Polygraph: Accountable Byzantine Agreement

Pierre Civit
University of Sydney

Seth Gilbert
National University of Singapore

Vincent Gramoli
University of Sydney
Data61-CSIRO

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 ($f \geq n/3$) 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. We show that a commonly used state-of-the-art Byzantine fault tolerance consensus algorithm cannot be made accountable without storing and exchanging extra logs of at least $\Omega(n)$ rounds. By contrast, each round of Polygraph exchanges only $O(n^3)$ bits and $O(n^2)$ messages. Finally, we use a blockchain application to evaluate Polygraph on a geodistributed system of 80 machines to commit ten thousand of transactions per second.

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. The (virtual) gold rush is on, and as in the Wild West of yore, the outlaws are ever present. Byzantine agreement protocols act as the locks on the bank doors, preventing the gangs from making off with the loot.

Limitations. 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.[1] And yet, as mining pools become larger and more centralized, we are in danger of a single group's power surpassing the $1/3$ threshold. (For example, the largest Bitcoin mining pool today controls approximately 19% of the hashing power.) This problem is even more severe in consortium and private blockchains where a small number of predetermined parties maintain control over the chain, and hence are each more likely to control a significant fraction of the compute power of the network.

At first, one might hope to relax the liveness guarantees, while always ensuring safety. For example, perhaps we could ensure that on termination, the honest users would always agree; if the dishonest users have too much power, then the protocol never terminates. Alas, in a partially synchronous network, this type of guarantee is impossible (see Theorem 1). If the adversary controls more than $1/3$ of the computing power, it can always force disagreement.

Acountability. Since our Byzantine agreement "locks" are not good enough to protect the banks, we need a new sheriff in town to bring the guilty parties to justice. 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 perpetrator of those illegal actions? Much in the way we prevent crime in the real world, we can prevent bad blockchain behavior via defense-in-depth: the basic Byzantine agreement protocol disallows illegal actions if the attacker has strictly less than $1/3$ of the network under their control, or if the network infrastructure is working properly and ensuring timely message delivery; when these guarantees fail, we record sufficient information to catch the criminal and take remedial actions. For example, the offending transactions could be voided or reparations provided to those that were stolen from; a mining pool that behaves badly could be banned from the blockchain; or in the case of a consortium blockchain, the offline legal system could be used to pursue damages. Alternatively, the threat of being caught might function as an incentive for good

[1] Bitcoin is typically assumed to be correct if at least 50% of the network is honest. However, this holds only under the assumption that communication is synchronous and timely. Eclipse attacks [17], for example, can violate correctness even with a majority honest.

behavior (e.g., if a certain deposit/stake were at risk).

The idea of accountability in distributed systems was pioneered by Haeberlen, Kuznetsov, and Druschel [15]. They show how to transform any distributed system into one that is accountable, as long as the network is synchronous. (In a partially synchronous sytem, they guarantee that faulty processes will be suspected forever, though definitive evidence may not be obtained.)

Over the last several years, accountability has been increasingly discussed as an important goal in blockchains, especially in proof-of-stake systems; see, e.g., [6], discussed below, which provides accountability for a synchronous blockchain protocol.[2] There is an increasing consensus in the community that blockchain systems *should* provide accountability.

Intuition. At first glance, accountability might seem impossible, potentially violating the impossibility of guaranteeing agreement when a majority of processes are malicious. However, technically, there is nothing preventing this sort of accountability, even in a partially synchronous system. From an intuitive perspective, the impossibility of Byzantine agreement depends on a lack of timely communication. For example, the adversary controlling strictly more than $1/3$ of the network might divide the remaining honest users into two groups where communication between groups is delayed; this would allow the adversary to convince each of the groups of a different outcome. However, when communication is eventually restored between the groups, they can discover the trickery and generate a proof that can bring the guilty parties to justice. Thus, in a partially synchronous network in which communication can be sometimes delayed (whether by simple network failures or by adversarial attacks), it remains possible to ensure strong forms of accountability as long as communication can eventually be restored among the honest parties.

A key challenge, however, is to protect the honest users from false accusation. It is not enough that you fail to receive a message from someone: they might be malicious, or the network might just be slow. And it is not enough for a small number of users to observe the bad behavior and report it to the authorities. How do we know whether those users were themselves honest? (The fact that a few users claim not to have received some messages they hoped to receive is really not sufficient evidence to slash a user's account or throw them in jail.) Instead, we need enough participants to observe the bad behavior to prove irrefutably that the criminal really did perpetrate the crime. And we need cryptographically sound evidence that concretely proves the bad behavior.

Most Byzantine consensus protocols already collect signatures and certificates and other forms of cryptographic evidence in order to ensure that the decisions properly guarantee agreement. For example, in PBFT [7], at each stage of agreement, the honest nodes construct a quorum certificate containing signatures from honest participants that attest to the decision being made. One might think that the certificates already being recorded are sufficient to ensure accountability. Perhaps if every node simply saves all the certificates it observes during the protocol, that will be sufficient to ensure accountability (even though the protocol was not designed with accountability in mind)?

In fact, that is not the case, and coming up with a generic solution is not trivial, especially as the Byzantine processes may refuse to participate in the detection algorithm, making it impossible (without synchrony) for honest nodes to prove their guilt. Even with partial synchrony [13] the problem usually requires a large amount of information to be logged and exchanged. In PBFT specifically, the malicious nodes can create enough uncertainty that the remaining honest nodes cannot pinpoint precisely which nodes are malicious. In particular, we demonstrate an execution where no verifier that collects the entire historical log from the honest nodes can correctly identify the malicious users. Specifically, the verified cannot distinguish two scenarios with different sets of faulty processes, and thus cannot hold faulty processes accountable. (We demonstrate such a scenario in Theorem 3.)

Results. In this paper, we develop a new Byzantine agreement algorithm among n nodes out of which t can be Byzantine, which we refer to as the *Polygraph Protocol*, with the following guarantees: (i) if $t < n/3$, then consensus is guaranteed, i.e., every honest device produces the same valid output; (ii) no matter the number of Byzantines nodes, if a disagreement occurs between two honest nodes, every honest node eventually produces an irrefutable proof as to the identity of at least $n/3$ malicious users.[3]

Polygraph message and communication complexities are $O(n^3)$ and $O(n^4)$, respectively, thanks to a bounded size justification. Notably, there is no asymptotic increase in message complexity over the non-accountable version of the protocol. This bound on message complexity is one of the hardest challenges in achieving accountability, and one of the main differentiators between Polygraph and more naive solutions. For example, a simple detection mechanism in which each signed message was first sent by all processes to all other processes (and only accepted with received signed correctly from $\lfloor \frac{2n}{3} \rfloor + 1$ nodes). This would help to prevent lies, but it would increase the message complexity by a factor of $\Omega(n)$, which Polygraph avoids.

We also show 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 1); and processes cannot detect even one guilty party by a fixed time limit (e.g., prior to decision), since

[2] An interesting suggestion in Casper is that when a user is caught cheating, their stake is slashed, incentivizing good behavior.

[3] While it might seem counterintuitive that Polygraph works regardless of the number of Byzantine nodes, note that when $t = n-1$ or $t = n$, no disagreement can occur by definition.

(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 describe how Polygraph can be used to hold participants accountable for their misbehavior in a blockchain application. First, we explain how to transform the accountable binary Byzantine consensus algorithm to accept arbitrary values, like the blocks of a blockchain. Second, we explain how consensus nodes can also be held accountable to client nodes that do not run the consensus as long as $t < 2n/3$. To evaluate Polygraph we implement it in the Red Belly Blockchain [11], and deploy it on 80 geodistributed machines. In particular, we compare the performance of the Red Belly Blockchain to this new "accountable" Red Belly Blockchain and observe that the cost associated with accountability, although largely visible, remains manageable.

Basic idea. The key observation underlying Polygraph is as follows: we can design Byzantine agreement protocols that can be defeated only by *dissembling*, i.e., by a malicious user sending different messages to different honest users, when (according to the protocol) it was supposed to send the same message to every honest user. And if there is dissembling, then the honest users have evidence of the bad behavior. Unfortunately, the malicious users might lie and accuse honest nodes of dissembling, and so extra care is needed before we can prove bad behavior.

The basic algorithm is based on the basic structure of the Byzantine agreement protocol in [10]. There are only a few different general approaches to solving Byzantine agreement, and most existing protocols are subtle variations of one or another. We chose this particular framework because its symmetric nature was compatible wth efficient accountability. By contrast, it is interestng that we found the PBFT framework (also the basis of HotStuff [30]) seemed less well suited, as the strong reliance on a primary seems to require more informatino exchange. (See Appendix B for an example scenario where PBFT does not distribute enough information to guarantee accountability.)

