theorem 4.3. *HotStuff, PBFT and Tendermint as well as all their t_0-bounded extensions are not accountable.*

(4) Intuition of the proof. For the proof we consider two executions e^1 and e^2, in which process i decides s_i at view v_i while process j decides $s_j \neq s_i$ at view $v_j >> v_i$. Process i's decision implies preparation of s_i by a quorum Q_i at view v_i. In a later view numbered v_z, a set ϕ of processes send a set $\underline{NV}^{v_z-1}$ of new-view messages to the leader ℓ_{v_z} of this view. A set $B = \phi \cap Q_i$ of at least $t_0 + 1$ guilty processes did not propagate their preparation of s_i and provoked the disagreement. The view v_z is the first link of a chain of successive views $\chi = [v_z, v_z + 1, \ldots, v_z + k - 1]$ where the leaders of views χ are in P, $|P| \leq t_0 - 1$. At view $v_z + k$, process j prepares s_j and eventually decides s_j at view $v_j \geq v_z + k$. When i and j detect the disagreement, they can neither distinguish e^1 from e^2 nor identify the senders ϕ of $\underline{NV}^{v_z-1}$. Process j cannot wait without deciding because we can construct an execution e^0 indistinguishable by j from e^1 with less than t_0 Byzantine processes, where the leaders P (and i) of the chain χ appear mute to j and where j must decide. Leaders of P prepare s_j as j ignores the decision s_i. After the disagreement, P does not reveal $\underline{NV}^{v_z-1}$ that is necessary to detect the guilty processes. This argument holds as long as $k < t_0$. The full proof is deferred to Appendix F.

This result (Theorem 4.3) simply shows that piggybacking t_0-bounded justifications is insufficient to make PBFT-like algorithms accountable, however, it does not mean that they cannot be transformed into an accountable algorithm. First, one could probably make PBFT-like algorithms accountable with a longer justification, exchanging $\Omega(\kappa \cdot n^2)$ times more bits, where κ is the security parameter of the signature scheme. This new extension would result in an accountable version of Tendermint, HotStuff and PBFT requiring between $\Omega(\kappa^2 \cdot n^5)$ and $\Omega(\kappa^2 \cdot n^6)$ bits. Second, transforming any of these algorithms into Polygraph (Section 5) is a way of obtaining accountability with a lower complexity than the previous extension. Such a transformation would however be non-trivial because Polygraph relies on DBFT that differs from PBFT-like algorithms in various ways: every process participating in DBFT can propose a value, DBFT is signature-free and there is no view change in DBFT as there is no need to recover from a failed leader.

5 POLYGRAPH, AN ACCOUNTABLE BYZANTINE CONSENSUS ALGORITHM

In this section, we introduce *Polygraph*, a Byzantine agreement protocol that is accountable. We begin by giving the basic outline of the protocol for ensuring agreement when $t < n/3$. The protocol is derived from the DBFT consensus algorithm [14] that was proved correct using the ByMC model checker [34] and that does not use the leader-based pattern mentioned in the proof of Theorem 4.3. Then, we focus on the key aspects that lead to accountability, specifically, the "ledgers" and "certificates." For the sake of simplicity, this section tackles the binary agreement, however, Appendix C generalizes this result to arbitrary values. In Appendix B, we prove that the algorithm is correct.

As a notation, we indicate that a process p_i sends a message to every other process by: broadcast(TAG, m) $\longrightarrow$ *messages*, where *TAG* is the type of the message, m is the message content, and *messages* is the location to store any messages received.

Throughout we assume that every message is signed by the sender so the receiver can authenticate who sent it. (Any improperly signed message is discarded.) Thus we can identify messages sent by distinct processes. Similarly, the protocol will at times include cryptographically signed "ledgers" in messages; again, any message that is missing a required ledger or has an improperly formed ledger is discarded. (See the discussion below regarding ledgers.)

5.1 Protocol overview

The basic protocol operates in two phases, after which a possible decision is taken. Each process maintains an estimate. In the first phase, each process broadcasts its estimate using a reliable broadcast service, bv-broadcast (discussed below), as introduced previously [1]. The protocol uses a rotating coordinator; whoever is the assigned coordinator for a round broadcasts its estimate with a special designation.

All processes then wait until they receive at least one message, and until a timer expires. (The timeout is increased with each iteration, so that eventually once the network stabilizes it is long enough.) If a process receives a message from the coordinator, then it chooses the coordinator's value to "echo", i.e., to rebroadcast to everyone in the second phase. Otherwise, it simply echoes all the messages received in the first phase.

At this point, each process p_i waits until it receives enough *compatible* ECHO messages. Specifically, it waits to receive at least $(n - t_0)$ messages sent by distinct processes where every value in those messages was also received by p_i in the first phase. In this case, it adopts the collection of values in those $(n - t_0)$ messages as its candidate set. In fact, if a process p_i receives a set of $(n - t_0)$ messages that *all* contain exactly the coordinator's value, then it chooses only that value as the candidate value.

Finally, the processes try to come to a decision. If process p_i has only one candidate value v, then p_i adopts that value v as its estimate. In that case, it can decide v if it matches the parity of the round, i.e., if $v = r_i \mod 2$. Otherwise, if p_i has more than one candidate value, then it adopts as its estimate $r_i \mod 2$, the parity of the round.

To see that this ensures agreement (when $t < n/3$), consider a round in which some process p_i decides value $v = r_i \mod 2$. Since p_i receives $(n - t_0)$ echo messages containing *only* the value v, we know that every honest process must have value v in every possible set of $(n - t_0)$ echo messages, and hence every honest process included v in its candidate set. Every honest process that *only* had v as a candidate also decided v. The remaining honest processes must have adopted $v = r_i \mod 2$ as their estimate when they adopted the parity bit of the round. And if all the honest processes begin a round r with estimate v, then that is the only possible decision due to the reliable broadcast bv-broadcast in Phase 1 (see below) and all honest processes decide at round $r + 2$ or earlier (regardless of whether τ_{GST} is reached).

Processes always continue to make progress, if $t < n/3$. Termination is a consequence of the coordinator: eventually, after GST when the network stabilizes, there is a round where

Algorithm 1 The Polygraph Protocol

```
bin-propose($v_i$):
    $est_i = v_i$
    $r_i = 0$
    $timeout_i = 0$
    $ledger_i[0] = \emptyset$
    repeat:
        $r_i \leftarrow r_i + 1$;                                   ▷ increment the round number and the timeout
        $timeout_i \leftarrow timeout_i + 1$
        $coord_i \leftarrow ((r_i - 1) \mod n) + 1$                 ▷ rotate the coordinator
    ▹ Phase 1:
        bv-broadcast(EST$[r_i]$, $est_i$, $ledger_i[r_i - 1]$, $i$, $bin\_values_i[r_i]$)   ▷ binary value broadcast the current estimate
        if $i = coord_i$ then                                       ▷ coordinator rebroadcasts first value received
            wait until ($bin\_values_i[r_i] = \{w\}$)               ▷ bin_values stores messages received by binary value broadcast
            broadcast(COORD$[r_i]$, $w$) $\longrightarrow messages_i$
        StartTimer($timeout_i$)                                     ▷ reset the timer
        wait until ($bin\_values_i[r_i] \neq \emptyset \wedge timer_i$ expired)
    ▹ Phase 2:
        if (COORD$[r_i]$, $w$) $\in messages_i$ from $p_{coord_i} \wedge w \in bin\_values_i[r_i]$) then   ▷ favor the coordinator
            $aux_i[r_i] \leftarrow \{w\}$
        else $aux_i[r_i] \leftarrow bin\_values_i[r_i]$             ▷ otherwise, use any value received
        $signature_i$ = sign($aux_i[r_i]$, $r_i$, $i$)              ▷ sign the messages
        broadcast(ECHO$[r_i]$, $aux_i[r_i]$, $signature_i$) $\longrightarrow messages_i$   ▷ broadcast second phase message
        wait until $values_i$ = ComputeValues($messages_i$, $bin\_values_i[r_i]$, $aux_i[r_i]$) $\neq \emptyset$
    ▹ Decision phase:
        if $values_i = \{v\}$ then                                  ▷ if there is only one value, then adopt it
            $est_i \leftarrow v$
            if $v = (r_i \mod 2)$ then                              ▷ decide if value matches parity
                if no previous decision by $p_i$ then decide($v$)
        else
            $est_i \leftarrow (r_i \mod 2)$                         ▷ otherwise, adopt the current parity bit
        $ledger_i[r_i]$ = ComputeJustification($values_i$, $est_i$, $r_i$, $bin\_values_i[r_i]$, $messages_i$)   ▷ broadcast certificate
```

Rules:

(1) Every message that is not properly signed by the sender is discarded.
(2) Every message that is sent by bv-broadcast without a valid ledger after Round 1, except for messages containing value 1 in Round 2, are discarded.
(3) On first discovering a ledger ℓ that conflicts with a certificate, send ledger ℓ to all processes.

the coordinator is honest and the timeout is larger than the message delay. At this point, every honest process receives the coordinator's Phase 1 message and echoes the coordinator's value. In that round, every honest process adopts the coordinator's estimate, and the decision follows either in that round or the next one (if $t < n/3$).

5.2 Binary value broadcast

The protocol relies in Phase 1 on a reliable broadcast routine bv-broadcast proposed before [1], which is used to ensure validity, i.e., any estimate adopted (and later decided) must have been proposed by some honest process. Moreover, it guarantees that if every honest process begins a round with the same value, then that is the only possible estimate for the remainder of the execution (if $t < n/3$). Specifically, bv-broadcast guarantees the following critical properties while $t < n/3$: (i) every message broadcast by $t_0 + 1$ honest processes is eventually delivered to every honest process (see Lemma B.2); (ii) every message delivered to an honest process was broadcast by at least $t + 1$ processes (see Lemma B.1).

These properties are ensured by a simple echo procedure. When a process first tries to bv-broadcast a message, it broadcasts it to everyone. When a process receives $t_0 + 1$ copies of a message, then it echoes it. When a process receives $n - t_0$ copies of a message, then it delivers it. Notice that if a message is not bv-broadcast by at least $t_0 + 1$ processes, then it is never echoed and hence never delivered. And if a message is bv-broadcast by $t_0 + 1$ (honest) processes is echoed by every honest process and hence delivered to every honest process.

This reliable broadcast routine ensures validity, since a Phase 1 message that is echoed in Phase 2 must have been delivered by bv-broadcast, and hence must have been bv-broadcast by at least one honest process.

Algorithm 2 Helper Components

```
1:  bv-broadcast(MSG, $val$, $ledger$, $i$, $bin\_values$):
2:    broadcast(BVAL, $\langle val, ledger, i\rangle$) $\rightarrow$ $msgs$                ▷ broadcast message
3:    After round 2, and in round 1 if $val = 0$, discard all messages received without a proper ledger.
4:    upon receipt of (BVAL, $\langle v, \cdot, j\rangle$)
5:      if (BVAL, $\langle v, \cdot, \cdot\rangle$) received from $(t_0 + 1)$ distinct processes and (BVAL, $\langle v, \cdot, \cdot\rangle$) not yet broadcast then
6:        Let $\ell$ be any non-empty ledger received in these messages.      ▷ one of the received ledgers is enough
7:        broadcast(BVAL, $\langle v, \ell, j\rangle$)                       ▷ Echo after receiving $(t_0 + 1)$ copies.
8:      if (BVAL, $\langle v, \cdot, \cdot\rangle$) received from $(2t_0 + 1)$ distinct processes then
9:        Let $\ell$ be any non-empty ledger received in these messages.      ▷ one of the received ledgers is enough
10:       $bin\_values \leftarrow bin\_values \cup \{\langle v, \ell, j\rangle\}$     ▷ deliver after receiving $(2t_0 + 1)$ copies

11: ComputeValues($messages$, $b\_set$, $aux\_set$):                   ▷ check if there are $n - t_0$ compatible messages
12:   if $\exists S \subseteq messages$ where the following conditions hold:
13:     (i) $S$ contains $(n - t_0)$ distinct ECHO$[r_i]$ messages
14:     (ii) $aux\_set$ is equal to the set of values in $S$.
15:     then return($aux\_set$)
16:   if $\exists S \subseteq messages$ where the following conditions hold:
17:     (i) $S$ contains $(n - t_0)$ distinct ECHO$[r_i]$ messages
18:     (ii) Every value in $S$ is in $b\_set$.
19:     then return($V$ = the set of values in $S$)
20:   else return($\emptyset$)

21: ComputeJustification($values_i$, $est_i$, $r_i$, $bin\_values_i$, $messages_i$):   ▷ compute ledger and broadcast certificate
22:   if $est_i = (r_i \mod 2)$ then
23:     if $r_i > 1$ then
24:       $ledger_i[r_i] \leftarrow$ ledger $\ell$ where (EST$[r_i]$, $\langle v, \ell, \cdot\rangle$) $\in bin\_values_i$
25:     else $ledger_i[r_i] \leftarrow \emptyset$
26:   else $ledger_i[r_i] \leftarrow (n - t_0)$ signed messages from $messages_i$ containing only value $est_i$
27:   if $values_i = \{(r_i \mod 2)\} \wedge$ no previous decision by $p_i$ in previous round then
28:     $certificate_i \leftarrow (n - t_0)$ signed messages from $messages_i$ containing only value $est_i$
29:     broadcast($est_i$, $r_i$, $i$, $certificate_i$)                    ▷ transmit certificate to everyone
30:   return $ledger_i[r_i]$
```

5.3 Ledgers and certificates

In order to ensure accountability, we need to record enough information during the execution to justify any decision that is made, and hence to allow processes to determine accountability. For this purpose, we define two types of justifications: ledgers and certificates. A ledger is designed to justify adopting a specific value. A certificate justifies a decision. We will attach ledgers to certain messages; any message containing an invalid or malformed ledger is discarded.

We define a ledger for round r and value v as follows. If $v \neq r \bmod 2$, then the ledger consists of the $(n-t_0)$ ECHO messages, each properly signed, received in Phase 2 of round r that contain only value v (and no other value). If $v = r \bmod 2$, then the ledger is simply a copy of *any* other ledger from the previous round $r - 1$ justifying value v. (The asymmetry may seem strange, but is useful in finding the guilty parties!)

We define a certificate for a decision of value v in round r to consist of $(n - t_0)$ echo messages, each properly signed, received in Phase 2 of round r that contain only value v (and no other value).

5.4 Accountability

We now explain how the ledgers and certificates are used. In every round, when a process uses bv-broadcast to send a message containing a value, it attaches a ledger from the previous round justifying why that value was adopted. (There is one exception: in Round 1, no ledger will be available to justify value 1, so no ledger is generated in that case.)