The Polygraph protocol proceeds in (asynchronous) rounds, where processes try to converge on a single estimate, and try to decide if enough processes have already adopted that estimate. First, a reliable broadcast (similar to that described in [4]) is used to distribute the proposal values. Then, a second phase of communication is used to determine whether enough processes have converged on a single value. Finally the processes decide, if they can; and if not, they update their estimate in an attempt to converge on a single value.

The important new aspects (and the subtlest parts of the protocol) are related to accountability, and the key to *efficient* accountability is twofold: we require that each of the messages in the second phase of communication are signed, and we collect carefully chosen *ledgers* of signed signatures that justify the estimate adopted. The exact requirement of the ledger depends on the situation; the important aspect is that a ledger guarantees that in a certain round, at least $2n/3$ processes broadcast a proposal for a single value. If we have two ledgers showing that in the same round, $2n/3$ processes broadcast a proposal for '1' and $2n/3$ processes broadcast a proposal for '0', then we can identify at least $n/3$ processes that dissembled, proposing both '0' and '1' in the same round.

We can show that, by propagating the ledgers carefully, we can ensure that any process that decides has sufficient information to justify their decision. Honest processes that eventually disagree can exchange sufficient information to ensure that every process can provably identify at least $n/3$ users that acted maliciously.

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 Polygraph, which solves the accountable binary Byzantine consensus problem. Section 6 analyses empirically Polygraph in a geodistributed blockchain and Section 7 concludes. Appendices A, B, C, D and E present the proof of the impossibility result, the proof of correctness of Polygraph, the multivalue extension and the application of Polygraph to blockchains, and the counter-example that PBFT is not accountable, respectively.

2 Background and Related Work

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

PeerReview. Haeberlen, Kuznetsov, and Druschel [15] 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!

A natural alternative to The Polygraph Protocol, then, would be to take a standard existing consensus protocol like Paxos [27] and apply the PeerReview technique. The first issue is that, unlike in PeerReview, we want to guarantee correct outcomes when $t < n/3$, and rely on accountability when t is large. Thus our base protocol must already be a (deterministic) Byzantine Agreement protocol, such as PBFT [7]

or DBFT [10].

A second 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.[4] The Polygraph Protocol, by contrast, produces a concrete proof of malicious behavior that is completely under the control of the honest processes.

A third issue has to do with message (and communication) complexity. The Polygraph Protocol has no (asymptotic) increase in message complexity over the base Byzantine Agreement Protocol on which it is based, and it increases the communication complexity by a factor of $\Theta(n)$. By contrast, PeerReview increases the message complexity by $\Theta(n^2)$ per audit interval (or $\Theta(n \log n)$ for probabilistic guarantees), while also increasing the communication complexity by a factor of $\Theta(n)$. Thus The Polygraph Protocol is more message efficient, while still preserving deterministic accountability guarantees.

As such, The Polygraph Protocol is more suitable for use in blockchain protocols. The key disadvantage of The Polygraph Protocol, of course, is that it is not general: it is solving a specific problem (Byzantine Agreement), rather than giving a solution for any distributed system. Moreover, The Polygraph Protocol detects misbehavior only when there is disagreement. The Byzantine processes can violate the protocol as much as they like, without detection, as long as the resulting consensus has no disagreement!

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 [6] 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 The Polygraph Protocol'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. Consider, for example, that Byzantine validators always create the blocks at indices multiple of 100 and sends the block to less $2n/3$ honest validators.[5] Repeating this process guarantees that $2n/3$ honest processes can never vote for the same link, and consensus will never be reached.

Even so, as a result of Casper, there has been much discussion in and around the blockchain community about accountability. It is certainly a popular idea today that accountability would be a desirable property of blockchains.

Accountability in the context of blockchain fairness was raised by Herlihy and Moir in a keynote address [18], and the idea of "accountable Byzantine fault tolerance" has been discussed [5]. 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.

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., [22, 31, 32]). Yumerefendi and Chase [32] developed an accountable system for network storage, and Repeat and Compare [26] 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. [2] 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. And Intrusion Detection Systems (e.g., [12, 19, 24] provided heuristics and techniques for detecting malicious behaviors in a variety of different systems.

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. They tend to be designed for use in fully asynchronous systems (i.e., to capture synchrony assumptions), and protocols that rely on them tend to assume that $t < n/3$.

[4] To be precise, they refer to this situation as one where a malicious process is permanently *suspected* but never *exposed*; our goal is to guarantee that at least $n/3$ malicious processes are exposed, in their terminology.

[5] Note that this requires Byzantine users have a large mining power if proof-of-work is needed for block creation.

Most of the work in this area has focused on detecting crash failures (see, e.g., [8]). There has been some interesting work extending this idea to detecting Byzantine failures [15, 16, 20, 24]. Malkhi and Reiter [24] 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 [20] 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 [25] 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 [23] 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. 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 for Byzantine behavior.

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 t_{GST} known as the *global stabilization time*, unknown to the processes, such that any message sent after time t_{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 t_{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 [21], 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 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 [28], but Lemma 9 shows that our accountable binary Byzantine consensus protocol ensures even a stronger validity property.

4 Impossibility Results

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 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 $t_0 < t$.*

We say that a verification algorithm V is *swift* if it guarantees: 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 2. *For a consensus solved while $t < n/3$, there does not exist a swift verification algorithm for $t_0 < t$.*

The next theorem states that PBFT [7], which already exchange signed messages, cannot be made accountable if processes simply exchange all their locally stored information and cross check messages.

Theorem 3. *Without storing and exchanging additional information, in case of disagreement in an execution with $t > t_0$, no verification algorithm ensuring the detection of more than $O(1)$ Byzantine exists. A fortiori, no verification algorithm ensuring the detection of $t_0 + 1$ Byzantine exists, which means no verification algorithm can make PBFT accountable.*

The complete proof with the two indistinguishable scenarios is deferred to Appendix E but the intuition relies on two processes deciding different values in different views, without being able to detect enough guilty processes.

PBFT. In PBFT, processes move through consecutively numbered views. Each view has one primary process that executes three phases and changes when sufficiently many VIEW-CHANGE messages are collected. If in the second phase, a process i receives enough PREPARE messages supporting m, then i broadcasts a COMMIT message for m. If a process receives $n - t_0$ commit messages with m and the proper view and sequence number, then it decides m.

Bad execution. An execution where $t \geq t_0 + 1$ and two correct processes i and j decide differently without being able to output $t_0 + 1$ guilty processes is as follows. In view v_0, both i and j receive $2t_0 + 1$ PREPARE messages for message m and send COMMIT messages.

- **From i's standpoint**: process i receives $2t_0 + 1$ commit messages from some processes so i commits m, it then stop participating in the protocol until j decides null.
- **From j's standpoint**: the view v_0 ends as j receives $2t_0 + 1$ VIEW-CHANGE messages but only t_0 of them containing m. View v_1 starts and j does not receive any message from the primary of view v_1 or any VIEW-CHANGE messages. At the end of v_1, j receives a new view message from p_2 containing the properly signed view change message from enough nodes with a null reference for the current sequence number and j receives sufficient information in v_2 to decide null for that sequence number.

How do the malicious nodes force a decision at j without revealing too much information? From i's perspective, there are $2t_0 + 1$ nodes that committed to m, and so there are at least $t_0 + 1$ of those nodes that must have participated in the view change to view v_1. (The other t_0 who committed to m may have been omitted due to network synchrony issues.) Any node of those $t_0 + 1$ nodes that did not include m in its view change message must be Byzantine. From j's perspective, t_0 of those nodes did not lie, and so if i and j communicate, they can identify a single Byzantine node. Denote by V_1^j the view change messages seen by j.

However, the leader of v_1 includes a different set of nodes in the view change messages sent to everyone else. Denote by V_1^p these view change messages, which consist of the other $2t_0 + 1$ identities, and no mention of m. If i and j had access to V_1^p, then they could readily detect more Byzantine nodes. But they will never get this information.

Since there is now a large collection of nodes that believe view v_0 ended with no commitment to m, it is easy for the primary for v_1 to convince the nodes to commit to null instead of m in view v_1. Then, another view change operation occurs to view v_2 (which helps to hide the set V_1^p from j), and node j learns about the decision of null and commits to it.

After i and j realize the disagreement, the other processes stay mute forever (e.g., because they fail). Nodes i and j never see V_1^p, and will never be able to prove that $t_0 + 1$ Byzantine nodes are guilty. They can, at best, prove that one node in V_1^j was Byzantine, along with the primary in v_1. The large majority of Byzantine nodes escape unscathed.

Observations. There are two interesting observations here. One is that the only way around this problem is for nodes to exchange information about all prior view changes; this ensures that a node that decides is aware of all the earlier view change messages. Unfortunately, piggybacking that information would be expensive (and could get more expensive as more views occur).

Second, threshold signatures do not appear to help. A threshold signature provides a proof that enough nodes signed a message, but it provides no good way to prove *which specific* nodes were signatories. (And, from an information theory perspective, a threshold signature does not contain enough information.) Thus for PBFT, you cannot take a shortcut by using threshold signatures and piggybacking more information.

Note that we also show in Theorem 9 that even if one changes PBFT so that processes store and exchange the log of received messages of the $t_0 - 1$ preceding views, then it is impossible to make the resulting algorithm accountable. This result is deferred to Appendix E by lack of space.

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 [10] that was proved correct using the ByMC model checker [29] and that does not use the leader-based pattern mentioned in the proof of Theorem 3. Then, we focus on the key aspects that lead to accountability, specifically, the "ledgers" and "certificates." While this section tackles the binary agreement, Appendix C generalizes this result to arbitrary values. In Appendix B, we prove that the algorithm is correct.

As a notational issue, we indicate that a process p_i sends a message to every other process by: $\mathsf{broadcast}(\mathit{TAG}, m) \rightarrow$ *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.)

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 with estimate v, then that is the only possible decision due to the reliable broadcast bv-broadcast in Phase 1 (see below).

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 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 $t < n/3$).

BV-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 3); (ii) every message delivered to an honest pro-

Algorithm 1 The Polygraph Protocol

```
 1: bin-propose(v_i):
 2:   est_i = v_i
 3:   r_i = 0
 4:   timeout_i = 0
 5:   ledger_i[0] = ∅
 6:   repeat:
 7:     r_i ← r_i + 1;                                    ▷ increment the round number and the timeout
 8:     timeout_i ← timeout_i + 1
 9:     coord_i ← ((r_i − 1)  mod n) + 1                  ▷ rotate the coordinator
   ▷ Phase 1:
10:     bv-broadcast(EST[r_i], est_i, ledger_i[r_i − 1], i, bin_values_i)   ▷ binary value broadcast the current estimate
11:     if i = coord_i then                               ▷ coordinator rebroadcasts first value received
12:       wait until (bin_values_i[r_i] = {w})            ▷ bin_values stores messages received by binary value broadcast
13:       broadcast(COORD[r_i], w) → messages_i
14:     StartTimer(timeout_i)
15:     wait until (bin_values_i[r_i] ≠ ∅ ∧ timer_i expired)
   ▷ Phase 2:
16:     timer_i ← timeout_i                               ▷ reset the timer
17:     if (COORD[r_i], w) ∈ messages_i from p_coord_i ∧ w ∈ bin_values_i[r_i]) then   ▷ favor the coordinator
18:       aux_i ← {w}
19:     else aux_i ← bin_values_i[r_i]                    ▷ otherwise, use any value received
20:     signature_i = sign(aux_i, r_i, i)                 ▷ sign the messages
21:     broadcast(ECHO[r_i], aux_i[r_i], signature_i) → messages_i   ▷ broadcast second phase message
22:     wait until values_i = ComputeValues(messages_i, bin_values_i, aux_i) ≠ ∅
   ▷ Decision phase:
23:     if values_i = {v} then                            ▷ if there is only one value, then adopt it
24:       est_i ← v
25:       if v = (r_i  mod 2) then                        ▷ decide if value matches parity
26:         if no previous decision by p_i then decide(v)
27:     else
28:       est_i ← (r_i  mod 2)                            ▷ otherwise, adopt the current parity bit
29:     ledger_i[r_i] = ComputeJustification(values_i, est_i, r_i, bin_values_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.

cess was broadcast by at least $t+1$ processes (see Lemma 2).

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.

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).

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

Algorithm 2 Helper Components

```
1:  bv-broadcast(MSG, val, ledger, i, bin_values):
2:      broadcast(BVAL, ⟨val, ledger, i⟩) → 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, ⟨v, ·, j⟩)
5:        if (BVAL, ⟨v, ·, ·⟩) received from (t0 + 1) distinct processes and (BVAL, ⟨v, ·, ·⟩) not yet broadcast then
6:          Let ℓ be any non-empty ledger received in these messages.       ▷ one of the received ledgers is enough
7:          broadcast(BVAL, ⟨v, ℓ, j⟩)                                     ▷ Echo after receiving (t0 + 1) copies.
8:        if (BVAL, ⟨v, ·, ·⟩) received from (2t0 + 1) distinct processes then
9:          Let ℓ be any non-empty ledger received in these messages.       ▷ one of the received ledgers is enough
10:         bin_values ← bin_values ∪ {⟨v, ℓ, j⟩}                          ▷ deliver after receiving (2t0 + 1) copies

11: ComputeValues(messages, b_set, aux_set):                      ▷ check if there are n − t0 compatible messages
12:     if ∃S ⊆ messages where the following conditions hold:
13:       (i) |S| contains (n − t0) 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 ∃S ⊆ messages where the following conditions hold:
17:       (i) |S| contains (n − t0) 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(∅)