The bv-broadcast ignores the ledger for the purpose of deciding when to echo a message. When it echoes a message m, it chooses any arbitrary non-empty ledger that was attached to a message containing m (if any such ledgers are available). However, every message that does not contain a valid ledger justifying its value is discarded, with the following exception: in Round 2, messages containing the value 1 can be delivered without a ledger (since no justification is available for adopting the value 1 in Round 1).

Whenever there is only one candidate value received in Phase 2, a process adopts that value and either: (i) decides and constructs a certificate, or (ii) does not decide and constructs a ledger. In both cases, this construction simply relies on the signed messages received in Phase 2 of that round (and hence is always feasible).

If a process decides a value v in round $r > 1$, or adopts v because it is the parity bit for round $r > 1$, then it also constructs a ledger justifying why it adopted that value v. It accomplishes this by examining all the bv-broadcast messages received for value v and copying a round $r - 1$ ledger. Again, this is always possible since any message that is not accompanied by a valid ledger is ignored. (The only possible problem occurs in Round 2 where messages for value 1 are not accompanied by a ledger; however ledgers for value 1 in round 2 do not require copying old ledgers.)

5.5 Proving culpability

How do disagreeing processes decide which processes were malicious? When a process decides in round r, it sends its certificate to all the other processes. Any process that decides a different value in a round $> r$ can prove the culpability of at least $\lceil n/3 \rceil$ Byzantine processes by comparing this certificate to its logged ledgers. (It can then broadcast the proper logged ledgers to ensure that everyone can identify the malicious processes.)

We will say that a certificate (e.g., from p_1) and a ledger (e.g., from p_2) *conflict* if they are constructed in the same round r, but for different values v and w. That is, both the certificate and the ledger attest to $(n - t_0)$ ECHO messages from round r sent to p_1 and p_2 (respectively) that contain only value v and only value w, respectively. Since every two sets of size $(n - t_0)$ intersect in at least $(n - 2 \cdot t_0)$ locations, fixing $t_0 = \lceil n/3 \rceil - 1$ helps identify at least $(t_0 + 1)$ processes that sent different Phase 2 messages in round r to p_1 and p_2 and hence they are malicious.

We now discuss how to find conflicting certificates and ledgers. Assume that process p_i decides value v in round r, and that process p_j decides a different value w in a round $> r$. (Recall that v is the only possible value that can be decided in round r.) There are two cases to consider, depending on whether p_j decides in round $r + 1$ or later.

- *Round* r+1*:* If p_j decides in round r+1, then value w was the only candidate value after Phase 2. This implies that w was received by some bv-broadcast message. Since $r > 1$, we know that the message must have contained a valid ledger ℓ from round r for value $w \neq v$. This ledger ℓ conflicts with the decision certificate of p_i.
- *Round* $\geq r + 2$*:* Since p_j decides $w \neq v$, it does not decide v in round $r + 2$. This means that p_j has w as a candidate value, which implies that p_j received w in a bv-broadcast. Since $r > 1$, we know that the message must have contained a valid ledger ℓ from round $r + 1$ for value $w \neq v$. This ledger ℓ consists of a copy of a ledger from round r for value w which conflicts with the decision certificate of p_i.

In either case, if p_j does not decide v, then, by looking at the messages received in round $r + 1$ and $r + 2$, it can identify a ledger that conflicts with the decision certificate of p_i and hence can prove the culpability of at least $t_0 + 1$ malicious processes.

5.6 Analysis of the Polygraph Protocol

In Appendix B.1, we show that the the BV-broadcast routine provides the requisite properties. This then allows us to prove the main correctness theorem, which follows immediately from Lemma B.7, Corollary B.9, and Lemma B.10 in Appendix B.2:

THEOREM 5.1. *The Polygraph Protocol is a correct Byzantine consensus protocol guaranteeing agreement, validity, and termination.*

Accountability follows from Lemma B.11, which shows that disagreement leads every honest process to eventually receive a certificate and a ledger that conflict:

THEOREM 5.2. *The Polygraph Protocol is accountable.*

If all the processes are honest, then the protocol terminates in $O(1)$ rounds after GST. Otherwise, it may take $t + 1$ rounds after GST to terminate. Lastly, we bound the message and communication complexity of the protocol. The number of rounds depends on when the network stabilizes (i.e., we cannot guarantee a decision for any consensus protocol prior to GST). We bound, however, the communication complexity of each round:

LEMMA 5.3 (POLYGRAPH COMPLEXITY). *After τ_{GST}, the Polygraph protocol has message complexity $O(n^3)$ and communication complexity $O(\kappa \cdot n^4)$, where n is the number of participants and κ is the security parameter.*

PROOF. The Polygraph protocol terminates in $O(t)$ rounds after τ_{GST} both to reach consensus or detect processes responsible for disagreement. As each round executes a bv-broadcast of $O(n^2)$ messages and as $t = O(n)$ we obtain $O(n^3)$ messages. The communication complexity is $O(\kappa \cdot n^4)$ since each message may contain a ledger of $O(n)$ signatures or $O(\kappa \cdot n)$ bits. The remainder of the protocol involves only $O(n^2)$ messages and only $O(n^3)$ communication complexity (e.g., for the coordinator to broadcast its message, and for processes to send their ECHO messages). □

Note that in the good case (after τ_{GST} and when $t < t_0$) Polygraph reaches consensus in three message delays. In Appendix C, we show that the multivalue generalization of the Polygraph protocol is also correct and accountable.

6 EXPERIMENTS: POLYGRAPH WITH A BLOCKCHAIN APPLICATION

To understand the overhead of Polygraph over a non-accountable consensus, we compare the throughput of the original Red Belly Blockchain [15] based on DBFT [14] and the "Accountable Red Belly Blockchain" based on Polygraph. The reason is that DBFT and Polygraph are both one-shot consensus protocols while a blockchain application allows

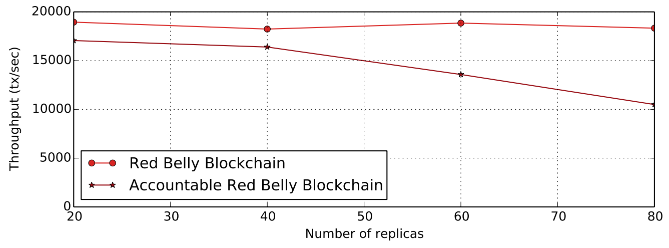

Figure 1: The overhead of accountability in the Red Belly Blockchain with 400 byte transactions cryptographically verified with ECDSA signatures and parameters secp256k1 when deployed on 80 replicas geo-distributed in Frankfurt, Ireland, London, N. California and N. Virginia.

for a more realistic comparison of the performance. To this end, we implemented the naive multivalue generalization of Polygraph as described in Appendices C and E. We then turn the Multivalue Polygraph Protocol into a state machine replication with the classic technique [2, Fig.4] by tagging messages with their consensus instance. Finally, we build a blockchain layer using the Red Belly Blockchain UTXO model with signed Bitcoin-style transaction requests. We implemented Polygraph using the RSA 2048 bits signature scheme to authenticate messages.

We deployed both blockchains on up to $n = 80$ c4.xlarge AWS virtual machines located in 5 availability zones on two continents: Frankfurt, Ireland, London, North California and North Virginia. All machines issue transactions, insert transactions in their memory pool, propose blocks of 10,000 transactions, verify transaction signatures (and account integrity, and run their respective consensus algorithm with $t = t_0 = \lceil \frac{n}{3} \rceil - 1$, before storing decided blocks to non-volatile storage. Red Belly Blockchain commits tens of thousands of Bitcoin TPS on hundreds of geo-distributed processes [15].

Figure 1 represents the throughput while increasing the number of consensus participants from 20 (4 machines per zone) to 80 (16 machines per zone). We observe that the cost of accountability varies from 10% at 20 processes to 40% at 80 processes. The reason is twofold: (i) the accountability presents an overhead due to the signing and verification of messages authenticated using RSA 2048 bits in addition to the verifications of built-in Red Belly Blockchain UTXO transaction signatures. (ii) the c4.xlarge instances are low-end instances with an Intel Xeon E5-2666 v3 processor of 4 vCPUs, 7.5 GiB memory, and “moderate” network performance. On the one hand, as was observed [15], even in this low-end situation, the Red Belly Blockchain scales in that its performance does not drop. On the other hand, we can see the Accountable Red Belly Blockchain still offers a throughput of more than 10,000 transactions per second at 80 geo-distributed processes, which remains superior to most non-accountable blockchains. Finally, the Accountable Red Belly Blockchain commits several thousands of transactions per second on 80 geodistributed machines, which indicates that the cost of accountability remains practical.

7 CONCLUSION

We introduced Polygraph, the first accountable Byzantine consensus algorithm. If $t < n/3$, it ensures consensus, otherwise it eventually detects users that cause disagreement. Thanks to its bounded justification size, Polygraph can be used to commit tens of thousands of blockchain transactions.

Acknowledgements. We wish to thank Alejandro Ranchal Pedrosa for his help in reproducing the experimental results. This research is supported under Australian Research Council DP180104030 and FT180100496.

REFERENCES

[1] Mostéfaoui A., Moumen H., and Raynal M. Signature-free asynchronous Byzantine consensus with $T < N/3$ and $O(N^2)$ messages. In *PODC*, pages 2–9, 2014.
[2] Abhinav Aggarwal and Yue Guo. A simple reduction from state machine replication to binary agreement in partially synchronous or asynchronous networks. *IACR Cryptology ePrint Archive*, 2018.
[3] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *SOSP*, 2005.
[4] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Dissecting tendermint. In *NETYS*, pages 166–182, 2019.
[5] Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Blockchain abstract data type. In *SPAA*, pages 349–358, 2019.
[6] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation 75:130-143*, 1985.
[7] Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, The University of Guelph, June 2016.
[8] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. Technical Report 1807.04938v3, arxiv, 2018.
[9] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. Technical Report 1710.09437v4, arXiv, Jan 2019.
[10] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *SRDS*, pages 191–202, 2005.
[11] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.
[12] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM, Volume 43 Issue 2, Pages 225-267*, 1996.
[13] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2019:587, 2019.
[14] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *NCA*, pages 1–8, 2018.
[15] Tyler Crain, Chris Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *S&P*, 2021.

[16] Dorothy E. Denning. An intrusion-detection model. *IEEE Trans. Software Eng.*, 13(2), 1987.
[17] The eth2 upgrades. Accessed: 2020-12-12, https://ethereum.org/en/eth2/.
[18] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. *SOSP*, 2007.
[19] Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *OPODIS*, pages 99–114, 2009.
[20] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *LICS*, pages 27–30, 2016.
[21] libhotstuff. Accessed: 2021-03-01 https://github.com/hot-stuff/libhotstuff.
[22] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Detection and removal of malicious peers in gossip-based protocols. In *In Proceedings of FuDiCo*, June 2004.
[23] Kim P. Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. Byzantine fault detectors for solving consensus. *British Computer Society*, 2003.
[24] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *CCS*, pages 1751–1767, 2020.
[25] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
[26] Butler W. Lampson. Computer security in the real world. In *In Proc. Annual Computer Security Applications Conference*, December 2000.
[27] Jinyuan Li and David Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, pages 10–10, 2007.
[28] Dahlia Malkhi and Michael K. Reiter. Unreliable intrusion detection in distributed computations. In *CSFW*, pages 116–125, 1997.
[29] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *PODC*, pages 108–117, 2002.
[30] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *NSDI*, page 11, 2007.
[31] Roberto De Prisco, Dahlia Malkhi, and Michael K. Reiter. On *k*-set consensus problems in asynchronous systems. In *PODC*, pages 257–265, 1999.
[32] Alejandro Ranchal-Pedrosa and Vincent Gramoli. Blockchain is dead, long live blockchain! Technical Report 10541v2, arXiv, 2020.
[33] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. Technical Report 2010.06785, arXiv, 2020.
[34] Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain Byzantine fault tolerance. Technical Report 1909.07453v2, arXiv, 2019.
[35] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.
[36] Aydan R. Yumerefendi and Jeffrey S. Chase. Trust but verify: accountability for network services. In *Proceedings of the 11st ACM SIGOPS European Workshop*, page 37, 2004.
[37] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. *TOS*, 3(3):11:1–11:33, 2007.

A PROOFS OF THE IMPOSSIBILITY RESULT

Process. More formally, a process $p_i, i \in [1, n]$ is seen as a Mealy machine $(S, S_0, \Sigma, \Lambda, \delta, \omega)$ where S is a set of states, S_0 a set of initial states, Σ an input alphabet of receivable message, Λ an output alphabet of messages to send, $\delta : \Sigma^n \times S \rightarrow S$ a transition function and $\omega : \Sigma^n \times S \rightarrow \Lambda^n$ an output function. The absence of message is denoted by $\{\epsilon\}$, $\{\epsilon\} \in \Sigma$ and $\{\epsilon\} \in \Lambda$. The process is fitted with a clock which allows it to take a transition after a certain timeout with $\{\epsilon\}$ as argument for δ.

The agreement according to a local view. A process executing an agreement protocol follows an ordered succession of states $p(s_k) = s_0, ..., s_k$ called a path. On the path, the process has received messages which constitutes an history $h(p(s_k))$. The set S is a partition of the set of states before decision S_{bd} and the set of states after decision S_{ad}. The image of the restriction of δ on S_{ad} is S_{ad}, which means a decision is irrevocable. Among the attribute of $s \in S$, we note $s.r$ the associated round number and $s.dec$ the decided value.

To show that it is impossible to devise a swift Accountability algorithm, we show that no consensus algorithm can be safe when $t \geq n/3$ and that it is impossible given a consensus algorithm to design a swift verification algorithm for it when $t_0 < t$. Let $n \in \{3t_0 + 1, 3t_0 + 2, 3t_0 + 3\}$. Let P, Q and R be three sets of n processes such that $|P| \leq t_0$, $|R| \leq t_0$ and $|Q| = n - |P| - |R|$. The proofs rely on the indistinguishability between pairs of distinct scenarios A, B and C such that processes in P cannot distinguish A from C and R cannot distinguish B from C.