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:         return ledger[r_i]_i = ledger ℓ where (EST[r_i], ⟨v, ℓ, ·⟩) ∈ bin_values_i
25:       else return ledger[r_i]_i = ∅
26:     else return ledger[r_i]_i = (n − t0) signed messages from messages_i containing only value est_i
27:     if values_i = {(r_i mod 2)} ∧ no previous decision by p_i in previous round then
28:       certificate_i = (n − t0) signed messages from messages_i containing only value est_i
29:       broadcast(est_i, r_i, i, certificate_i)                                   ▷ transmit certificate to everyone
```

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.)

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 (t_0+1) locations, this conflict identifies $(n-t_0)$ 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.

Analysis. 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 8, Corollary 1, and Lemma 10 in Appendix B.2:

Theorem 4. *The Polygraph Protocol is a correct Byzantine agreement protocol guaranteeing agreement, validity, and termination.*

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

Theorem 5. *The Polygraph Protocol is accountable.*

If all the processes are honest, or if the Byzantine corruptions are oblivious to the processor identities, 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 1 (Complexity)**.** *Each round of the Polygraph protocol has message complexity $O(n^2)$ and communication complexity $O(n^3)$.*

Proof. The BV-broadcast has message complexity $O(n^3)$, as each of the n processes echoes up to 2 (binary) BVAL messages, each of which is re-transmitted to all n processes. The communication complexity is at most $O(n^3)$, since each message may contain a ledger which contains $O(n)$ signatures. 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). □

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 [11] based on DBFT [10] and the "Accountable Red Belly Blockchain" based on Polygraph as described in Appendices C and D. We implemented Polygraph using the RSA 2048 bits signature scheme to authenticate messages. Note that threshold signatures do not store sufficient bits to trace back the identity of the guilty processes and message authentication codes would not offer the transferrable authentication we need [9]. We deploy both blockchains on up to $n=80$ c4.xlarge AWS virtual machines located in 5 availability zones on two continents: Frankfurt, Ireland, London, N. California and N. 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 is a blockchain systems that builds upon DBFT [10] that implements a UTXO model and commits tens of thousands of 400-byte transactions per second up to hundreds of geo-distributed processes [11]. We introduced verification sharding so that between $t+1$ and $2t+1$ processes verify the signature of each transaction using the elliptic curve digital signature algorithm with secp256k1 parameters, whereas all processes participate in the (accountable) Byzantine consensus to decide a unique block at each index of the chain.

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 nodes to 40% at 80 nodes. 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 GiBRAM, and "moderate" network performance. On the one hand, as expected even if this low-end situation, the Red Belly Blockchain scales in that its performance does not drop [11]. On the other hand, we can see the Accountable Red Belly

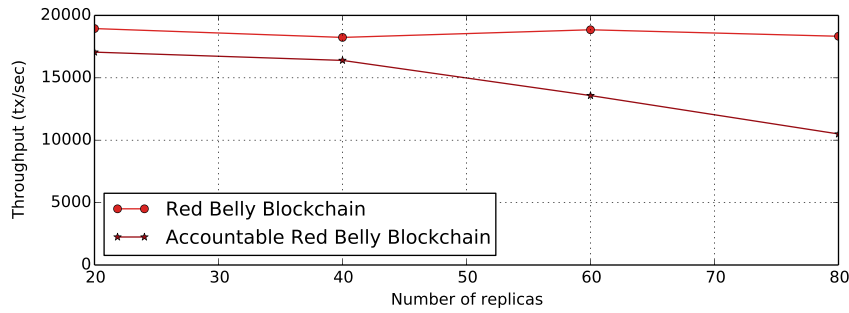

Figure 1: The overhead of accountability in the Red Belly Blockchain with 400 B transactions cryptographically verified with ECDSA signatures and parameters secp256k1 when deployed on 80 replicas geo-distributed in Frankfurt, Ireland, London, N. California and N. Virginia

Blockchain still offers a throughput of more than 10,000 transactions per second at 80 geo-distributed nodes, which remains superior to most non-accountable blockchains. Finally, the Accountable Red Belly Blockchain commits several thousands of transactions per second at 80 nodes, 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 malicious users that cause disagreement. Polygraph is practical thanks to its bounded justification size that does not increase with the number of rounds. As future work, we would like to explore whether an algorithm that solves simultaneous Byzantine agreement [14] could be easily changed into an accountable Byzantine agreement solution.

Acknowledgements

We wish to thank Alejandro Ranchal Pedrosa for his help with the experiments. This research is supported under Australian Research Council Discovery Projects funding scheme (project number 180104030) entitled “Taipan: A Blockchain with Democratic Consensus and Validated Contracts” and Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled “The Red Belly Blockchain: A Scalable Blockchain for Internet of Things”.

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] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. *SOSP*, 2005.

[3] Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Blockchain abstract data type. *IACR Cryptology ePrint Archive*, 2018:561, 2018.

[4] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation 75:130-143*, 1985.

[5] Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, The University of Guelph, June 2016.

[6] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. Technical Report 1710.09437v4, arXiv, Jan 2019.

[7] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[8] 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.

[9] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC ’12, page 301?308, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (NCA’18)*. IEEE, 2018.

[11] Tyler Crain, Chris Natoli, and Vincent Gramoli. Evaluating the Red Belly Blockchain. Technical Report 1812.11747, arXiv, 2018.

[12] Dorothy E. Denning. An intrusion-detection model. *IEEE Trans. Software Eng.*, 13(2), 1987.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery, Vol. 35, No. 2, pp.288-323*, 1988.

[14] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a byzantine environment I: crash failures. In *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA, USA, March 1986*, pages 149–169, 1986.

[15] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. *SOSP'07*, 2007.

[16] Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *Principles of Distributed Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings*, pages 99–114, 2009.

[17] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symp. (USENIX Security 15)*, pages 129–144. USENIX Association, 2015.

[18] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 27–30, 2016.

[19] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Detection and removal of malicious peers in gossip-based protocols. In *In Proceedings of FuDiCo*, June 2004.

[20] Kim P. Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. Byzantine fault detectors for solving consensus. *British Computer Society*, 2003.

[21] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[22] Butler W. Lampson. Computer security in the real world. In *In Proc. Annual Computer Security Applications Conference*, December 2000.

[23] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 10–10, 2007.

[24] Dahlia Malkhi and Michael K. Reiter. Unreliable intrusion detection in distributed computations. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 116–125, 1997.

[25] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 108–117, 2002.

[26] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*, 2007.

[27] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the PAXOS algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

[28] Roberto De Prisco, Dahlia Malkhi, and Michael K. Reiter. On k-set consensus problems in asynchronous systems. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 257–265, 1999.

[29] Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain Byzantine fault tolerance. In *In the Workshop on Verification of Distributed Systems (VDS'19)*, June 2019.

[30] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[31] Aydan R. Yumerefendi and Jeffrey S. Chase. Trust but verify: accountability for network services. In *Proceedings of the 11st ACM SIGOPS European Workshop, Leuven, Belgium, September 19-22, 2004*, page 37, 2004.

[32] 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 $\{\varepsilon\}$, $\{\varepsilon\} \in \Sigma$ and $\{\varepsilon\} \in \Lambda$. The process is fitted with a clock which allows it to take a transition after a certain timeout with $\{\varepsilon\}$ as argument for δ.

The agreement according to a local view. A node processing an agreement protocol follows an ordered succession of states $p(s_k) = s_0, ..., s_k$ called a path. On the path, the node 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 node 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 node 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 6 (Theorem 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 7 (Theorem 2)**.** *For a consensus solved while $t < n/3$, it does not exist a swift verification algorithm for $t_0 < t$.*

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, $\forall i \in P, V(s_{bd}^i) = \emptyset$ and $\forall j \in R, V(s_{bd}^j) = \emptyset$. But in scenario C, we should have $\exists i \in PV(s_{bd}^i) \neq \emptyset$ or $\exists j \in R, V(s_{bd}^j) \neq \emptyset$ 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 [10], 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 2 (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 3 (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 $\mathsf{BV\text{-}broadcast}(\mathsf{MSG}, v, \cdot, \cdot, \cdot)$. 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 4 (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 5 (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 3. ☐

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 6 (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 7. *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 3 that v is eventually delivered to every honest process, and from Lemma 2 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 8 (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 7, 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 9) than the one required (Corollary 1). (The general Polygraph protocol stated in Appendix C that applies to arbitrary values only ensures the required validity.)

Lemma 9 (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 2 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 2 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 1 (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 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 5, 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 4, 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 7. □

Thus we conclude (see Theorem 4) 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 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 6 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 6 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 [10]: 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 12 (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.

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 13 (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 [4]. 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 [4] 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

Algorithm 3 The General Polygraph Protocol

```
1:  gen-propose(v_i):
2:    RB-broadcast(EST, ⟨v_i, i⟩) → 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 ∃ v,k : (EST, ⟨v,k⟩) ∈ messages_i then
6:        if BIN-CONSENSUS[k] not yet invoked then
7:          BIN-CONSENSUS[k].bin-propose(1) → bin-decisions[k]_i
8:    until ∃k : bin-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) → bin-decisions[k]_i      ▷ for these, propose 0
12:   wait until for all k, bin-decisions[k] ≠ ⊥                  ▷ wait until all the instances decide
13:
14:   j = min{k : bin-decisions[k] = 1}                            ▷ choose the smallest instance that decides 1
15:   wait until ∃ v : (EST, ⟨v, j⟩) ∈ messages_i                  ▷ wait until you receive that value
16:   decide v                                                     ▷ return that value
```

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 ℓ_i containing the n − t_0 signed messages (ECHO, v, j).
7:    broadcast(READY, v, ℓ_i, j)                                  ▷ send READY message and ledger for v to all.
8:  upon receiving t_0 + 1 distinct messages (READY, v, ·, j) and not having sent a READY message:
9:    Set ℓ_i to be one of the (valid) ledgers received (READY, vℓ, j).
10:   broadcast(READY, v, ℓ, j)                                    ▷ send READY message for v to all.
11: upon receiving n − t_0 distinct messages (READY, v, ·, j) and not having delivered a message from j:
12:   Let ℓ be one of the (valid) ledgers received (READY, v, ℓ, j).
13:   deliver(v, j)                                                ▷ send READY message for v to all
```

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:

Lemma 14 (*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 15. *(*RB-Validity*) If* $t \leq t_0$*, 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 $(\text{READY}, v, \cdot, j)$ 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 $(\text{READY}, v, \cdot, j)$. 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 16. *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 $(\text{ECHO}, v, \cdot)$ and has constructed a ledger ℓ_i containing those signed messages, or (ii) it has received $t_0 + 1$ distinct READY messages $(\text{ECHO}, v, \cdot, j)$, 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 $t \leq t_0$, every honest process

delivers its message:

Lemma 17. *(*RB-Send*) If $t \leq t_0$, 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 $t \leq t_0$, 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 $t \leq t_0$, 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 $t \leq t_0$.

Lemma 18. *(*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 16.) 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 19. 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 16, 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 Application to Blockchain

In this section we explain how the general Polygraph protocol can held blockchain service providers accountable to blockchain client nodes 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 nodes, 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. [3]. 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 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.

Now we consider three cases depending on the number of Byzantine processes, namely the nominal case, the degrading 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 nodes, 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 nodes 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.

E PBFT Does Not Solve the Accountable Byzantine Agreement

In this section, we show that PBFT does not collect enough information using quorum certificates to ensure accountability. This is expected for the unchanged PBFT algorithm as it was not designed with this goal in mind, what is more surprising is that it cannot be made accountable by cross-checking the signed messages it exchanges. Our impossibility result even holds for non-trivial changes as long as the algorithm does not exchange and store strictly more information.

The proof relies on the existence of an execution with up to $2t_0 + 1$ faults violating agreement where correct processes cannot detect the $t_0 + 1$ malicious processes responsible for the disagreement.

Preliminaries on PBFT For the simplicity in the presentation, we use the original notation of Castro and Liskov [7] in the reminder of this section. In PBFT, the replicas move through a succession of configurations called *views* that are numbered consecutively. In a view, one replica is the *primary* and the others are *backups*. View changes are carried out when the primary is suspected to have failed.

When the primary receives a client request, it starts a three-phase protocol to atomically multicast the request to the replicas. The three phases are pre-prepare, prepare, and commit. If at the second phase of a view v, a process i receives enough messages supporting the message m for sequence number sn, it satisfies the predicate $prepared(m, v, sn, i)$, which allows it to broadcast a (m, v, sn)-commit message to every process. If a process receives $n - t_0$ (m, v, sn)-commit messages with the same triplet (m, v, sn), then it commits the message m for the sequence number sn irrevocably, which is denoted by the predicate $committed\text{–}local(m, v, sn, j)$. PBFT uses a garbage collector and snapshots to reclaim memory, however, we consider for simplicity that PBFT is given infinite memory so that our example also holds without the issues potentially raised

by the garbage collector.