- Scenario A: All initial values are 0 and the processes in R are inactive. The messages sent from $P \cup Q$ to $P \cup Q$ are delivered in time 1. By t-resiliency, processes in P reach a decision (0 by validity) within a certain time, noted T_A. To take this decision, every process i of P followed a path $s_0^i, ..., s_{bd}^i, s_{ad}^i$ where $s_{bd}^i \in S_{bd}$ and $s_{ad}^i.dec = 0$.
- Scenario B: All initial values are 1 and the processes in P are inactive. The messages sent from $R \cup Q$ to $R \cup Q$ are delivered in time 1. By t-resiliency, processes in R reach a decision (1 by validity) within a certain time, noted T_B. To take this decision, every process j of R followed a path $s_0^j, ..., s_{bd}^j, s_{ad}^j$ where $s_{bd}^j \in S_{bd}$ and $s_{ad}^j.dec = 1$
- Scenario C: All initial values in P are 0, all initial values in R are 1 and processes of Q are Byzantine. These Byzantine processes behave with respect to those in P exactly as they do in Scenario A and with respect to those in R exactly as they do in Scenario B. Messages sent from $P \cup Q$ to $P \cup Q$ are delivered in time 1, as well as the ones from $R \cup Q$ to $R \cup Q$, while the ones from $P \cup R$ to $P \cup R$ are delivered in a time greater than $\max(T_A, T_B)$.

THEOREM A.1 (THEOREM 4.1). *In a partially synchronous system, no algorithm solves both the Byzantine consensus problem when $t < n/3$ and the agreement and validity of the Byzantine consensus problem when $t_0 < t$.*

Proof. Assume for the sake of contradiction that it exists an algorithm preserving the agreement for $t_0 < t$. Because scenarios A and B are indistinguishable from P's standpoint while scenarios B and C are indistinguishable for R's standpoint, P must decide 0 while R must decide 1 in scenario C. This yields a contradiction. □

Theorem A.2 (Theorem 4.2). *For an algorithm that solves consensus when $t < n/3$, there is no swift verification algorithm when $t > t_0$.*

Proof. Assume for the sake of contradiction that such a swift verification algorithm exists. The paths followed by the honest processes are the same in the scenario C, than in the scenarios A and B. Because no comission message has been sent in A and B, $\forall i \in P, V(s^i_{bd}) = \emptyset$ and $\forall j \in R, V(s^j_{bd}) = \emptyset$. But in scenario C, we should have $\exists i \in PV(s^i_{bd}) \neq \emptyset$ or $\exists j \in R, V(s^j_{bd}) \neq \emptyset$ which yields a contradiction. □

B PROOF OF CORRECTNESS OF POLYGRAPH

In this section, we provide the complete proofs that the binary consensus protocol presented in Section 5 is a correct, accountable Byzantine agreement protocol. First, in Section B.1, we focus on the properties satisfied by the accountable BV-Broadcast algorithm. Then we return to show the correctness of the consensus algorithm, as well as prove that it ensures accountability.

B.1 Acccountable BV-Broadcast

The BV-broadcast algorithm is presented in lines 1–9 of Algorithm 2. The protocol and proof presented here are quite similar to that in [14], with small changes to accomodate ledgers.

First, we prove that it satisfies certain useful properties when $t \leq t_0$. (These properties will not necessarily hold when there are more than t_0 Byzantine processes.) We begin with a simple structural property (that is later useful for ensuring validity of consensus): a value is only delivered if at least one honest process sends it.

Lemma B.1 (BV-Justification). *If $t \leq t_0$ and if p_i is honest and $v \in$ bin_values, then v has been BV-broadcast by some honest process.*

Proof. By contraposition, assume v has been BV-broadcast only by faulty processes, i.e., by at most t_0 processes. Then no process ever receives $t_0 + 1$ distinct BVAL messages for v, and hence the conditions on lines 4 and 7 are never met. Thus, no process ever adds v to *bin_values*. □

We next show that BV-broadcast satisfies some typical properties of reliable broadcast, i.e., if at least $t_0 + 1$ process BV-broadcast the same value, then every honest process delivers it; and if p_i is honest and delivers a value, then every honest process also delivers it.

Lemma B.2 (BV-Obligation). *If $t \leq t_0$ and at least $(t_0 + 1)$ honest processes BV-Broadcast the same value v, then v is eventually added to the set bin_values of each non-faulty process p_i.*

Proof. Let v be a value such that $(t_0 + 1)$ honest processes invoke BV-broadcast(MSG, $v, \cdot, \cdot, \cdot$). Each of these processes then sends a BVAL message with value v and a valid ledger, and consequently each honest process receives at least $t_0 + 1$ BVAL messages for value v along with valid ledgers for v. Therefore each honest process (i.e., at least $2t_0 + 1 \leq n - t_0$) broadcasts a BVAL message for v with a valid ledger, and consequently eventually every honest process receives receives at least $2t_0 + 1$ BVAL messages for v. Thus every honest process adds value v to *bin_values.* □

Lemma B.3 (BV-Uniformity). *If $t \leq t_0$ and a value v is added to the set bin_values of an honest process p_i, eventually $v \in$ bin_values at every honest process p_j.*

Proof. If a value v is added to the set *bin_values* of an honest process p_i, then this process has received at least $(2t_0 + 1)$ distinct BVAL messages for v with valid ledgers (line 7). This implies that at least $(t_0 + 1)$ different honest processes sent BVAL messages with valid ledgers. So every non-faulty process receives at least $(t_0 + 1)$ BVAL messages with the value v. Therefore every honest process eventually broadcasts a BVAL message for v, and so eventually every honest process receives receives at least $2t_0 + 1 \leq n - t_0$ BVAL messages for v with valid ledgers. Thus every honest process adds value v to *bin_values.* □

Finally, we show termination:

Lemma B.4 (BV-Termination). *If $t \leq t_0$ and if every honest process BV-broadcasts some value, then eventually, every honest process has at least one value in bin_values.*

Proof. As there are at least $(n - t_0)$ honest processes, each of them BV-broadcasts some value, and '0' and '1' are the only possible values, it follows that there is a value $v \in \{0, 1\}$ that is BV-broadcast by at least $(n - t_0)/2 \geq t_0 + 1$ processes (since one of the two values must be BV-broadcast by at least half the honest processes). The claim then follows by Lemma B.2. □

Finally, we observe the straightforward fact that messages are only delivered with valid ledgers (with the exception of messages in Round 1, and message containing value 1 in Round 2):

Lemma B.5 (BV-Accountability). *If a value v is added to the set bin_values of a non-faulty process p_i in round r (with the exception of messages in Round 1, and message containing value 1 in Round 2), then associated with the value v is a valid ledger from round $r-1$.*

Proof. Since every BVAL message without a valid ledger is discarded (aside from above mentioned exceptions), it follows immediately that when the conditions on lines 4 and 7 are met, then the process has access to a valid ledger, which is then included when the value is added to *bin_values*. □

B.2 Accountable Byzantine Agreement

Here, we prove that the Polygraph protocol is a correct accountable Byzantine agreement protocol. We begin with the standard properties of consensus, which hold when $t \leq t_0$, and then continue to discuss accountability. First, we observe that if every honest process begins a round r with the same estimate, then that value is decided either in round r or round $r+1$. This follows immediately from the fact that if every honest process BV-broadcasts the same value, then that is the only value delivered, and so it is the only value that remains in the system.

Lemma B.6. *Assume that each honest process begins round r with the estimate v. Then every honest process decides v either at the end of round r or round $r+1$.*

Proof. Since every honest process BV-broadcasts the value v, we know from Lemma B.2 that v is eventually delivered to every honest process, and from Lemma B.1 that v is the only value delivered to each honest process. Since v is the only value in bin_values_i for each honest p_i, it is also the only value echoed via aux_i messages, and eventually it is the only value in $values_i$.

There are now two cases. If $v = r \mod 2$, every honest process decides v. Otherwise, every honest process continues to the next round with its estimate equal to v. In the next round, $v = (r+1) \mod 2$, and by the same argument, every process will then decide v in round $r+1$. □

We can now observe that if $t \leq t_0$, we get agreement, because after the first round in which some process decides v, then every process adopts value v as its estimate.

Lemma B.7 (Agreement). *If $t \leq t_0$ and some honest processes p_i and p_j decide v and w, respectively, then $v = w$.*

Proof. Without loss of generality, assume that p_i decides no later than p_j. Assume for the sake of contradiction that $v \neq w$. Assume that p_i decides v in round r. If process p_j also decides in round r, then $v = w = r \mod 2$. Thus, we conclude that p_j decides in some round $> r$.

In round r, we know that $values_i = \{v\}$. This implies that i received at least $n - t_0$ distinct ECHO messages containing only value v. Consider now the ECHO messages received by some other honest process p_k. If $values_k = \{v\}$ or $values_k = \{v, w\}$, then process p_k adopts estimate v. Otherwise, process p_k has $values_k = \{w\}$, which implies that it received at least $n - t_0$ distinct ECHO messages containing only value w.

Thus there are at least $t_0 + 1$ processes that sent an ECHO message to p_i containing only v and an ECHO message to p_k containing only w. That, however, is illegal (see line 21), as a process must send the same ECHO message to all. Since only $t \leq t_0$ processes are Byzantine, this is impossible, so we conclude that every honest process adopts estimate v by the end of round r.

We then conclude, by Lemma B.6, that every honest process decides value v in either round $r+1$ or round $r+2$. (In this case, of course, it will be round $r+2$.) □

It is immediate from BV-broadcast that Polygraph guarantees a stronger form of validity (Lemma B.8) than the one required by blockchains (Corollary B.9). (The general Polygraph protocol stated in Appendix C that applies to arbitrary values only ensures the required validity.)

Lemma B.8 (Strong validity). *If $t \leq t_0$ and an honest process decides v, then some honest process proposed v.*

Proof. Let round r be the first round where a process p_i adopts a value v that was not initially proposed by an honest process at the beginning of round 1. There are two possibilities depending on whether p_i adopts v in line 24 or line 28.

Assume that honest process p_i in round r adopts value v in line 24. Then $values_i = \{v\}$ in round r. This can only happen if $bin\_values_i = \{v\}$, since ComputeValues only returns values that are in bin_values_i or aux_i, and aux_i only includes values in bin_values_i. However, bin_values_i only includes values delivered by BV-broadcast, and Lemma B.1 implies that every value delivered value was BV-broadcast by an honest process. So value v was the estimate of an honest process p_j at the beginning of round r. If $r = 1$ then we are done. If $r > 1$, then we conclude that v was adopted as an estimate in round $r-1$ by process p_j, and hence by induction, we conclude that v was initially proposed by an honest process at the beginning of round 1.

Assume that honest process p_i executes line 28, adopting the parity of the round number as its estimate. This implies that $values_i = \{0, 1\}$ in round r. This can only happen if $bin\_values_i = \{0, 1\}$ also, since ComputeValues only returns values that are in bin_values_i or aux_i, and aux_i only includes values in bin_values_i. However, bin_values_i only includes values delivered by BV-broadcast, and Lemma B.1 implies that every value delivered value was BV-broadcast by an honest process. So at least one honest process p_j began round r with value '0' and at least one honest process p_k began round r with '1'. If $r = 1$, then we are done. Otherwise, we conclude that p_j adopted '0' in round $r-1$ and p_k adopted '1' in round $r-1$, and hence by induction we conclude that both '0' and '1' were initially proposed by honest processes at the beginning of round 0. □

Corollary B.9 (Validity). *If all processes are honest and begin with the same value, then that is the only decision value.*

Finally, we argue that the protocol terminates:

Lemma B.10 (Termination). *If $t \leq t_0$, every honest process decides.*

Proof. First, we observe that processes continue executing increasing rounds (i.e., no process gets stuck in some round). Assume, for the sake of contradiction, that r is the first round where some process gets stuck forever, and p_i is the process that gets stuck.

A process cannot get stuck in line 12 or line 15 waiting for a BV-broadcast, since every honest process performs a BV-broadcast and so by Lemma B.4, every honest process eventually delivers a value. (And a process cannot get stuck waiting on a timer, since the timer will always eventually expire.)

A process also cannot get stuck waiting on line 22: Eventually process p_i will receive ECHO messages from each of the $n - t_0$ honest processes. And by Lemma B.3, every value that is delivered by BV-broadcast to an honest process will eventually be delivered to p_i. Specifically, every value that is included in an ECHO message from an honest process is eventually delivered to bin_values_i. Thus, eventually a set of $n - t_0$ messages is identified, and the waiting condition on line 22 is satisfied.

The last issue that might prevent progress is if process p_i, or some other process, cannot transmit a proper message due to missing ledgers. This can only be a problem with messages being BV-broadcast. If a process completed round $r-1$ and its estimate was $\neq r-1 \mod 2$, then it could always construct a proper ledger in round $r-1$ from the estimates received. If a process completed round $r-1$ and its estimate was $= r-1 \mod 2$, then it must have received a valid ledger for the value in round $r-1$ as part of the BV-broadcast; otherwise, it could not have completed round $r-1$.

Thus we conclude that every process executes an infinite number of rounds. The remaining question is whether processes ever decide. Consider the first round r after GST where the timer is sufficiently large that (i) every BV-broadcast message is delivered, and (ii) the coordinators message is delivered before the timer expires. In this case, every honest process will prioritize the coordinator's value, adopting it as their *aux* message in line 18, echoing it in line 21, and adding only that value to *values* in line 22. Thus at the end of round r, every process adopts the same value, and hence decides either in round r or round $r+1$ by Lemma B.6. □

Thus we conclude (see Theorem 5.1) that when $t \leq t_0$, the binary agreement protocol is correct. We now consider the case where $t > t_0$ and show that it still provides accountability.

Lemma B.11 (Accountability). *If $t > t_0$ and two honest processes p_i and p_j decide different values v and w, then eventually every honest process receives a ledger and a certificate that conflict (providing irrefutable proof that a specific collection of $t_0 + 1$ processes are Byzantine).*

Proof. Assume that p_i decided v in round r and p_j decided w in the round r' where $w = not(v) = 1 - v$ and $r \leq r'$. It is undeniable (by construction, line 25–26) that $v = r \ mod\ 2$. There are only four possible cases to consider:

(1) *Case 1:* $values_j^r \neq \{0, 1\}$
(2) *Case 2:* $values_j^r = \{0, 1\}$ and $values_j^{r+1} \neq \{v\}$
(3) *Case 3:* $values_j^r = \{0, 1\}$ and $values_j^{r+1} = \{v\}$ and $values_j^{r+2} \neq \{v\}$
(4) *Case 4:* $values_j^r = \{0, 1\}$ and $values_j^{r+1} = \{v\}$ and $values_j^{r+2} = \{v\}$

We now consider each of the cases in turn.

Case 1. If $values_j^r = \{v\}$, then process p_j would have decided $v \neq w$ in round r. So we conclude that $values_j^r = \{w\}$. In that case, $ledger[r]_j$ and $certificate[r]_i$ conflict.

Case 2. Assume $values_j^r = \{0, 1\}$ and $w \in values_j^{r+1}$. This implies that $w \in bin\_values_j$ in round $r+1$. Notice that round $r+1$ cannot be round 1, and if it is round 2, then the value $v = 1$ and so $w \neq 1$. Thus, BV-Accountability (Lemma B.5 indicates that p_j receives a valid ledger for w from round r. That ledger conflicts with the certificate for v from r (and consists of $n - t_0$ distinct ECHO messages containing only value w in round r).

Case 3. Assume $values_j^r = \{0, 1\}$, $values_j^{r+1} = \{v\}$, and $w \in values_j^{r+2}$. This implies that $w \in bin\_values_j$ in round $r+2$. Since round $r+2 > 2$, BV-Accountability (Lemma B.5 indicates that p_j receives a valid ledger for w from round $r+1$. Since $w = r+1 \mod 2$, the ledger for w from round $r+1$ is a copy of a ledger for w from round r, which therefore conflicts with the certificate for v for round r (and consists of $n - t_0$ distinct ECHO messages containing only value w in round r).

Case 4. Assume $values_j^r = \{0, 1\}$, $values_j^{r+1} = \{v\}$, and $values_j^{r+2} = \{v\}$. Then process p_j decides v in round $r+2$ and there is agreement.

All the cases have been examined, and in each case, process p_j has a ledger constructed in round r conflicting with the certificate delivered from process p_i. The conflicting ledger/certificate each contain $n - t_0$ signed, distinct ECHO messages containing only value v and only value w respectively. Since any two sets of size $n - t_0$ have an intersection of size $t_0 + 1$, the signatures in the conflicting ledgers prove the existence of a set G of $t_0 + 1$ Byzantine processes. □

C MULTIVALUE CONSENSUS

In this section, we discuss how to generalize the binary consensus to ensure accountable Byzantine agreement for arbitrary values. We follow the approach from [14]: First, all n processes use a reliable broadcast service to send their proposed value to all the other n processes. Then, all the process participate in parallel in n binary agreement instances, where each instance is associated with one of the processes. Lastly, if j is the smallest binary consensus instance to decide 1, then all the processes decide the value received from process p_j.

The key to making this work is that we need the reliable broadcast service to be accountable, that is, if it violates the reliable delivery guarantees, then each honest process has irrefutable proof of the culpability of $t + 1$ processes. Specifically, we want a single-use reliable broadcast service that allows each process to send one message, delivers at most one message from each process, and guarantees the following properties:

- *RB-Validity:* If an honest process RB-delivers a message m from an honest process p_j, then p_j RB-broadcasts m.
- *RB-Send:* If $t \leq t_0$ and p_j is honest and RB-broadcasts a message m, then all honest processes eventually RB-deliver m from p_j.
- *RB-Receive:* If $t \leq t_0$ and an honest process RB-delivers a message m from p_j (possibly faulty) then all honest processes eventually RB-deliver the same message m from p_j.
- *RB-Accountability:* If an honest process p_i RB-delivers a message m from p_j and some other honest process p_j RB-delivers m' from p_j, and if $m \neq m'$, then eventually every process has irrefutable proof of the culpability of $t_0 + 1$ processes.

The resulting algorithm provides a weaker notion of validity: if all processes are honest, then the decision value is one of the values proposed. (A stronger version of validity could be achieved with a little more care, but is not needed for blockchain applications, which depend on an external validity condition.)

We first, in Section C.1, present the general multivalue algorithm, and prove that it is correct—assuming the existing of a reliable broadcast service satisfying the above properties. Then, in Section C.2, we describe the reliable broadcast service and prove that it is correct.

C.1 Accountable Byzantine Agreement

We now present the algorithm in more detail. The general algorithm has three phases.

- First, in lines 1–8, each process uses reliable broadcast to transmit its value to all the others. Then, whenever a process receives a reliable broadcast message from a process p_k, it proposes '1' in binary consensus instance k. The first phase ends when there is at least one decision of '1'.
- Second, in lines 10–12, each process proposes '0' in every remaining binary consensus instance for which it has not yet proposed a value. The second phase ends when every consensus instance decides.
- Third, in lines 14–16, each process identifies the smallest consensus instance j that has decided '1'. (If there is no such consensus instance, then it does not decide at all.) It then waits until it has received the reliable broadcast message from p_j and outputs that value.

First, we argue that it solves the consensus problem as long as $t \leq t_0$:

Lemma C.1 (Agreement). *If $t \leq t_0$, then every honest process eventually decides the same value. If all processes are honest and propose the same value, that is the only possible decision.*

Proof. First we focus on termination. Since $t \leq t_0$, by the RB-Send property, we know that every honest process eventually delivers every value that was proposed by an honest process. Assume for the sake of contradiction that no binary consensus instance every decides '1'. Then eventually, every honest process proposes '1' for every binary consensus instance associated with an honest process. By the validity and termination properties of binary consensus (and since $t \leq t_0$), we conclude that these instances all decide '1', which is a contradiction. Thus eventually every honest process executes line 10.

Since every honest process eventually proposes a value to every binary consensus instance, we conclude (since $t \leq t_0$) that eventually every binary consensus instance decides and every honest process reaches line 15. Let j be the minimum binary consensus instance that decides '0'. Assume (for the sake of contradiction) that no honest process received the value from p_j. Then every honest process proposed '0' to the binary consensus instance for j, and hence by the validity property, the decision would have been '0', i.e., a contradiction. Thus we know that at least one honest process received the value that was reliably broadcast by p_j.

Algorithm 3 The (Naive) Multivalue Polygraph Protocol

```
1: gen-propose($v_i$):
2:   RB-broadcast(EST, $\langle v_i, i\rangle$) $\longrightarrow$ $messages_i$          ▷ reliable broadcast value to all
3:
4:   repeat:          ▷ when you recieve a value from $p_k$, begin consensus instance $k$ with a proposal of 1
5:     if $\exists\, v, k$ : (EST, $\langle v, k\rangle$) $\in messages_i$ then
6:       if BIN-CONSENSUS[$k$] not yet invoked then
7:         BIN-CONSENSUS[$k$].bin-propose(1) $\longrightarrow$ $bin\text{-}decisions[k]_i$
8:   until $\exists k$ : $bin\text{-}decisions[k] = 1$          ▷ wait until the first decision
9:
10:  for all $k$ such that BIN-CONSENSUS[$k$] not yet invoked do          ▷ begin consensus on the remaining instances
11:    BIN-CONSENSUS[$k$].bin-propose(0) $\longrightarrow$ $bin\text{-}decisions[k]_i$          ▷ for these, propose 0
12:  wait until for all $k$, $bin\text{-}decisions[k] \neq \bot$          ▷ wait until all the instances decide
13:
14:  $j = \min\{k : bin\text{-}decisions[k] = 1\}$          ▷ choose the smallest instance that decides 1
15:  wait until $\exists\, v$ : (EST, $\langle v, j\rangle$) $\in messages_i$          ▷ wait until you receive that value
16:  decide $v$          ▷ return that value
```

Finally, by the RB-Receive property, we know that every honest process must eventually deliver the value that was reliably broadcast by p_j, and hence every process eventually returns a value, i.e., we satisfy the termination property.

Next, we argue agreement. Since $t \leq t_0$, by the guarantees of the binary consensus instances, every honest process decides the same thing for each of the instances. Therefore, all honest processes will choose the same j that is the minimum binary consensus instance that decides 1. As we have already argued, every honest process must eventually deliver the value reliably broadcast by p_j, and that must be the same value. This guarantees agreement.

Finally, we argue validity: if all the processes are honest, then every value received by reliable broadcast is from an honest value, and hence validity is immediate. □

Next, we prove that if there is any disagreement, then the algorithm guarantees accountability:

Lemma C.2 (Accountability). *If two honest process p_i and p_j decide different values, then honest processes eventually receive irrefutable proof of at least $t_0 + 1$ Byzantine processes.*

Proof. First, assume that p_i and p_j decide different values for some binary consensus instance. Then, by the accountability of binary consensus, we know that every process eventually receives the desired irrefutable proof. Alternatively, if p_i and p_j agree for every instance of binary consensus, then they choose the same value k that is the minimum binary consensus instance to decide '1', and they output the value delivered by the reliable broadcast service from p_k. (Recall that the reliable broadcast service delivers only one value from each process, and p_i and p_j do not decide until they receive that value.) However, by the RB-accountability property, we conclude that eventually every process receives irrefutable proof of at least $t_0 + 1$ Byzantine processes. □

C.2 Reliable Broadcast

We now describe the reliable broadcast service, which is a straightforward extension of the broadcast protocol proposed by Bracha [6]. A process begins by broadcasting its message to everyone. Every process that receives the message directly, echoes it, along with a signature. Every process that receives $n - t_0$ distinct ECHO messages, sends a READY message. And if a process receives $t_0 + 1$ distinct READY messages, it also sends a READY message. Finally, if a process receives $n - t_0$ distinct READY messages, then it delivers it.

The key difference from [6] is that, as in the binary value consensus protocol, we construct ledgers to justify the messages we send. Specifically, when a process sends a READY message, if it has received $n - t_0$ distinct ECHO messages, each of which is signed, it packages them into a ledger, and forwards that with its READY message. Alternatively, if a process sends a READY message because it received $t_0 + 1$ distinct READY messages, then it simply copies an existing (valid) ledger. Either way, if a process p_i sends a READY message for value v which was sent by process p_j, then it has stored a ledger containing $n - t_0$ signed ECHO messages for v, and it has sent that ledger to everyone.

As before, two ledgers conflict if they justify two different values v and v', both supposedly sent by the same process p_j. In that case, one ledger contains $n - t_0$ signed ECHO message for v and the other contains $n - t_0$ signed ECHO message for v'. Since any two sets of size $n - t_0$ have an intersection of size $t_0 + 1$, this immediately identifies at least $t_0 + 1$ processes that illegally sent ECHO messages for both v and v'. These process can by irrefutably proved to be Byzantine.

We now prove that the reliable broadcast protocol satisfies the desired properties. First, we show that it delivers only one value from each process, and that if $t \leq t_0$, then it only delivers a value if it was previously RB-broadcast by that process:

Algorithm 4 Reliable Broadcast

```
1:  RB-broadcast($v_i$):                                                  ▷ only executed by the source
2:      broadcast(INITIAL, $v_i$)                                         ▷ broadcast value $v_i$ to all
3:  upon receiving a message (INITIAL, $v$) from $p_j$:
4:      broadcast(ECHO, $v$, $j$)                                         ▷ echo value $v$ to all
5:  upon receiving $n - t_0$ distinct messages (ECHO, $v$, $j$) and not having sent a READY message:
6:      Construct a ledger $\ell_i$ containing the $n - t_0$ signed messages (ECHO, $v$, $j$).
7:      broadcast(READY, $v$, $\ell_i$, $j$)                              ▷ send READY message and ledger for $v$ to all.
8:  upon receiving $t_0 + 1$ distinct messages (READY, $v$, $\cdot$, $j$) and not having sent a READY message:
9:      Set $\ell_i$ to be one of the (valid) ledgers received (READY, $v\ell$, $j$).
10:     broadcast(READY, $v$, $\ell$, $j$)                                ▷ send READY message for $v$ to all.
11: upon receiving $n - t_0$ distinct messages (READY, $v$, $\cdot$, $j$) and not having delivered a message from $j$:
12:     Let $\ell$ be one of the (valid) ledgers received (READY, $v$, $\ell$, $j$).
13:     deliver($v$, $j$)                                                 ▷ send READY message for $v$ to all
```

Lemma C.3 (*RB-Unicity*). *At most one value (v, j) is delivered from process p_j.*

Proof. Follows immediately by inspection: only one message from p_j is every delivered to each process.. □

Lemma C.4. *(*RB-Validity*) If $t \le t_0$, and if an honest process* RB-deliver*s a value v from an honest process p_j, then p_j* RB-broadcast*s m.*

Proof. A process only delivers a value v for p_j if it received (READY, v, $\cdot$, j) messages from $n - t_0$ processes, implying that at least one honest process sent a READY message for v. Let p_i be the first honest process to send a (READY, v, $\cdot$, j). In that case, we know that p_i must have received $n - t_0$ distinct (ECHO, v, j), implying that at least one honest process sent an ECHO message for v. An honest process only sends an (ECHO, v, j) message if it received v directly from p_j. And if p_j is honest, it onlys sends v if the value was RB-broadcast. □

Next, we prove a key lemma, showing that either all honest process send READY messages only for one value, or two conflicting ledgers are eventually received by all honest processes.

Lemma C.5. *Assume that p_i is an honest process that sends a* READY *message for value v and that p_j is an honest process that sends a* READY *message for value v'. Then either $v = v'$ or the ledgers ℓ_i and ℓ_j constructed by p_i and p_j, respectively, conflict.*

Proof. Process p_i sends the READY message for v either because (i) it has received $n - t_0$ distinct messages (ECHO, v, $\cdot$) and has constructed a ledger ℓ_i containing those signed messages, or (ii) it has received $t_0 + 1$ distinct READY messages (ECHO, v, $\cdot$, j), each containing a valid ledger for v, one of which it copies as ℓ_i. Either way, process p_i has a valid ledger ℓ_i containing $n - t_0$ signed echo messages for v. Similarly, by the same logic, process p_j has a valid ledger ℓ_j containing $n - t_0$ signed echo message for v'.

Since any two sets of size $n - t_0$ must have an intersection of size at least $t_0 + 1$, we conclude that if $v \neq v'$, then the ledgers ℓ_i and ℓ_j conflict, i.e., prove that at least $t_0{+}1$ processes illegally sent ECHO messages for both v and v'. □

We can now show that if an honest process performs a RB-broadcast, then as long as $t \le t_0$, every honest process delivers its message:

Lemma C.6. *(*RB-Send*) If $t \le t_0$, and if p_j is honest and* RB-broadcast*s a value v, then all honest processes eventually* RB-deliver *v from p_j.*

Proof. If p_j is honest, then it broadcasts its value v, properly signed, to all processes. All honest processes receive it directly from p_j and immediately broadcast an ECHO message. (Moreover, there is no other message they could echo, because there is no other message they could have received directly from p_j.)

Since $t \le t_0$, we know that there are at least $n - t_0$ honest processes that perform the ECHO, and hence every honest process receives at least $n - t_0$ ECHO messages, and hence every honest process broadcasts a READY message. (Of course there is no other message that an honest process could send a READY message for, since the first honest process to send a READY message for some other value must have received at least $n - t_0$ distinct ECHO messages for that value; at least one of those ECHO messages must have been sent by an honest process which received it directly from p_j, which—being honest—only sent value v.)