The view change is scheduled as follows: a backup mantains a timer at each view. If the backup timer expires in view $v-1$, the backup starts a view change to move the system to view v, stops accepting messages (other than CHECKPOINT, VIEW-CHANGE and NEW-VIEW messages) and multicasts a (v,P)-VIEW-CHANGE message to all replicas, where P is a set containing the messages that the backup prepared in the past. If message $m \in P$, we say that the backup propagated message m from view $v-1$ to view v. Otherwise, if it is faulty, it may omit to include these messages in P. When the new primary designated for view v receives $n-t_0$ valid VIEW-CHANGE messages for view v from other replicas, it multicasts a (v,V,O)-NEW-VIEW message to all other replicas, where V is a set containing the valid (v,P)-VIEW-CHANGE messages received by the primary and O contains the set of messages propagated to the view v. Before committing any message at view v, a process waits for $(n-t_0)$ VIEW-CHANGE messages for all the views preceding v.

How to disagree Our goal is to construct an execution of PBFT where at least two correct processes disagree, and yet they cannot determine for sure who is faulty. What would such an execution necessarily look like? Assume $t \geq n/3$ and that $i,j \in C$ disagree on the sequence number sn^*, namely we have both *committed–local*(m,v,sn^*,i) and *committed–local*(m',v',sn^*,j). To reach this state, i has collected $2t_0+1$ (m,v,sn^*)-commit messages from a set of processes, say R_i^c, and j has collected $2t_0+1$ (m',v',sn^*)-commit messages from a set of processes, say R_j^c, where $v \neq v'$. Without loss of generality, let us fix $v < v'$, hence process j went through the subsequent views after v until reaching v'. The idea is to avoid the possibility for i and j to gather enough information to detect the guilty processes who sent (m,v,sn)-commit messages to i and did not propagate m to view $v+1$. This crucial information is stored in a set V_1 messages received by the primary p_1 of view v_1. Because of the primary-based structure of PBFT, j has to accept that p_1 is potentially Byzantine and can commit m^{null} without the reception of V_1 from p_1. Then, without V_1, i and j will not be able to say who omit the propagation of m, so that PBFT cannot be made accountable.

Accountability violation We now show that PBFT, as currently stated [7], does not store and/or share sufficient information to ensure accountability, i.e., to prove t_0+1 processes guilty.

The proof relies on the impossibility for all honest processes to distinguish two executions where distinct sets of processes are Byzantine, so that no verification algorithm can output t_0+1 Byzantine processes after disagreement.

Definition 2 (Indistinguishable execution)**.** *Let α and α' be two execution fragments of a deterministic algorithm $\mathcal{A}$. We note for every process i participating to the execution of $\mathcal{A}$, log_i^α the set of messages received by i. Because of determinism, the execution of i is defined by the received messages. Thus we say that α is indistinguishable from α' from i's point of view, noted $\alpha \overset{i}{\sim} \alpha'$, if $log_i^\alpha = log_i^{\alpha'}$, to capture the idea that i cannot tell if it executed α or α'. Furthermore, after having executed an execution fragment, a set of nodes can exchange information to attempt to identify the scenario they executed. Thus we say that α is indistinguishable from α' from W's point of view, noted $\alpha \overset{W}{\sim} \alpha'$, if $\bigcup_{i\in W} log_i^\alpha = \bigcup_{i\in W} log_i^{\alpha'}$ to capture the idea that even with cooperation, no processes from the set W can tell if it executed α or α'.*

The following lemma discusses an *untouchable* set U of processes that cannot be detected without violating accountability because for each process of U there exists an indistinguishable execution where this process is correct.

Lemma 20 (Untouchable set)**.** *Let $\mathcal{A}$ be a t_0-resilient Byzantine Agreement protocol. Let B and B' are two scenarios of $\mathcal{A}$ with $t > t_0$. Let C^B and $C^{B'}$ be the sets of correct processes in B and B' respectively. Let $U = C^B \cup C^{B'}$ be the set of process who is correct in B or B'. For any verification algorithm V executed after B or B', V cannot output any process in U and cannot make $\mathcal{A}$ accountable, if the three following properties hold:*

1. *Processes i and j are two correct processes who disagree in B and B';*
2. *Scenarios B and B' are indistinguishable from i and j's point of view, namely $B \overset{i,j}{\sim} B'$;*
3. *There are at least $n-t_0$ processes in U, namely $|U| \geq n-t_0$.*

Proof. Due to the indinstiguishability, V outputs the same set in B and B'. But V outputs only undeniable guilty process. Thus if $p_x \in U$, V cannot output a set containing p_x. That is, V cannot output any process in V. Moreover if $|U| \geq n-x$, V can at most detect x undeniably guilty processes, so if $|U| \geq n-t_0$, V cannot detect more than t_0 undeniably guilty nodes, which means V cannot make $\mathcal{A}$ accountable. □

In the remainder, we show that the three properties of Lemma 20 hold, such that Theorem 8 follows. First, we build the expected B and B' scenarios. However, if j is able to deduce that it received a message from at least one Byzantine process before decision, it can wait for another message before deciding. Thus, we build a set of scenarios, linked by indistinguishability relationships to prove that j has to decide in B and B' without receiving any additional information.

The scenarios of interest are Z, Z', A and A' with $t \leq t_0$, and B and B' with $t > t_0$, whose relationships are depicted in Figure 2. Processes of $W_3(\Diamond)$ trivially commit m^{null} in Z and Z'. Because $A \overset{W_3}{\sim} Z$ and $A \overset{W_3}{\sim} Z'$, processes of W_3 still commit m^{null} in A and A'. In scenario A, at each view, j receives $n-t_0$ messages from only correct processes, so that j does not demand other messages in scenarios indistinguishable from A

$$\begin{array}{ccc} Z & & Z' \\ \diamond \wr & & \wr \diamond \\ A & \overset{j}{\sim} & A' \\ \diamond \wr\wr\ j & & j\ \wr\wr \diamond \\ B & \overset{i,j}{\sim} & B' \end{array}$$

Figure 2: **- Chaining of scenarios by indinstinguishability relationships.** $X \overset{S}{\sim} Y$ means the set S, even with cooperation, is not able to dinstinguish X from Y, while $\diamondsuit$ is the symbol for W_3 a set of t_0 correct nodes (before being mute) who will be used in the proof, thus $X \overset{\diamondsuit}{\sim} Y$ means $X \overset{W_3}{\sim} Y$.

before committing, and because of agreement with W_3 correct, j commits m^{null}. In scenarios B and B', i commits m because of too many Byzantine nodes. In the extension of B and B' where i and j realize the disagreement, they collect log of i and j but still cannot distinguish B and B' with an untouchable set (see Lemma 20) containing at least $\{i, j\} \cup W_1 \cup W_2 \cup W_3$, that is at least $3t_0 - 2$ processes, making accountability impossible by Lemma 20.

First, we build Z and Z' where every correct nodes eventually commits m^{null} (see Lemma 21). Second, we will build A and A' and show that j has to decide m^{null} (see Lemma 22 and 23). Third, we build B and B' showing that $B \overset{j}{\sim} A$ and $B' \overset{j}{\sim} A'$, which means j has to decide without receiving additional messages, and so disagrees with i (see Lemmas 24 and 25). Finally, we show that $B \overset{i,j}{\sim} B'$ with an untouchable set of at least $3t_0 - 2$ nodes (see Lemma 26) which is enough to prove that PBFT is not accountable (see Theorem 8).

We consider disjoints set of nodes $W_1, W_2, W_3, \{w_4\}, \{w_5\}, \{w_6\}, \{i\}$ and $\{j\}$, where $|W_1| = t_0 - 2, |W_2| = t_0 - 2$ and $|W_3| = t_0$. In all scenarios, we consider three consecutive views v_0, $v_1 = v_0 + 1$ and $v_2 = v_0 + 2$, with respective primaries $p_0 \in W_3$, w_6 and $p_2 \in W_3$. We also define the following sets:

- $W_7 = W_2 \cup W_3 \cup \{w_4\} \cup \{w_5\} \cup \{w_6\}$;
- $W_8 = W_1 \cup W_3 \cup \{w_4\} \cup \{w_5\} \cup \{j\}$;
- $W_9 = W_1 \cup W_3 \cup \{w_4\} \cup \{w_5\} \cup \{w_6\}$.

Below, we build a sequence of indistinguishable executions to show how i and j can see different things within the same execution.