Since $t \le t_0$, there are at least $n - t_0$ honest processes that send READY messages, and so every honest process receives $n - t_0$ READY messages and delivers the value v from p_j. (Of course p_j cannot have delivered any other values v' from p_j earlier, since p_j is honest there is no other value v' that it RB-broadcast.) □

Next, we can show that if any honest process delivers a value v, then every honest process also delivers value v—as long as $t \le t_0$.

Lemma C.7. *(*RB-Receive*) If $t \leq t_0$ and an honest process p_k* RB-deliver*s a value v from p_j (possibly faulty), then all honest processes eventually* RB-deliver *the same message v from p_j.*

Proof. Assume p_k delivers v from p_j. In this case, p_k must have received at least $n - t_0$ valid READY messages for (v, j). Therefore, there must have been at least $t_0 + 1$ honest processes that broadcast valid READY messages. This implies that every honest process receives at least $t_0 + 1$ valid READY messages for (v, j), and hence also sends a READY message for (v, j). (Since $t \leq t_0$, we know that an honest process cannot send a READY message for any other value $v' \neq v$ for process j, by Lemma C.5.) Therefore everyone honest process receives at least $2t_0 + 1$ valid READY message for value v for process j, and hence delivers value v from p_j. □

Finally, we show the accountability property: either all honest processes deliver the same value, or two conflicting ledgers are received by every honest process:

Lemma C.8. RB-Accountability: *If an honest process p_i* RB-deliver*s value v from p_j and some other honest process p_j* RB-deliver*s v' from p_j, and if $v \neq v'$, then eventually every honest process receives two ledgers ℓ and ℓ' that conflict.*

Proof. If process p_i delivers v from p_j, then it received at least $n - t_0$ READY messages for value v, which implies that at least one honest process p_u sent a READY message for v. Similarly, since p_j delivers v' from p_j, we know that at least one honest process p_w sent a READY message for v'. By Lemma C.5, we know that if $v \neq v'$, then the ledgers ℓ_u and ℓ_w conflict. Moreover, since both p_u and p_w are honest, they sent their respective READY messages to all processes, and hence every honest process receives the conflicting ledgers. □

D OPTIMIZING THE MULTIVALUE POLYGRAPH

In this section, we present some optimizations (Algorithms 5, 6 and 7) to the Multivalue Polygraph Protocol to improve the bit-complexity by a linear multiplicative factor, from $O(\kappa \cdot n^5)$ to $O(\kappa \cdot n^4)$. To this end, we use two techniques:

First, we wait for the termination of certain "quick" instances that complete in 1 round. When these quick instances are decided, a correct process knows what it will propose for all the n different instances. This allows to couple the n instances, where each correct process sends a vector message that contains n message contents for the n different instances. (see algorithm A vector of n *aux* messages is called an hyper-*aux* message.

Second, we remark that (1) the Cachin-Tessaro optimization [10] reduces the bit complexity of the reliable broadcast for value with great length ℓ ($O(n\ell + \kappa' n^2)$ instead of $O(n^2\ell)$) and (2) if the *aux* messages were reliable broadcast, it would be not necessary to broadcast the heavy ledger in the Phase 1. Note that to bypass the need of a trusted dealer, we build upon recent results [24]. Indeed, if a correct is able to compute a ledger attached to a value v from some RB-delivered aux-messages, it will be eventually the case for any other correct process.

With these two observations, we can improve the bit-complexity of the algorithm, avoiding the costly broadcast of ledgers and replacing this cost by a cheaper one, i.e., the reliable broadcast of hyper-*aux* messages with the optimization of [10].

Quick and slow instances. We add to the Multivalue Polygraph protocol a variable *wait*, initially *wait* = 1. As long as *wait* = 1, we have: (a) any proposal for bit 0 is stored but not taken into account, (b) any proposal for round ≥ 2 is stored but not take into account.

We replace line 8 of algorithm 3 by line 10 at updated algorithm 5 with the following predicate, called the *(n-t_0)-predicate*: "**until** $\exists K_i, |K_i| \geq n - t_0, \forall k \in K_i, bin\_decision[k] = 1$". After line 10, the variable *wait* is updated to 0, that is the (a) proposals with bit 0 and (b) proposals for round $r > 1$ can be taken into account for the instances in $[1, n] \setminus K_i$.

Lemma D.1 (Liveness). *This transformation preserve Liveness.*

Proof. Eventually, it exists a set K_c, $|K_c| \geq n - t_0$ such that for every instance $k \in K_c$, the proposal is RB-delivered by every correct process, which then invokes *BIN-CONSENSUS*$[k]$*.bin_propose*(1). Thus, eventually, the set K_c receives $n - t_0$ proposals for the binary value 1 such that every process can decide binary value 1 at round 1. □

Lemma D.2 (Local 1 round shot). *For every process i, it exists a set K_i, $|K_i| \geq n - t_0$ such that process i decides the binary value* 1 *within* 1 *round at each instance $k \in K_i$.*

Proof. As liveness is preserved, the (n-t_0)-predicate is satisfied. □

Combining the instances. Each process i knows (1) it will eventually reach a state σ_i where K_i is decided in one round and (2) this is the case for any other correct process j.

Thus after reaching such a state, it becomes possible to start **all** the instances $\mathcal{I} = \{i_1, ..., i_n\}$ and **combine** the proposals of all the instances. To do so, the processes send vector that include all these instances, namely, the ones where processes already proposed at round 1. Thus at *hyper-round* 1 (which is this new round 1 within which all instances are treated altogether), a correct process is "repeating itself", which is not a problem. If a vector of hyper-round 1 does not

Algorithm 5 The Optimized Multivalue Polygraph Protocol

```
1: gen-propose($v_i$):
2:     $wait_i = 1$
3:     $hyper\text{-}proposal \in \{0,1\}^n$, initially $\forall k \in [1,n]$, $hyper\text{-}proposal[k] = 0$
4:     RB-broadcast(EST, $\langle v_i, i\rangle$) $\longrightarrow messages_i$          ▷ reliable broadcast value to all
5:
6:     repeat:          ▷ when you recieve a value from $p_k$, begin consensus instance $k$ with a proposal of 1
7:         if $\exists\, v, k$ : (EST, $\langle v, k\rangle$) $\in messages_i$ then
8:             if BIN-CONSENSUS$[k]$ not yet invoked then
9:                 BIN-CONSENSUS$[k]$.bin-propose(1) $\longrightarrow$ $bin\text{-}decisions[k]_i$
10:    until $\exists K_i, |K_i| \geq n - t_0, \forall k \in K_i, bin\_decision[k] = 1$          ▷ wait until the $n - t_0$ first decisions
11:
12:    $wait_i = 0$
13:    $\forall k \in K_i$, $hyper\text{-}proposal[k] = 1$          ▷ re-propose 1 for all the decided instances in $K_i$ and propose 0 for the other instances
14:    BIN-CONSENSUS.combine-propose($hyper\text{-}proposal$) $\longrightarrow$ $bin\text{-}decisions_i$          ▷ One message for all the instances
15:    wait until for all $k$, $bin\text{-}decisions[k] \neq \perp$          ▷ wait until all the instances decide
16:
17:    $j = \min\{k : bin\text{-}decisions[k] = 1\}$          ▷ choose the smallest instance that decides 1
18:    wait until $\exists\, v$ : (EST, $\langle v, j\rangle$) $\in messages_i$          ▷ wait until you receive that value
19:    decide $v$          ▷ return that value
```

correspond to the binary values that have been sent in the first phase (without vector), the vector is ignored. When a process j decides at instance k at round r_j^k, it continues to "help" until round $r_j^* + 2$, where r_j^* is the highest round of decision of j among all the instances.

At each round $r \in \mathbb{N}$, , for every process p, we note $\underline{AUX}_p[r] = \langle aux_p^{i_1}[r], ..., aux_p^{i_n}[r]\rangle_{\sigma_p}$ and call this message an hyper-*aux* message.

In the optimized version, the hyper-*aux* message is RB-bcast with the optimization of Cachin and Tessaro [10]) (see line 22 in algoritm 6).

Avoiding the bv-bcast of ledgers. Now, an hyper-estimation is justified by an hyper-*aux* message that has been RB-delivered. Thus, this justification will be eventually RB-delivered by all the correct processes, avoiding the costly broadcast of ledgers.

A disagreement implies a binary disagreement at a particular instance k. Two correct processes i and j that disagree at instance k at rounds r_i and r_j with $r_j > r_i$, can compare the hyper-*aux* messages that have been RB-delivered at round r_i.

At the end, after GST, the optimized multivalue protocol has a message complexity of $O(n)$ processes times $O(t_0)$ rounds times $O(n^2)$ (RB-bcast), which is $O(n^4)$ messages. The optimized multivalue protocol has a bit-complexity of $O(\kappa \cdot n^4)$, since the bit complexity of $RB-bcast$ is only $O(\kappa \cdot n^2)$ [10].

A brief explanation of the optimized Reliable-broadcast [10]. The algorithm of Cachin and Tessaro works as follows: At each round r, each process is supposed to reliably broadcast its hyper-*aux* $\underline{AUX}_j[r]$ to every other process. To do so it can use a Solomon code to build $\underline{\tilde{AUX}}_j[r]$ where $|\underline{\tilde{AUX}}_j[r]| = O(1)|\underline{AUX}_j[r]|$. Then it can split $\underline{\tilde{AUX}}_j[r]$ into n chunks $\underline{\tilde{AUX}_j^p}[r]_{p\in[1,n]}$ and send to each process the chunk and an additional constant-size piece of information used for verification. When a process p receives its chunk, it can verify it and then echo it. When a process q receives $n - t_0$ chunks $(\underline{\tilde{AUX}_j^p}[r])_{p\in\phi,|\phi|=n-t_0}$, it can build $\underline{AUX}_j[r]$ and verify it.

E APPLICATION TO BLOCKCHAIN

In this section we explain how the general Polygraph protocol can held blockchain service providers accountable to blockchain client processes that do not run the consensus as long as $t < 2n/3$. Note that a blockchain service can be implemented with a replicated state machine to which separate clients send requests. A predetermined set of n processes, called a *consortium*, can propose and decide valid blocks that they append to their local view of the blockchain through the *General Polygraph Protocol* that accepts arbitrary values. A client can send a get requests to the members of the consortium and if it receives the same view of the blockchain from a certain number m of members ($m = (n - t_0)$ by default), it considers the transactions of this common view as valid. In case of disagreement ($t > t_0$), the blockchain forks in that multiple blocks get appended to the same index of the chain, which could lead to *double spending* if the resulting branches have conflicting transactions.

Preliminaries. We now restate the existing blockchain formalism by Anceaume et al. [5]. A *blockchain* is a chain of blocks whose score() function takes as input a blockchain and returns its score s as a natural number, which can be its

Algorithm 6 Coupled Polygraph Protocols

```
1:  combined-propose($hyper\text{-}proposal$):
2:    $hyper\text{-}est_i = hyper\text{-}proposal$
3:    $r_i = 0$
4:    $timeout_i = 0$
5:    $hyper\text{-}aux\text{-}msgs_i[-1] = hyper\text{-}aux\text{-}msgs_i[0] = \emptyset$
6:    repeat:
7:      $r_i \leftarrow r_i + 1$;
8:      $timeout_i \leftarrow timeout_i + 1$
9:      $coord_i \leftarrow ((r_i - 1) \mod n) + 1$
    ▷ Phase 1:
10:     for $k \in [1:n]$ do
11:       light-bv-bcast(LEST$[r_i][k]$, $hyper\text{-}est[k]_i$, $i$, $hyper\text{-}bin\_values_i[k]$, $hyper\text{-}aux\text{-}msgs_i[r-1]$, $hyper\text{-}aux\text{-}msgs_i[r-2]$)   ▷ binary value broadcast the current estimate of all the instances, without broadcast any ledger. The bit-complexity is $O(n)$ per instance, per node, per round that is $O(n^4)$ in total after GST
12:     wait until $\forall k \in [1,n]$, $(hyper\text{-}bin\_values_i[r_i][k] \neq \emptyset$ and
13:       note $\{\underline{w}_i = (\{w^1\}, ..., \{w^n\})\})$ where $\forall k \in [1,n]$, $w^k$ is the first value bv-delivered at instance $k$ by $coord_i$.   ▷ hyper-bin_values stores binary messages received by binary value broadcast for all the instances
14:     if $i = coord_i$ then   ▷ coordinator rebroadcasts the first value received at each instance
15:       broadcast(HCOORD$[r_i]$, $\underline{w}_i$) $\longrightarrow hyper\text{-}messages_i$
16:     StartTimer($timeout_i$)
17:     wait until $timer_i$ expired
    ▷ Phase 2:
18:     if (HCOORD$[r_i]$, $\underline{w}_c$) $\in hyper\text{-}messages_i$ from $p_{coord_i}$ $\wedge \forall k \in [1,n]$, $\underline{w}_c[k] \in hyper\text{-}bin\_values_i[r_i][k]$) then
19:       $hyper\text{-}aux_i \leftarrow \underline{w}_c$
20:     else $hyper\text{-}aux_i \leftarrow \underline{w}_i$   ▷ otherwise, use any value received
21:     $signature_i = \text{sign}(hyper\text{-}aux_i, r_i, i)$   ▷ sign the hyper-aux messages. Each signed hyper-aux message has a size of $O(n+\kappa)$ (strictly lower than $O(n \cdot \kappa)$)
22:     RB-broadcast(HECHO$[r_i]$, $hyper\text{-}aux_i[r_i]$, $signature_i$) $\longrightarrow hyper\text{-}aux\text{-}msgs_i[r]$ ▷ Reliable broadcast the second phase messages of all the instances in a unique hyper-aux message with Cachin-Tessaro optimization. The bit-complexity per sender, per round is $O(n^2 \cdot \kappa)$, that is $O(n^4 \cdot \kappa)$ in total after GST
23:     wait until $\forall k \in [1,n]$, $hyper\text{-}values_i[k]$ = ComputeValues($hyper\text{-}aux\text{-}msgs_i[r][k]$, $hyper\text{-}bin\_values_i[k]$, $hyper\text{-}aux_i[k]$) $\neq \emptyset$
24:       where $hyper\text{-}aux\text{-}msgs_i[r][k] = \{haux[k] | haux \in hyper\text{-}aux\text{-}msgs_i[r]\}$
    ▷ Decision phase:
25:     for $k \in [1,n]$ do   ▷ for each instance
26:       if $hyper\text{-}values_i[k] = \{v\}$ then   ▷ if there is only one value at instance $k$, then adopt it
27:         $hyper\text{-}est_i[k] \leftarrow v$
28:         if $v = (r_i \mod 2)$ then   ▷ decide at instance $k$ if value matches parity
29:           if no previous decision by $p_i$ then decide($k, v$)
30:       else
31:         $hyper\text{-}est_i[k] \leftarrow (r_i \mod 2)$   ▷ otherwise, adopt the current parity bit
32:     nothing to do   ▷ No ledger to broadcast anymore because of the RB-bcast of Phase 2
```

Rules:

(1) Every message that is not properly signed by the sender is discarded.
(2) Every message that is sent by bv-broadcast without a valid hyper-ledger after Round 1, except for messages containing value 1 in Round 2, are discarded.
(3) On first discovering an hyper-ledger ℓ that conflicts with an hyper-certificate at an instance k, send ledger ℓ to all processes.

height, its weight, etc. A *blocktree* is a Mealy's machine whose states are countable, with an input alphabet comprising operation append(*block*) to append a block to the blocktree and operation read() that returns a blockchain, and an oracle Θ distributing permission tokens to processes for them to include a new block. The *blocktree strong (resp. eventual) consistency* is the conjunction of the following properties:

- *block validity*: each block in a blockchain returned by a read() operation is valid and has been inserted in the blocktree with the append() operation.
- *local monotonic read*: given a sequence of read() operations at the same process, the score of the returned blockchains never decreases.
- *ever growing tree*: given an infinite sequence of append() and read() operations, the score of the returned blockchains eventually grows.
- *strong (resp. eventual) prefix property*: for each blockchain returned by a read() operation with score s, then (resp. eventually) all the read() operations return blockchains sharing the same maximum common prefix of at least s blocks.

Algorithm 7 Coupled Helper Components

```
light-bv-broadcast(LBVAL[$k$][$r$], $val$, $i$, $bin\_values$, , $hyper\text{-}aux\text{-}msgs_{-1}$, $hyper\text{-}aux\text{-}msgs_{-2}$):
  broadcast(LBVAL[$k$][$r$], $val$) $\rightarrow$ $msgs$                                    ▷ broadcast message
  upon receipt of (LBVAL[$k$][$r$], $v$)
    if (LBVAL[$k$], $v$) received from ($t_0 + 1$) distinct processes and (LBVAL[$k$][$r$], $v$) not yet broadcast then
      broadcast($LBVAL[k][r]$, $v$)                                                       ▷ Echo after receiving ($t_0 + 1$) copies.
    if (BVAL[$k$], $v$) received from ($2t_0 + 1$) distinct processes and
      IsJustified($v$, $hyper\text{-}aux\text{-}msgs_{-1}$, $hyper\text{-}aux\text{-}msgs_{-2}$, $r$, $k$)        ▷ Need to build a correctly signed ledger to bv-deliver a value. then
      $bin\_values \leftarrow bin\_values \cup \{v\}$                                    ▷ deliver after receiving ($2t_0 + 1$) copies only

ComputeValues($messages$, $b\_set$, $aux\_set$):                                          ▷ No modification
  if $\exists S \subseteq messages$ where the following conditions hold:
    (i) $|S|$ contains ($n - t_0$) distinct ECHO[$r_i$] messages
    (ii) $aux\_set$ is equal to the set of values in $S$.
     then return($aux\_set$)
  if $\exists S \subseteq messages$ where the following conditions hold:
    (i) $|S|$ contains ($n - t_0$) distinct ECHO[$r_i$] messages
    (ii) Every value in $S$ is in $b\_set$.
     then return($V$ = the set of values in $S$)
  else return($\emptyset$)

IsJustified($v$, $hyper\text{-}aux\text{-}msgs_{-1}$, $hyper\text{-}aux\text{-}msgs_{-2}$, $r$, $k$):   ▷ check if a bv-broadcasted value is justified by RB-delivered enough hyper-aux messages
  if $r > 1$ then
    if $v == (r \mod 2)$ then
      if $|\{h\text{-}aux \in hyper\text{-}aux\text{-}msgs_{-1} | h\text{-}aux[k] == v\}| \geq n - t_0$ then
        return true
      else return false
    else
      if $|\{h\text{-}aux \in hyper\text{-}aux\text{-}msgs_{-2} | h\text{-}aux[k] == v\}| \geq n - t_0$ or $r == 2$ then
        return true
      else return false
  else
    return true
```

Now we consider three cases depending on the number of Byzantine processes, namely the nominal case, the degraded case and the zombie case when respectively $t \leq t_0$, $t_0 < t < (n - t_0)$ and $(n - t_0) \leq t \leq n$.

Nominal case ($t \leq t_0$). In this case the strong consistency is preserved. To read the state of a blockchain, a client asks the n members of the consortium (read() invocation) and waits for $m = n - t_0$ identical answers (read() response events). Indeed, if the assumption $t \leq t_0$ holds, the consensus will finish (the ever growing tree property is ensured) and the client will eventually receive at least $n - t_0$ identical answers. But nothing prevents the Byzantine processes to stay mute forever or give false answers, that is why a client does not expect (a priori) more than $n - t_0$ answers. While $t \leq t_0$, the frugal $\Theta_{F,k=1}$ oracle manages tokens in a controlled way to guarantee that no more than $k = 1$ forks can occur on a given block (k-fork coherence), thus the consortium blockchain implements the blocktree strong consistency.

Degraded case ($t_0 < t < (n - t_0)$). In this case the ever-growing tree is violated but not the eventual prefix property. Moreover, if the threat of punishment (allowed by the accountability) disincentivizes a malicious coalition from attacking, then the strong prefix property is ensured. Let $t_0 < t < n - t_0$. A malicious coalition can do either one of these actions:

- Follow the protocol.
- Stay mute to violate the liveness property of the consensus and so the ever growing tree property of the blockchain (and so the (even eventual) consistency of the blockchain).
- Attempt an attack to create a disagreement among the consortium. Whatever the result of the bid, a proof of guilt will eventually be spread among all the honest processes (consortium members and clients). Then, regardless the sentence applied to the malicious processes, a special fork can be created to drop the illegitimate forks (labelled in consequence) due to the attack. The selection function returns then the unique blockchain which does not pass by a fork labelled illegitimate. The eventual prefix property is then satisfied. Moreover, if the potential punishment discourages any attempt of disagreement-attack (for example with a negative utility function in game theory approach), then the strong prefix property is ensured.

So, as long $t < (n - t_0)$, the eventual prefix property is ensured, but the ever-growing tree property is violated if $t > t_0$.

Zombie case ($(n - t_0) \leq t \leq n$). In this case a super coalition of $t > 2n/3$ Byzantine processes can override the General Polygraph Protocol by proposing directly two conflicting views to two different clients to then perform a double-spending attack. The coalition does not participate to the consensus in order to violate the liveness property. It follows that the ever growing tree property is violated. Note that safety is also violated: When a client invokes the read() primitive, the coalition can answer arbitrary values, despite the non-termination of the legitimate consensus. The client is supposed to trust the coalition, like all the other clients who can forever receive a different output for the read() primitive. Hence, for $t \geq n - t_0$, the eventual prefix property is violated. This makes the blockchain vulnerable to a double-spending attack.

F PBFT, TENDERMINT AND HOTSTUFF DO NOT SOLVE THE ACCOUNTABLE BYZANTINE AGREEMENT

We now propose an extension that applies to a class of classic Byzantine fault tolerant consensus, that, despite being intuitive, cannot make these algorithms accountable. This algorithms corresponds to variants of PBFT [11] in that they inherits the leader, the views as well as preparation and decision values exchanged across view changes. In particular, we show that the result applies at least to PBFT [11], Tendermint [4] and HotStuff [35].

Algorithms. Each process i is assigned an algorithm $\mathcal{A}_i = (M_i, TI_i, TO_i, X_i, \Sigma_i, \sigma_i^0, \alpha_i)$, where M_i is the set of messages i can send or receive, TI_i is a set of terminal inputs i can receive, TO_i is a set of terminal outputs i can produce, X_i is a set of variables, Σ_i is a set of states, based on an evaluation of the variables, $\sigma_i^0 \in \Sigma_i$ is the initial state, and $\alpha_i : \Sigma_i \times \mathcal{P}(M_i \cup TI_i) \rightarrow \Sigma_i \times \mathcal{P}(M_i \cup TO_i)$ maps a set of inputs and the current state to a set of outputs and the new state. Here, $\mathcal{P}(X)$ denotes the power set of X. For convenience, we define $\alpha(\sigma, \emptyset) := (\sigma, \emptyset)$ for all $\sigma \in \Sigma_i$.

Distributed algorithms. A distributed algorithm is a tuple $(A_1, ..., A_n)$, one algorithm per process, such that $M_i = M_j$ for all i, j. (The message does not need to contain the sender or the receiver identifier.) When we say that an execution e is an execution of a distributed algorithm A, this implies that each process i is considered correct or faulty in e with respect to the algorithm A_i it has been assigned. We write $corr(A, e)$ to denote the set of processes that are correct in e with respect to A. We note log_i the variable representing the set of messages received by i.

Execution. We define an *event* as a tuple (p, I, O), where p is a process on which the event occurs, I represents a set of received messages and observed internal events by p and O represents a set of sent messages and internal events p produces.

A *behavior fragment* β_p is a sequence of events $(p, I_1, O_1), (p, I_2, O_2), ...$ We say that $\beta_p = (p, I_1, O_1), (p, I_2, O_2), ...$ is *valid* if and only if it conforms to a protocol Π_p, i.e., there exists a sequence of states $s_0, s_1, ...$ in Σ_p such that for all $i \geq 1$, $\alpha_p(s_{i-1}, I_i) = (s_i, O_i)$.

A *behavior* β_p is a behavior fragment so that $s_0 = \sigma_p^0$.

An *execution fragment* e is a sequence of events. For every process p, we note $\beta_p = e|_p$ the behavior fragment obtained after projection on the process p. We say that an execution e is valid if and only if for every process p, the behavior fragment $\beta_p = e|_p$ is valid.

An *execution* is an execution fragment so that for every process p, the behavior fragment $\beta_p = e|_p$ is a behavior.

Class of PBFT-like algorithms. We define a class $\mathcal{L}_0$ of algorithms that correspond to the leader-based consensus algorithms derived from PBFT, including Tendermint and HotStuff. They proceed in views, that can be triggered after a timeout, and whose leader, chosen in a round-robin fashion, sends a proposal (that we call here a *suggestion*). Two variables, preparation and decisions are key to the protocols and dictate what value can be proposed by the leader in a new view. Only when messages from sufficiently many distinct processes are collected, can the preparation be changed. This class $\mathcal{L}_0$ allows us to generalize the same result to other algorithms by simply showing that the algorithms belong to $\mathcal{L}_0$.

Definition F.1 (class $\mathcal{L}_0$). A t_0-resilient asynchronous Byzantine consensus algorithm is in class $\mathcal{L}_0$ if it verifies the following properties:

(1) There is a round-robin rotation of leader that can propose a suggestion at each view: $\forall v \in \mathbb{N}$, $\ell_v = p_x \in \Psi$ with $x = v \ mod \ n$.
(2) Each process stores two variables preparation and decision. These variables comprise a value s (suggested by the leader of the last update), an integer representing the view number of the last update and a set of messages justifying the update. $\forall i \in \Psi$ $preparation_i, decision_i \in X_i$, $type(preparation_i) = type(decision_i) = Val \cup \{\bot\} \times \mathbb{N} \times \mathcal{P}(M)$. Initially, $\sigma_i^0.preparation_i.val = \sigma_i^0.decision_i.val = \bot$. The decision variable can be updated only once: a decision is irrevocable.
(3) Variables preparation and decision can be assigned in a view v (variable decision is assigned only once) if enough messages are collected from $n - t_0$ distinct processes in this same view v. $\exists J \in \mathcal{P}(M)$ where $\forall m \in J, m.view = v$, $\sigma_i \in \Sigma_i$ where $\sigma_i.preparation_i.view = v'$, $\sigma_i' \in \Sigma_i$ where $\sigma_i.preparation_i.view = v$ such that $\alpha_i(\sigma_i, J) = (\sigma_i', .)$.
(4) The variable decision in a view v implies preparation in the same view v. If it exists $v \in \mathbb{N}, s \in Val, J \in \mathcal{P}(M) \cap log_i$ s. t. $decision_i = (s, v, J)$ then $\exists J' \in \mathcal{P}(M) \cap log_i$ $decision_i = (s, v, J')$.

(5) In a fully correct execution, a decision implies the preparation of the same value by $n - t_0$ other processes and the reception of as many messages that claim this fact. If it exists $v \in \mathbb{N}, s \in Val, J^d \in \mathcal{P}(M) \cap log_i$ s. t. $decision_i = (s, v, J^d)$, then J^d stores a set $\underline{m}$ of set of messages $(\underline{m}_k)_{k\in\Phi}$ ($\Phi \subset \Psi$, $p_k \in \Psi$, $|\Phi| \geq n - t_0$), s. t. ($\underline{m}_k$ sent in an execution e by p_k and p_k correct) implies (it exists $J_k^p \in \mathcal{P}(M)$ and a state x_k of the process k in the execution e, s. t. $x_k.preparations_k = (s, v, J_k^p)$).

(6) After a timer expires for view v, a process sends a new-view message to the leader ℓ_{v+1} of the next view $v + 1$, containing its preparation and potential additional collected messages to justify its preparation. $\forall v \in \mathbb{N}$, $\tau_i^v =$ timer of view v expired $\in TI_i$, $\alpha(\sigma_i, \tau_i^v) = (\sigma_i', \{nv_i^v\})$ where $nv_i^v = (NV, i, v, (s_i^p, v_i^p, J_i^p))$ with $dest(nv_i^v) = \ell_{v+1}$ and $preparation = (s_i^p, v_i^p, J_i'^p)$.

(7) A suggestion s^v for a view v can only be proposed by the leader (7.1).
Additionnaly (the case of HotStuff and PBFT) this suggestion can be motivated by $n - t_0$ new-view messages sent in the previous view $v - 1$ (7.2), the suggestion will be the value of the propagated preparation prepared at the highest view number (7.3). The collected new-view messages will allow to build a justification for the computed suggestion (7.4). $\alpha_i(\sigma_i, I) = (\sigma_i', O)$ with O containing a suggestion $sugg(s^v, v, J^{s,v})$ implies (a) i is leader of view v and (b) $\sigma_i.log_i \cup I$ contains a set of at least $n - t_0$ new-view messages $(nv_k^{v-1})_{k\in\Phi}$ ($\Phi \subset \Psi$, $p_k \in \Psi$, $|\Phi| \geq n - t_0$) s. t. $\forall k \in \Phi$ $(nv_k^{v-1}).view = v - 1$ (3) $nv_h^{v-1} argmax\{nv_k^{v-1}.v_k^p\}$ and $s = nv_h^{v-1}.s^p$ (4) $J^{s,v} \subset \bigcup_{k\in\Phi}(nv_k^{v-1}.J^p \cup nv_k^{v-1})$.