Notations To simplify the understanding, we illustrate the different scenarios with tables where we note:

- r stands for round. There are two lines per round, the first line is about the sent messages, while the second line is about the received messages. We refer to a process receiving the same message *msg* from different processes as $msg : \{\triangle, \diamondsuit, ...\heartsuit\}$ with the associated symbol for each set of processes.
- *nv* stands for new view.
- *pre* stands for prepare.
- *com* stands for commit.
- $(v_x, m)VC$: m attached to VIEW-CHANGE message at view v_x.
- $(v_y, \emptyset)VC$: m does not appear in the VIEW-CHANGE message of view v_y.
- $(v_x, m, V_x)NV$: m and V_x attached to the NEW-VIEW message where V_x holds the VIEW-CHANGE message received by the primary for this view v_x.
- $(v_x, \emptyset, V_y)NV$: m is not propagated with the NEW-VIEW message, which means V_y cannot contain VIEW-CHANGE messages propagating m.
- ✓ indicates a prepared message.
- ✓✓ indicates commited message.

Scenario Z Consider the scenario where $W_1 \cup \{i\} \cup \{j\}$ (of size t_0) failed by crashing directly, so that the remaining set W_7 is correct. W_7 attempt to decide m at view v_0 but no nodes prepare m (e.g., due to network asynchrony) so that m is not propagated to view v_1 and finally m^{null} gets decided at view v_2.

view v_0: The primary $p_0 \in W_3$ broadcasts the pre-prepared message for (m, v_0, sn^*). Let assume that the timer of every process expires before any process of W_7 prepared this message so that they all broadcast a (v_1, P)-VIEW-CHANGE message without mentioning m.

view v_1: Let the primary w_6 of view v_1 deliver a set of $2t_0 + 1$ VIEW-CHANGE messages signed by W_7, without any mention of m for sequence number sn^*. The primary puts those messages in set V_1, builds a set O_1 with a *pre-prepare* message m^{null} for sequence number sn^* and broadcasts its NEW-VIEW message with V_1, and O_1. This message is delivered by W_7, but again, because of network delays, the timer of every node expires before any node of W_7 prepared this message so that they all broadcast a (v_2, P)-VIEW-CHANGE message without mentioning m.

view v_2: The primary $p_2 \in W_3$ of view v_2 delivers a set of $2t_0 + 1$ VIEW-CHANGE messages signed by W_7 without mentioning m for sequence number sn^*. The primary puts those messages in set V_2, builds a set O_2 with a *pre-prepare* message m^{null} for sequence number sn^* and broadcasts its NEW-VIEW message with V_2, and O_2. Finally, the third phase is completed and every node in W_7 commits (m^{null}, v_2, sn^*).

Scenario Z' This scenario is the same as Scenario Z except that $W_2 \cup \{i\} \cup \{j\}$ failed by crashing while W_9 replaces W_7. For the sake of completeness, we re-state it below: Processes

view	r	$i:+$	$j:\times$	$W_1:\square\square$	$W_2:\triangle\triangle$	$W_3:\lozenge\lozenge\lozenge\lozenge$	$\omega_4:\clubsuit$	$\omega_5:\spadesuit$	$\omega_6:\heartsuit$
v_0	nv	.	.	.	.	$(v_0,m)NV$	.	.	.
v_0	nv	.	.	.	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$
v_0	pre	.	.	.	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$
v_0	pre	.	.	.	.	.	.	.	.
v_1	vc	.	.	.	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$
v_1	vc	.	.	.	.	.	.	.	$(v_1,\emptyset):V_1=\{\ \triangle\triangle\ \lozenge\lozenge\lozenge\lozenge\ \clubsuit\ \spadesuit\ \heartsuit\ \}$
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1)NV$
v_1	nv	.	.	.	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$
v_1	pre	.	.	.	.	.	.	.	.
v_2	vc	.	.	.	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$
v_2	vc	.	.	.	.	$(v_2,\emptyset):V_2=\{\ \triangle\triangle\ \lozenge\lozenge\lozenge\lozenge\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	.	.	.
v_2	nv	.	.	.	.	$(v_2,\emptyset,V_2)NV$	.	.	.
v_2	nv	.	.	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$
v_2	pre	.	.	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_2	pre	.	.	.	✓	✓	✓	✓	✓
v_2	com	.	.	.	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_2	com	.	.	.	✓✓	✓✓ : $\{\ \triangle\triangle\ \lozenge\lozenge\lozenge\lozenge\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	✓✓	✓	✓✓
.	.	.	.	.	.	.	.	.	.

Table 1: Scenario Z : Without news from $\{i\}\cup\{j\}\cup\{W_1\}$, the other nodes need to commit m^{null}; they do not expect messages from $\{i\}\cup\{j\}\cup\{W_1\}$ because they could have crashed

of W_9 are correct and attempt to decide m at view v_0 but no processes prepare m so that m is not propagated to view v_1 and finally m^{null} gets decided at view v_2.

view v_0: The primary $p_0 \in W_3$ broadcasts the pre-prepared message for (m, v_0, sn^*). Let assume that the timer of every process expires before any process of W_9 prepared this message so that they all broadcast a (v_1, P)-VIEW-CHANGE message without mentioning m.

view v_1: Let the primary w_6 of view v_1 deliver a set of $2t_0+1$ VIEW-CHANGE messages signed by W_9, without any mention of m for sequence number sn^*. The primary puts those messages in set V_1, builds a set O_1 with a *pre-prepare* message m^{null} for sequence number sn^* and broadcasts its NEW-VIEW message with V_1, and O_1. This message is delivered by W_9, but again, because of network delays, the timer of every process expires before any process of W_9 prepared this message so that they all broadcast a (v_2, P)-VIEW-CHANGE message without mentioning m.

view v_2: The primary $p_2 \in W_3$ of view v_2 delivers a set of $2t_0+1$ VIEW-CHANGE messages signed by W_9, without mentioning m for sequence number sn^*. The primary puts those messages in set V_2, builds a set O_2 with a *pre-prepare* message m^{null} for sequence number sn^* and broadcasts its NEW-VIEW message with V_2, and O_2. Finally, the third phase is completed and every process in W_7 commits (m^{null}, v_2, sn^*).

Lemma 21. *Scenario Z and Z′ run with $t \le t_0$. Every process from W_3 commits m^{null} at view v_2*

Proof. The processes only follow the protocol and cannot expect more messages because they are supposed to decide despite possibly t_0 failed processes. □

Scenario A Consider in this scenario that $W_2\cup\{w_6\}$ are faulty and do not send any messages to members of $W_1\cup\{i\}\cup\{j\}$. Process i becomes very slow during view v_0 and until j's decision for sequence number sn^*. From the point of view of W_3, this scenario in indistiguishable from scenario Z and so W_3 behave exactly as in scenario Z. The members of $W_2\cup\{w_6\}$ behave as in scenario Z too, but omit their messages to $W_1\cup\{i\}\cup\{j\}$.

view v_0: Similar to scenario Z except that at the end, only $W_1\cup\{i\}\cup\{j\}$ prepared (m, v_0, sn^*) without committing it, they propagate the message m that they prepared to view v_1 even though they never received any message from $W_2\cup\{w_6\}$.

view v_1: Similar to scenario Z except that at the end, process j did not receive any NEW-VIEW message

view	r	$i:+$	$j:\times$	$W_1:\square\square$	$W_2:\triangle\triangle$	$W_3:\lozenge\lozenge\lozenge\lozenge$	$\omega_4:\clubsuit$	$\omega_5:\spadesuit$	$\omega_6:\heartsuit$
v_0	nv	.	.	.	.	$(v_0,m)NV$	.	.	.
v_0	nv	.	.	$(v_0,m)NV$	.	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$
v_0	pre	.	.	$pre(m)$	.	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$
v_0	pre	.	.	.	.	.	.	.	.
v_1	vc	.	.	$(v_1,\emptyset)VC$	.	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$
v_1	vc	.	.	.	.	.	.	.	$(v_1,\emptyset):V_1=\{\square\square\ \lozenge\lozenge\lozenge\lozenge\ \clubsuit\ \spadesuit\ \heartsuit\}$
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1)NV$
v_1	nv	.	.	$(v_1,\emptyset,V_1)NV$	.	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$
v_1	pre	.	.	.	.	.	.	.	.
v_2	vc	.	.	$(v_2,\emptyset)VC$	.	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$
v_2	vc	.	.	.	.	$(v_2,\emptyset):V_2=\{\square\square\ \lozenge\lozenge\lozenge\lozenge\ \clubsuit\ \spadesuit\ \heartsuit\}$	.	.	.
v_2	nv	.	.	.	.	$(v_2,\emptyset,V_2)NV$	.	.	.
v_2	nv	.	.	$(v_2,\emptyset,V_2)NV$	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$
v_2	pre	.	.	$pre(m^{null})$	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_2	pre	.	.	✓	.	✓	✓	✓	✓
v_2	com	.	.	$com(m^{null})$	.	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_2	com	.	.	✓✓	.	✓✓ : $\{\square\square\ \lozenge\lozenge\lozenge\lozenge\ \clubsuit\ \spadesuit\ \heartsuit\}$	✓✓	✓	✓✓
.	.	.	.	.	.	.	.	.	.