(8) A preparation of a value s at view v implies the reception of a suggest message from the leader ℓ_v at this view v containing the suggestion s, potentially with additional received messages to justify s so that s verifies a predicate $Safe$: If it exists $v \in \mathbb{N}, s \in Val, J \in \mathcal{P}(M) \cap log_i$ s. t. $preparation_i = (s, v, J)$ then $\exists J^{s,v} \in \mathcal{P}(M) \cap log_i$, $Sugg = sugg(s, v, J^{s,v}) \in log_i$ s. t. a predicate $Safe(Sugg)$ indicates the validity of the $Sugg$ messages in the corresponding algorithm.

Next, we explain why HotStuff [35], PBFT [11] and Tendermint [4] are part of this class $\mathcal{L}_0$. For simplicity, we consider the version of HotStuff without threshold signatures as evaluated in [35]. Note that as we mentioned before in Section 4.2, threshold signatures are insufficient for accountability and thus cannot help make HotStuff, PBFT or Tendermint accountable.

Lemma F.2. *HotStuff, Tendermint and PBFT are in $\mathcal{L}_0$.*

Proof. The proof consists of showing by examination of the code of HotStuff, PBFT and Tendermint that they verifiy the 8 properties of Def. F.1.

(1) There is a unique process per view, called here a *leader*, that is picked in a round robin fashion across subsequent views. This unique process is called leader in HotStuff, the proposer that changes at each "epoch" in Tendermint and the primary in PBFT.

(2) There are two variables we refer to as *preparation* and *decision*. These variables are called, respectively, prepareQC and commitQC in HotStuff, validValue and decision in Tendermint and prepare and commit in PBFT.

(3) Let $J^{p,v}$ and $J^{d,v}$ be the justifications for preparation p and view v, and decision d and view v, respectively.
   - HotStuff: $m : matchingMsg(m, prepare, v)$, $m' : matchingQC(m'.justify, prepare, v)$ and $m'' : matchingQC(m''.justify, commit, v)$
   - Tendermint:
     - Preparation: Line 22: if $\exists v' : 2f + 1\langle PROPOSE, h, e_i, v'\rangle \wedge valid(v') \wedge sendByProposer(h, e_i, v')$ then Line 23: $validValue_i \leftarrow v'$;
     - Decision: Line 24: if $\exists v_d, e_d : 2f + 1\langle VOTE, h, e_d, v_d\rangle \wedge valid(vd) \wedge decision_i = nil$ then line 25: $decision_i \leftarrow v_d$.
   - PBFT: a new-view and a pre-prepare from the primary and $n - t_0$ prepare messages.

(4) Decision implies preparation. (FIFO is implicitly implemented, s. t. m' is always sent after m means m' can be delivered only if m has already been delivered).
   - HotStuff: $broadcastMsg(decide, \perp, commitQC)$ comes after $broadcast\ Msg(pre\text{-}commit, \perp, prepareQC)$;
   - Tendermint: At line 18, $broadcast\ \langle VOTE, h, e_i, vote_i\rangle$ comes after line 4: $broadcast\ \langle PROPOSE, h, e_i, proposal_i\rangle$;
   - PBFT: a commit message is sent after a prepare message.

(5) Decision implies preparation of $(n - t_0)$ others.
   - HotStuff: To build a precommitQC, the leader need a threshold signature built from $n - t_0$ $voteMsg(pre\text{-}commit, m.justify.process, \perp)$ that comes after $prepareQC \leftarrow m.justify$. Then $broadcast\ Msg(decide, \perp, commitQC)$ comes after $broadcast\ Msg(commit, \perp, precommitQC)$
   - Tendermint: A vote message is triggered by the reception of $n - t_0$ valid propose messages. At line 8: if $\exists v : 2f + 1\ \langle PROPOSE, h, e_i, v\rangle \wedge valid(v) \wedge sendByProposer(h, e_i, v)$ then at sline 11: $vote_i \leftarrow v$ and at line 17: if $vote_i \neq nil$ then at line 18: broadcast $\langle VOTE, h, e_i, vote_i\rangle$.

   - PBFT: A commit message is triggered by the reception of $n-t_0$ prepare-messages that lead to the preparation.
(6) Regarding the new-view message:
   - HotStuff: $send\quad Msg(new\text{-}view, \perp, prepareQC)$ to $leader(v+1)$.
   - Tendermint: at line 14, $\forall v, p_j : (\langle \text{VOTE}, h, e_i, v\rangle, p_j) \in messageSet_i$, broadcast$\langle \text{VOTE}, h, e_i, v\rangle$
   - PBFT: a view-change message containing a set P that contains a set P_m for each message m prepared, where each P_m contains $n-t_0$ prepare messages for m.
(7) Regarding the suggestion message, we describe the point of view of the leader as follows:
   - In HotStuff, during the prepare phase, for a process that acts as the leader we have: Property (7.2) holds with the line: 'wait for $(n-f)$ new-view messages: $M \longleftarrow \{m \mid matchingMsg(m, new\text{-}view, v-1)\}$'. Properties (7.3) and (7.4) hold with the line: '$highQC \longleftarrow (\underset{argmax}{m} \in M\{m.justify.viewNumber\}).\ justify$'.
   Property (7.1) holds with the lines : '$curProposal \longleftarrow createLeaf(highQC.process, client's\ command)$ $broadcast\ Msg(prepare, curProposal, highQC)$'
   - In Tendermint, Property (7.1) holds with the lines 16 and 17: At line 16, if $proposer(h, e_i) = p_i$ then, at line 17, $broadcast\ \langle PRE\text{-}PROPOSE, h, e_i, proposal_i, validEpoch_i\rangle_i$. The variable $proposal_i$ is updated either a) at line 28 of PRE-PROPOSE round: $proposal_i \longleftarrow v_i$ or b) at line 41 of VOTE round : $proposal_i \longleftarrow validValue_i$. In case a) $proposal_i$ verifies conditions of line 27: $2f+1$ $\langle PROPOSE, h, validEpoch_j, v_i\rangle_i \wedge validEpoch_j \geq lockedEpoch_i \wedge validEpoch_j < e_i \wedge valid(v_i)$. In case b) the last update of $validValue_i$ corresponds to the last preparation.
   - In PBFT, a new-view messages containing V containing $n-t_0$ well-written view-change messages, O containing the pre-prepare messages according to V.
(8) Regarding the acceptation of a suggestion, we describe the point of view of a replica that accepts a suggestion only if it satisfies some properties chosen to ensure the suggestion is legitimate.
   - In HotStuff, during the prepare phase, a replica waits for message m from $leader(v)$, $m : matchingMsg(m, prepare, v)$. If $m.process$ extends from $m.justify.process$ and $safeprocess(m.process, m.justify)$ then $send\ voteMsg(prepare, m.process, \perp)$ to $leader(v)$.
   - In Tendermint, the condition $\exists v_j, e_j : sendByProposer(h, e_i, v_j, e_j)$ at line 21 and the condition $validEpoch_j \geq lockedEpoch_i \wedge validEpoch_j < e_i$ at line 27 have to be respected to take into account the suggestion v_j.
   - In PBFT, a replica check the well-formedness of a new-view message.

□

The next definitions are crucial to determine whether an algorithm is accountable. We first define "justification chain" as an alternating sequence of sets of messages that transitively justify a proposal sent by a leader, before defining the "cautiousness" and the "recklessness" properties for algorithm in $\mathcal{L}_0$.

Definition F.3 (Justification chain of any depth d). Let $A \in \mathcal{L}_0$, $k, d \in \mathbb{N}$, $k > d$ and $Sugg^k$ a suggestion of view k.

A *justification chain* for $Sugg^k$ of depth d is an alternating sequence $(\underline{NV}^{k-d}, Sugg^{k-(d-1)}, \underline{NV}^{k-(d-1)}, Sugg^{k-(d-2)},...,\underline{NV}^{k-1})$ of sets of messages $\underline{NV}^h \in \mathcal{P}(M), h \in [k-d, k-1]$ and $Sugg^h \in M, h \in [k-(d-1), k-1]$ where each $\underline{NV}^h$ is sent by a set of processes ϕ^h where $\phi^h = \{i_{h,1}, ..., i_{h,n-t_0}\}$ and each $Sugg^h$ is sent by ℓ_h so that all these messages can be explained by a fully-correct execution. More formally, there exists a valid reachable execution fragment $e \in E$, and e' a sub-execution of e, where $e' = (i_{k-d,1}, I_{k-d,1}, O_{k-d,1}), (i_{k-d,2}, I_{k-d,2}, O_{k-d,2}),$ ... $(i_{k-d,n-t_0}, I_{k-d,n-t_0}, O_{k-d,n-t_0})$ $(\ell_{k-(d-1)}, I_{k-(d-1)}, O_{k-(d-1)})$ ...$(i_{k-1,1}, I_{k-1,1}, O_{k-1,1}), (i_{k-1,2}, I_{k-1,2}, O_{k-1,2})$, ... $(i_{k-1,n-t_0}, I_{k-1,n-t_0}, O_{k-1,n-t_0})$ (ℓ_k, I_k, O_k) where these three properties are all satisfied:

(1) $\forall h \in [k-(d-1), k], Sugg^h \in O_h$,
(2) $\forall h \in [k-d, k-1], \underline{NV}^h \subset \bigcup_{x \in [1,n-t_0]} O_{h,x}$,
(3) $\forall h \in [k-d, k-1], \underline{NV}^h \subset I_{h+1}$ (the new-view messages trigger the suggestion (that could be never delivered by any process if the timer expires too soon) and
(4) $\forall h \in [k-d, k-1]$, if it exists $i_x \in \phi^h, nv^h_x \in \underline{NV}^h \cap O_{h,x}$ s. t. $nv^h_x.v^p = h$ and $nv^h_x.s^p = s$, then $Sugg^h = s$. (A preparation update is motivated by a new well-formed suggestion)(Nothing prevents the leader from having a very slow connection, s. t. nv^h_x is not motivated by $Sugg^h$.)

We say that a justification chain for a suggestion $Sugg^k$ of view k is *complete* if its depth is k.

We say that a justification chain is *extractable* from a set of messages $log \in \mathcal{P}(M)$ if it exists an algorithm that take log as input and outputs such a justification chain with a probability 1 (with a complexity polynomial with the security number κ).

A complete justification chain is a necessary piece of information to collect before decision to ensure accountability. If a "hole" appears in the chain, we call it a partial justification chain and we will show that this is not enough to ensure accountability.

Definition F.4 (Partial justification chain of any depth d). Let $A \in \mathcal{L}_0, k, d, q \in \mathbb{N} k > d > q$ and $Sugg^k$ a suggestion of view k. A partial justification chain for $Sugg^k$ of depth d is an alternating sequence of sets of messages that mimics a justification chain, notwithstanding the fact that it exists at least an integer $h' < k$ where the property (3) is not respected, that is, we do not know the new-view messasges that motivated $Sugg^{h'+1}$ in the chain. Since a set of new-view messages at view h' is missing, we say that such a partial justification chain contains a "hole at view h'".

If a partial justification chain pjc contains a hole at view h^f and for every other contained hole at another view h', we have $h^f > h'$, we say that h^f is the view of the *first* hole of pjc.

A partial justification chain pjc of a set J of different partial justifications for the same value suggested at the same view is *maximal* in J, if $\forall pjc' \in J$, the view h' of the first hole of pjc' verifies $h' < h$ where h is the view of the first hole of pjc.

We are ready to define two properties: recklessness and cautiousness. These properties are about the pieces of information that are collected before decision. For a t_0-resilient asynchronous Byzantine consensus algorithm, the required information before decision is big enough to ensure safety and small enough to ensure liveness. We have the same approach for accountability: we need enough information to ensure accountability in case of disagreement and not too much to continue to ensure liveness as long as $t < t_0$. For an algorithm A in $\mathcal{L}_0$ the necessary or sufficient pieces of information to collect before a decision at view v to ensure accountability can be reduced into a sufficiently deep justification chain.

If a complete jsutification chain is stored in the log of each correct process before decision, then we say that A ensures cautiousness and we will show (Theorem F.9) that it is enough to ensure accountability.

Definition F.5 (Cautiousness). Let $A \in \mathcal{L}_0$. The cautiousness property ensures that : $\forall e \in execs(A)\ \forall v_j \in \mathbb{N}\ \forall \sigma_j \in \Sigma_j$ s.t. $\sigma_j.decision_j = (s_j, v_j, J_j^d)$ then $\forall \sigma'_j$ following σ_j in e, $\sigma'_j.log_j$ contains a complete justification chain for s_j of depth $v_j - 1$.

However, if A allowed a correct process to decide at view v without an extractable sufficiently deep justification chain (but only a partial justification chain that contains a hole at view $h' \in [v - t_0 + 1, v]$), we say that A verifies recklessness and we will show (Theorem F.8) this is enough to build executions where accountability is impossible.

Definition F.6 (Recklessness). Let $A \in \mathcal{L}_0$. The recklessness property ensures that it exists $v' \in \mathbb{N}$, $\forall v_j \in \mathbb{N}, v_j > v'$, we can build an execution $e \in execs(A)$ where process j reaches a state σ_j and decides $s_j \in Val$ at view v_j without a sufficiently deep justification chain for s_j. More formally:

- $\exists s_j \in Val, \sigma_j.decision_j \in \{s_j\} \times \{v_j\} \times \mathcal{P}(\mathcal{M})$ and
- no justification chain with depth greater than $t_0 - 1$ can be extracted from $\sigma_j.log_j$, that is the maximal partial justification chain that can be extracted from log_j contains at least a hole in a view $v - t_0 \leq h < v$.

Finally, we will show (Theorem F.11) that for every algorithm $A \in \mathcal{L}_0$ that verifies recklessness, an extension $\bar{A}$ with bounded justification per message still verifies recklessness and by implication is still not accountable .

Definition F.7 (Class $\mathcal{L}$ and $\mathcal{L}'$). We define $\mathcal{L}$ the class of algorithm in $\mathcal{L}_0$ that verifies recklessness and $\mathcal{L}'$ the class of algorithm in $\mathcal{L}_0$ that ensures cautiousness.

Theorem F.8. *If an algorithm A is in $\mathcal{L}$, then it is not accountable.*

Intuition. We design three executions e^0, e^1, e^2 so that (1) $e^1 \overset{i,j}{\sim} e^2$ and (2) $e^0 \overset{j}{\sim} e^1 \overset{j}{\sim} e^2$. In execution e^1 and e^2, process i decides s_i at view v_i while process j decides s_j at view $v_j >> v_i$ where s_i and s_j are conflicting, which leads to a disagreement. To allow the decision of i, a quorum Q_i prepared s_i at view v_i. Later, a set ϕ of some processes send a set $\underline{NV}^{v_z - 1}$ of new-view messages to the leader ℓ_{v_z} of view v_z to convince it to suggest s_j. The processes in $B = \phi \cap Q_i$ with $|B| \geq t_0 + 1$ are guilty since they did not propagate their preparation of s_i. These commission faults are sufficient to reach disagreement. Later, when i and j detect the disagreement, they are not able to identify the sender ϕ of $\underline{NV}^{v_z - 1}$, s. t. the executions e^1 and e^2, with different Byzantine processes, will be indistinguishable for i and j (1).

The question then becomes whether it is possible for process j to suspend its judgment, waiting for additional pieces of information to collect before its decision to avoid such a situation. The answer is in the negative. We prove it by constructing an execution e^0 where the number of Byzantine processes is lower than t_0 and process j has to make progress without being able to expect additional messages since the algorithm has to ensure liveness when $t < t_0$ (2). To do so, we construct a chain χ where each process p that is a leader of a view in χ (we say $p \in P$) seems particularly slow to process j. Since P could be Byzantine, process j, has to decide without expecting messages from P. The set P will participate to the new preparation of s_j, which would be impossible if j knew that i already decided s_i before its own decision. After disagreement, P will stay mute and will not reveal the crucial information $\underline{NV}^{v_z-1}$ that would allow i and j to determine who are the Byzantine processes. □

Proof. Let assume $A \in \mathcal{L}_0$ and verifies recklessness and $v_j > t_0$.

We fix $k \in [1, t_0]$. The reader can fix $k = 1$ for this proof, but we will allow greater values of k for Theorem F.11).

We decompose Ψ in the partition $\{\{i, j\}, W_1, W_2, W_3, W_4, \{\omega_5\}, P\}$ where:

- $|W_1| = |W_2| = t_0 - (k + 1)$,
- $|W_3| = t_0$,
- $|W_4| = |P| = k$,
- $|\{\omega_5\}| = |\{i\}| = |\{j\}| = 1$,
- ω_5 is clearly Byzantine,
- $Q_i = \Psi \setminus W_3$, $|Q_i| = n - t_0$,
- $K = \Psi \setminus (W_2 \cup P \cup \{i\}) = K^{s_i} \cup K^{s_j}$, $|K| = n - t_0$ and
- $K^{s_i} = W_1 \cup W_4 \cup \{j\}$, $K^{s_j} = W_3 \cup \{\omega_5\}$, $|K^{s_i}| = t_0$, $|K^{s_j}| = t_0 + 1$.

We build a first "partial" execution $\tilde{e}$, as follows:

We fix $2 \leq v_i < v_j - 1$, so that a process j can decide s_j at view v_j without requesting a partial justification for s_j with depth $(v_j - v_i)$ (there is a hole at view v_i).

At view $v_i - 1$, ℓ_{v_i-1} legitimately suggests s and every process prepares s without deciding it.

At view v_i, ℓ_{v_i} legitimately suggests s_i (compatible with s) and every process in K^{s_i} prepares s_i without deciding it, while the ones in K^{s_j} do not prepare s_i.

At view v_j, every process in K prepares s_j (compatible with s but not with s_i) and decides s_j.

We note v_y the first view between v_i and v_j where K^{s_i} prepares s_j.

We build a chain $\chi = [v_{z_1}, ..., v_{z_k}]$ between v_i and $v_y - 1$ so that (1) $\forall h \in [1, k]$, $\ell_{v_{z_h}}$ suggests s_j as ℓ_{v_y} (2) $v_{z_k} = v_y - 1$ and (3) $\forall h \in [1, k-1], v_{z_{h+1}} = v_{z_h} + 1$.

The set $P = \bigcup_{h \in [1,k]} \ell_{v_{z_h}}$, while $\ell_{v_i}, \ell_{v_y} \in W_3$.

At each view $v_{z_h} \in \chi$, $W_2 \cup P \cup \{\omega_5\}$ updates its preparation ($\forall q \in W_2, preparation_q = (s_j, v_{z_h}, J^{p, v_{z_h}})$), sends a new-view message to $\ell_{v_{h+1}}$ and also sends a new-view message to ℓ_{v_y}.

Thus $\forall v \in [v_{z_1}, v_{z_k}]$ ℓ_v received enough new-view messages from $K^{s_j} \cup W_2 \cup P \cup \{\omega_5\}$ to justify its suggestion of s_j, which finally justifies the update of preparation of K^{s_i} into s_j at view v_y.

Now, we describe three executions $e^0_\chi, e^1_\chi, e^2_\chi$ that share the same common structure $\tilde{e}$ that we presented just before.

In e^0_χ, the messages from $W_2 \cup P$ to K are slow. Since $|W_2 \cup P| \leq t_0$, j has to decide without expecting additional messages, because j cannot distingush e^0_χ with another execution where $W_2 \cup P$ is Byzantine and refuse to communicate with K.

In e^1_χ, processes in $B_1 = W_1 \cup P \cup \{\omega_5\}$ are Byzantine. They prepared s_i from v_i until $v_{z_1} - 1$. At v_i, they convinced (with Q_i) process i to decide s_i. At $v_{z_1} - 1$, Byzantine processes in B_1 pretended they did not prepare s_i, but only s_j and convinced (with K^{s_j}) $\ell_{v_{z_1}}$ to suggest s_j.

In e^2_χ, $B_2 = W_2 \cup W_4 \cup \{\omega_5\}$ are Byzantine. They prepared s_i from v_i until $v_{z_1} - 1$. At v_i, they convinced (with Q_i) process i to decide s_i. At $v_{z_1} - 1$, Byzantine processes in B_2 pretended they did not prepare s_i, but only s_j and convinced (with K^{s_j}) $\ell_{v_{z_1}}$ to suggest s_j.

(Let us note that we could also construct e^3_χ with $B_3 = W_1 \cup W_4 \cup \{\omega_5\}$ or e^4_χ with $B_4 = W_2 \cup P \cup \{\omega_5\}$.)

We have $e^0_\chi \overset{j}{\sim} e^1_\chi \overset{j}{\sim} e^2_\chi$.

In each execution, because of recklessness, the correct processes i and j never know the new-view messages sent to $\ell_{v_{z_1}}$ to justify the suggestion s_j at view v_{z_1}, s. t. $e^1_\chi \overset{i,j}{\sim} e^2_\chi$ (as long as the processes in $\Gamma^{v_{z_1}}_q = B_q \cup W_3 \cup P$ (with $|\Gamma^{v_{z_1}}_q| > n - t_0$), which helped to make prepare s_j at view v_{z_1} stays mute, which is a key difference with [33] that assume $t_0 + 1$ processes are correct and will reveal the key information to determine the Byzantine processes that did not propagate their preparation).

We note $Attack(v_i, s_i, v_j, s_j, \chi)$ the triplet $(e^0_\chi, e^1_\chi, e^2_\chi)$.

The construction stays possible as long as j does not store a complete justification chain before its decision. Indeed if $\underline{NV}^h$ triggering $Sugg^{h+1}$ at view h is missing in log_j, then we chan build $Attack(v_i, s_i, v_j, s_j, \chi)$ with χ that starts at $v_{z_1} = h + 1$.

To summarize, there exists:

$e^0_\chi \in execs(A)$ s. t. $|corr(e^0_\chi, A)| = n - t_0$ where j decides s_j;

$e^1_\chi \in execs(A)$ s. t. $|corr(e^1_\chi, A)| < n - t_0$ where j decides s_j, i decides s_i conflicting wit s_j, W_2 is correct and W_3 is correct (but slow);

$e^2_\chi \in execs(A)$ s. t. $|corr(e^2_\chi, A)| < n - t_0$ where j decides s_j, i decides s_i conflicting wit s_j, W_1 is correct and W_3 is correct (but slow);

so that $e^0_\chi \overset{j}{\sim} e^1_\chi \overset{j}{\sim} e^2_\chi$ with $|corr(e^0_\chi, A)| = n - t_0$ and $e^1_\chi \overset{i,j}{\sim} e^2_\chi$ as long as processes in $\Gamma^{v_{z_1}}_q = B_q \cup W_3 \cup P$ are mute.

Thus in both e^1_χ and e^2_χ, there is a disagreement where we cannot ensure detection of a process, excepting ω_5.

Finally, A is not accountable. □

Theorem F.9. *An algorithm in $\mathcal{L}'$ is accountable.*

Proof. Before deciding s_i at view v_i process i received J^{d,v_i} at view v_i. Because of Property (5) of $\mathcal{L}_0$, Q_i prepared s_i at view v_i. Let v_z be the first view greater than v_i where a process q prepared s_q conflicting with s_i. Because of Properties (6), (7) and (8) of $\mathcal{L}_0$, s_q is justified with $\underline{NV}^{v_z-1}$ that contains $n - t_0$ new-view messages from ϕ^{v_z-1}, which propagated s_q. All processes in $Q_i \cap \phi^{v_z-1}$ are guilty since they did not propagate a correct value. Finally this culpability is proved thanks to the justification chain that allows to determine v_z and ϕ^{v_z-1}. □

Extension. In this paragraph, we propose an "intuitive" extension $\bar{A}$ of $A \in \mathcal{L}$. We aim to show that it is not enough to trivially add some justifications in a message to ensure accountability. Hower, this does not mean that no cheaper transformation exists to ensure accountability.

Definition F.10 (t_0-bounded extension). Let $d \in [1, t_0 - 1]$. We define τ_d the function that maps each algorithm $A \in \mathcal{L}_0$ into an algorithm $\bar{A}$, s. t.

- each suggestion $Sugg^v$ becomes $\bar{Sugg}^v = (Sugg^v, jc)$ where jc is a justification chain for $Sugg^v$ suggested at view v with depth $max(d, v)$.
- each new-view message $nv_i^v = (NV, i, v, s_i^p, v_i^p, J_i^p)$ becomes $\bar{nv}_i^v = (nv_i^v, jc)$ where jc is a justification chain for s_i^p suggested at view v_i^p with depth $max(d, v_i^p)$.

In the remainder, we simply refer to t_0-bounded extension as extension when it is clear from the context.

Theorem F.11 (Escape from $\mathcal{L}$). *Let $A \in \mathcal{L}$ and $d \in [1, t_0 - 1]$. Let $\bar{A} = \tau_d(A)$ be an extension of A with depth of justification d bounded by $t_0 - 1$. $\bar{A} \in \mathcal{L}$, which means $\bar{A}$ is not accountable.*

Proof. The proof is the same as the one for Theorem F.8 and we chose $k \geq d$. We fix $v_y > t_0, v_j > v_y, v_i = 2, v_{z_1} = v_y - k$. We fix $\chi = [v_{z_1}, v_y - 1]$. We fix $s, s_i, s_j \in Val$ so that s_i and s_j are not compatible while they are both compatible with s. We note $(e_\chi^0, e_\chi^1, e_\chi^2) = Attack(v_i, s_i, v_j, s_j, \chi)$. We note $(\bar{e}_\chi^0, \bar{e}_\chi^1, \bar{e}_\chi^2)$ the corresponding extended executions of $\bar{A}$. We want to show that 1) $\bar{e}_\chi^0 \overset{j}{\sim} \bar{e}_\chi^1 \overset{j}{\sim} \bar{e}_\chi^2$ to enforce the decision and 2) $\bar{e}_\chi^1 \overset{i,j}{\sim} \bar{e}_\chi^2$ to avoid accountability.

(1) With asynchrony, there is no way to distinguish e_χ^0 where $W_2 \cup P \cup \{i\}$ are very slow from another execution where $W_2 \cup P\{i\}$ are Byzantine and refuse to communicate. Thus is possible that no process in K receive any message in views in χ before decision. This claims stays true for $\bar{e}_\chi^0$. Thus j has to decide after $\bar{e}_\chi^0$.

(2) The difference between e_χ^1 and e_χ^2 is the set of processes W_1 or W_2 that did not propagate correctly its preparation to the leader $\ell_{v_{z_1}}$. Because the depth of the justification is bounded, the suggestion $\bar{Sugg}^{v_j} = (Sugg^{v_j}, jc)$ holds a justification chain that can end at view v_{z_1}, but no more. Thus $\bar{A}$ verifies recklessness too.

Theorem F.8 applies here: because i and j do not know the new-view messages that motivated the suggestion s_j of leader $\ell_{v_{z_1}}$, i and j cannot distinguish e_χ^1 and e_χ^2 after disagreement. Thus $\bar{A}$ is not accountable.

Let's note that after disagreement the set Γ of other processes that could be able to distinguish e_χ^1 and e_χ^2 crash. Then process i and j cannot say if Γ is slow or crashed. □

Theorem F.12 (Theorem 4.3). *HotStuff, PBFT and Tendermint as well as all their t_0-bounded extensions are not accountable.*

Proof. Let A be an algorithm among HotStuff, PBFT and Tendermint. At first, Lemma F.2 claims this algorithm is in $\mathcal{L}_0$. Then recklessness is verified for all of these algorithms since a justification for a preparation or suggestion never reaches a depth of t_0, so $A \in \mathcal{L}$. Let $\bar{A}$ be an extension of A with justification depth bounded by t_0. Because of Theorem F.11, recklessness continues to be verified s. t. $\bar{A} \in \mathcal{L}$.

Finally, because of Theorem F.8, $\bar{A}$ is not accountable, which finishes the proof. □

Remark 1. *We stress that this theorem does not claim that there is no cheap operation to transform an algorithm in $\mathcal{L}$ (like HotStuff, PBFT or Tendermint) into one in $\mathcal{L}'$ where accountability would be ensured. This theorem only shows why a naive approach would fail. Also, if a little change in these algorithms makes the previous attack not relevant anymore, that does not mean there is not another attack that could succeed.*

Remark 2. *This attack allows to understand why we have chosen to extend a "leaderless" algorithm like DBFT. Unlike a leader-based algorithm, such a kind of algorithm has the property P that if $t \leq t_0$ and a correct process i decides at round r_i, then every other correct process j eventually decides in a round $r_j \leq r_i + 2$. The idea is that we need a justification depth of only 2 to reveal why P did not hold in case of disagreement.*