Table 2: Scenario Z': Without news from $\{i\}\cup\{j\}\cup\{W_2\}$, the other nodes need to commit m^{null}; they do not expect messages from $\{i\}\cup\{j\}\cup\{W_2\}$ because they could have crashed

of primary p_1 before the expiration of its timer for view v_1 but it delivers VIEW-CHANGE messages from W_8 where only $W_1\cup\{j\}$ mentioned m.

view v_2: Similar to scenario Z except that at the end, process j receives NEW-VIEW for view v_2 without any mention of m. It received all the expected messages from W_8 so that it has to decide the same value as the correct members of W_3, that is, it commits (m^{null}, v_2, sn^*).

Scenario A' The scenario A' is the same as A, except that the messages from $W_2\cup\{w_6\}$ are very slow, nodes of W_1 are Byzantine and behave with W_9 as if they did not prepare m at view v_0. From j's point of view, scenario A and A' are indistiguishable.

Lemma 22. $A \overset{W_3}{\sim} Z$. $A' \overset{W_3}{\sim} Z'$. $A \overset{j}{\sim} A'$.

Proof. The state is true by contrustion, where the logs are the same. □

Lemma 23. *In scenario A and A', process j has to commit m^{null} without receiving additional messages.*

Proof. By construction, in scenario A at each view j received $n-t_0$ messages from only correct processes. Furthermore, $t\leq t_0$ which means j has to agree with correct processes from W_3, that is commit m^{null} due to Lemmas 21 and 22. This statement applies to every scenario that is indistinguishable from A from i's point of view. □

Scenario B Consider a different scenario B where $W_2\cup\{w_4\}\cup\{w_5\}\cup\{w_6\}$ are Byzantine processes that send a commit message m to process i in view v_0, while failing to attach m in the VIEW-CHANGE message for view v_1 (as they were supposed to do). B is indistinguishable from scenario A from j's point of view so that j behaves exactly as in A.

view v_0: Similar to scenario A except that at the end, $W_2\cup\{w_4\}\cup\{w_5\}\cup\{w_6\}$ send the commit message m to process i but leave the message out of their NEW-VIEW message for triggering the creation of view v_1. $W_2\cup\{w_6\}$ never send any message to $W_1\cup\{j\}$.

view v_1: Similar to scenario A except that at the end, the view change the messages received by process j from $\{w_4\}\cup\{w_5\}$ are faulty because they send a commit message at view v_0 to i. Similar to scenario A, except that now we consider the non propagation of m to i at view v_1 by $\{w_4\}\cup\{w_5\}$ as a commission fault. The reason we consider it a commission fault is because $\{w_4\}\cup\{w_5\}$ sent a $commit(m)$ message to process i at view v_0.

view v_2: Similar to scenario A except that at the end,

view	r	$i:+$	$j:\times$	$W_1:$ □□	$W_2:$ △△	$W_3:$ ◊◊◊◊	$\omega_4:$ ♣	$\omega_5:$ ♠	$\omega_6:$ ♡
v_0	nv	.	.	.	.	$(v_0,m)NV$	.	.	.
v_0	nv	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$
v_0	pre	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m) \mid .$
v_0	pre	✓	✓	✓	.	.	.	.	✓ $\mid$.
v_1	vc	$(v_1,m)VC$	$(v_1,m)VC$	$(v_1,m)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$
v_1	vc	.	$(v_1,\emptyset):\{$ ◊◊◊◊ ♣ ♠ $\}$ $(v_1,m):\{$ □□ $\times\}$	.	.	.	.	.	$(v_1,\emptyset):V_1=\{$ △△ ◊◊◊◊ ♣ ♠ ♡ $\}$
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1)NV$
v_1	nv	.	.	.	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$
v_1	pre	.	.	.	.	.	.	.	.
v_2	vc	.	$(v_2,m)VC$	$(v_2,m)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$
v_2	vc	.	.	.	.	$(v_2,\emptyset):V_2=\{$ △△ ◊◊◊◊ ♣ ♠ ♡ $\}$	.	.	.
v_2	nv	.	.	.	.	$(v_2,\emptyset,V_2)NV$	.	.	.
v_2	nv	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$
v_2	pre	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_2	pre	.	✓	✓	✓	✓	✓	✓	✓
v_2	com	.	$pre(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_2	com	.	✓✓ $:\{\times$ □□ ◊◊◊◊ ♣ ♠ $\}$	✓✓	✓✓	✓✓ $:\{$ △△ ◊◊◊◊ ♣ ♠ ♡ $\}$	✓✓	✓✓	✓✓
.	.	.	.	.	.	.	.	.	.

Table 3: Scenario A

view	r	$i:+$	$j:\times$	$W_1:$ □□	$W_2:$ △△	$W_3:$ ◊◊◊◊	$\omega_4:$ ♣	$\omega_5:$ ♠	$\omega_6:$ ♡
v_0	nv	.	.	.	.	$(v_0,m)NV$	.	.	.
v_0	nv	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$
v_0	pre	$pre(m)$	$pre(m)$	$pre(m) \mid .$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m) \mid .$
v_0	pre	✓	✓	✓ $\mid$.	.	.	.	.	✓ $\mid$.
v_1	vc	$(v_1,m)VC$	$(v_1,m)VC$	$(v_1,m)VC \mid (v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$
v_1	vc	.	$v_1,\emptyset):\{$ ◊◊◊◊ ♣ ♠ $\}$ $(v_1,m):\{$ □□ $\times\}$	.	.	.	.	.	$(v_1,\emptyset):V_1=\{$ □□ ◊◊◊◊ ♣ ♠ ♡ $\}$
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1)NV$
v_1	nv	.	.	$(v_1,\emptyset,V_1)NV$	.	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$
v_1	pre	.	.	.	.	.	.	.	.
v_2	vc	.	$(v_2,m)VC$	$(v_2,m)VC \mid (v_2,\emptyset)VC$	.	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$
v_2	vc	.	.	.	.	$(v_2,\emptyset):V_2=\{$ □□ ◊◊◊◊ ♣ ♠ ♡ $\}$	.	.	.
v_2	nv	.	.	.	.	$(v_2,\emptyset,V_2)NV$	.	.	.
v_2	nv	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$
v_2	pre	.	$pre(m^{null})$	$pre(m^{null})$	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_2	pre	.	✓	✓	.	✓	✓	✓	✓
v_2	com	.	$pre(m^{null})$	$com(m^{null})$	.	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_2	com	.	✓✓ $:\{\times$ □□ ◊◊◊◊ ♣ ♠ $\}$	✓✓	.	✓✓ $:\{$ □□ ◊◊◊◊ ♣ ♠ ♡ $\}$	✓✓	✓✓	✓✓
.	.	.	.	.	.	.	.	.	.

Table 4: Scenario A'

process j receives NEW-VIEW v_2 without mention of m. This scenario is indistiguishable from scenario A from the point of view of j, so that it commits (m^{null}, v_2, sn^*).

Scenario B' The scenario is the same as B, except that messages from $W_2 \cup \{w_6\}$ are very slow, processes in W_1 are Byzantine and behave with W_9 as if they did not prepare m at view v_0. W_7 can be replaced by W_9 whose processes progress independently. From j's point of view, scenario B and B' are indistinguishable.

Lemma 24. $A \overset{j}{\sim} B$. $A' \overset{j}{\sim} B'$. $B \overset{j}{\sim} B'$. $A \overset{W_3}{\sim} B$. $A' \overset{W_3}{\sim} B'$.

Proof. The statement is true by construction because the logs are the same. □

view	r	$i:+$	$j:\times$	$W_1:\square\square$	$W_2:\triangle\triangle$	$W_3:\Diamond\Diamond\Diamond\Diamond$	$\omega_4:\clubsuit$	$\omega_5:\spadesuit$	$\omega_6:\heartsuit$
v_0	nv	.	.	.	.	$(v_0,m)NV$	.	.	.
v_0	nv	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$
v_0	pre	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$
v_0	pre	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark \mid .$	.	$\checkmark \mid .$	$\checkmark \mid .$	$\checkmark \mid .$
v_0	com	$com(m)$	$com(m)$	$com(m)$	$com(m) \mid .$	.	$com(m) \mid .$	$com(m) \mid .$	$com(m) \mid .$
v_0	com	$\checkmark\checkmark : \{+\times\ \square\square\ \triangle\triangle\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	.	.	.	.	.	.	.
v_1	vc	$(v_1,m)VC$	$(v_1,m)VC$	$(v_1,m)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$
v_1	vc	.	$(v_1,\emptyset) : \{\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \}$ $(v_1,m) : \{\ \square\square\ \ \times\ \}$	.	.	.	.	.	$(v_1,\emptyset) : V_1 = \{\ \triangle\triangle\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \heartsuit\ \}$
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1)NV$
v_1	nv	.	.	.	$. \mid (v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$
v_1	pre	.	.	.	.	.	.	.	.
v_2	vc	.	$(v_2,m)VC$	$(v_2,m)VC$	$. \mid (v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$
v_2	vc	.	.	.	.	$(v_2,\emptyset) : V_2 = \{\ \triangle\triangle\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	.	.	.
v_2	nv	.	.	.	.	$(v_2,\emptyset,V_2)NV$	.	.	.
v_2	nv	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$. \mid (v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$
v_2	pre	.	$pre(m^{null})$	$pre(m^{null})$	$. \mid pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_2	pre	.	$\checkmark$	$\checkmark$	$. \mid \checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
v_2	com	.	$pre(m^{null})$	$com(m^{null})$	$. \mid com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_2	com	.	$\checkmark\checkmark : \{\times\ \square\square\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \}$	$\checkmark\checkmark$	$. \mid \checkmark\checkmark$	$\checkmark\checkmark : \{\ \triangle\triangle\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark$
.	.	.	.	.	.	.	.	.	.

Table 5: Scenario B

view	r	$i:+$	$j:\times$	$W_1:\square\square$	$W_2:\triangle\triangle$	$W_3:\Diamond\Diamond\Diamond\Diamond$	$\omega_4:\clubsuit$	$\omega_5:\spadesuit$	$\omega_6:\heartsuit$
v_0	nv	.	.	.	.	$(v_0,m)NV$	.	.	.
v_0	nv	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$	$(v_0,m)NV$
v_0	pre	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$
v_0	pre	$\checkmark$	$\checkmark$	$\checkmark \mid .$	$\checkmark$	.	$\checkmark \mid .$	$\checkmark \mid .$	$\checkmark \mid .$
v_0	com	$com(m)$	$com(m)$	$com(m) \mid .$	$com(m)$	.	$com(m) \mid .$	$com(m) \mid .$	$com(m) \mid .$
v_0	com	$\checkmark\checkmark : \{+\times\ \square\square\ \triangle\triangle\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	.	.	.	.	.	.	.
v_1	vc	$(v_1,m)VC$	$(v_1,m)VC$	$(v_1,m)VC \mid (v_2,\emptyset)VC$	$(v_1,m)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$	$(v_1,\emptyset)VC$
v_1	vc	.	$(v_1,\emptyset) : \{\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \}$ $(v_1,m) : \{\ \square\square\ \ \times\ \}$	.	.	.	.	.	$(v_1,\emptyset) : V_1 = \{\ \square\square\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \heartsuit\ \}$
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1)NV$
v_1	nv	.	.	$. \mid (v_1,\emptyset,V_1)NV$	.	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$	$(v_1,\emptyset,V_1)NV$
v_1	pre	.	.	.	.	.	.	.	.
v_2	vc	.	$(v_2,m)VC$	$(v_2,m)VC \mid (v_2,\emptyset)VC$	.	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$	$(v_2,\emptyset)VC$
v_2	vc	.	.	.	.	$(v_2,\emptyset) : V_2 = \{\ \square\square\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	.	.	.
v_2	nv	.	.	.	.	$(v_2,\emptyset,V_2)NV$	.	.	.
v_2	nv	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	.	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$	$(v_2,\emptyset,V_2)NV$
v_2	pre	.	$pre(m^{null})$	$pre(m^{null})$	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_2	pre	.	$\checkmark$	$\checkmark$	.	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
v_2	com	.	$com(m^{null})$	$com(m^{null})$	.	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_2	com	.	$\checkmark\checkmark : \{\times\ \square\square\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \}$	$\checkmark\checkmark$	.	$\checkmark\checkmark : \{\ \square\square\ \Diamond\Diamond\Diamond\Diamond\ \clubsuit\ \spadesuit\ \heartsuit\ \}$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark$
.	.	.	.	.	.	.	.	.	.

Table 6: Scenario B'

Lemma 25. *In scenarios B and B', process j has to commit m^{null} without receiving additional information whereas process i commits m, leading to a disagreement.*

Proof. This is true for A and A' because of Lemma 23. This is also true for B and B' because $A \overset{j}{\sim} B$. $A' \overset{j}{\sim} B'$ according to Lemma 24. $\square$

Lemma 26. *The prolongation of B and B', where the communication recovers between i and j so that $B \overset{i,j}{\sim} B'$. Furthermore the untouchable set $U = C^B \cup C^{B'}$ is such that $|U| \geq 3t_0 - 2$.*

Proof. The proof is by construction, because $log^B_{i,j} = log^{B'}_{i,j}$.

Moreover W_3 is correct in B and B', and W_1 is correct in B while W_2 is correct in B', which means the associated untouchable set contains at least $W_1 \cup W_2 \cup W_3 \cup \{i,j\}$, that is $|U| \geq 3t_0 - 2$. (We can remark that the results still hold with the cooperation of $W_1 \cup W_2$. Indeed after B, W_2 send the log $log_{W_2}^{B'}$, while after B', W_1 send the logs $log_{W_1}^{B}$, so that the collected log are still the same.) $\square$

Theorem 8 (Theorem 3). *Without storing and exchanging additional information, in case of disagreement in an execution with $t > t_0$, no verification algorithm ensuring the detection of more than $O(1)$ Byzantine exists. A fortiori, no verification algorithm ensuring the detection of $t_0 + 1$ Byzantine exists, which means no verification algorithm can make PBFT accountable.*

Proof. The proof comes from the conjunction of Lemmas 20, 25 and 26. Such a verification algorithm could at most try to prove guilty $W_4 \cup W_5 \cup W_6$ which is a constant set of nodes independent of n. (With our scenarios, only ω_5 can be proved guilty to have sent $commit(m)$ to process i at view v_0 and sent $(v_1, \emptyset)VC$ to process j at view v_1, which is a commission fault). It follows that no verification algorithm ensuring the detection of more than $O(1)$ (and a fortiori, the detection of t_0+1) Byzantine exists. And the result follows. $\square$

Storing and exchanging the logs of the $t_0 - 1$ preceding views is not sufficient The problem is that in both scenarios B and B', i and j cannot access the set of messages V_1 proving either the culpability of W_2 in scenario B or the culpability W_1 in scenario B'. The natural transformation of PBFT, which call PBFT[2] is to enforce to transmit V_1 to justify the m^{null} proposal at view v_2. Thus the primary of view v_f does not only expect $n - t_0$ $(v_f, \emptyset)VC$ messages but expect a set $V^i_{v_f - 1}$ attached to each of these messages to motivate the absence of m. Then those attached sets can be attached to the NEW-VIEW message $(v_f, \emptyset)NV$. In our scenarios B and B', j would only decide if it receives V_1 from $p_2 \in W_3$.

But unfortunately, we can repeat the same attack scheme with additional views. Let us fix $n \geq 7$ so that $t \geq 2$ and $0 < k < t_0$. We partition $\Pi = \{i\} \cup \{j\} \cup W_1^k \cup W_2^k \cup W_3^k \cup W_4^k \cup W_5^k \cup W_6^k$ with:

- $|W_1^k| = t_0 - (k+1)$
- $|W_2^k| = t_0 - (k+1)$
- $|W_3^k| = t_0$,
- $|W_4^k| = 1$,
- $|W_5^k| = k$,
- $|W_6^k| = k$
- $|\{i,j\}| = 2$

We build similar scenarios, Z^k, Z'^k, A^k, A'^k, B^k, B'^k, with with same indistinguishability relationships ($A^k \overset{j}{\sim} A'^k \overset{j}{\sim} B^k \overset{j}{\sim} B'^k$ and $Z^k \overset{W_3^k}{\sim} A^k \overset{W_3^k}{\sim} B^k$ and $Z'^k \overset{W_3^k}{\sim} A'^k \overset{W_3^k}{\sim} B'^k$). The difference is that, now j commits m at view v_{k+1} without having received any message from the k leaders of views $v_1, v_2, \ldots, v_k$, all members of W_6^k.

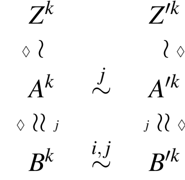

Figure 3: **- Chaining of scenarios by indinstinguishability relationships.** $X \overset{S}{\sim} Y$ means the set S, even with cooperation, is not able to dinstinguish X from Y, while $\Diamond$ is the symbol for W_3 a set of t_0 correct nodes (before being mute) who will be used in the proof, thus $X \overset{\Diamond}{\sim} Y$ means $X \overset{W_3}{\sim} Y$.

Scenarios Z^k and Z'^k are built as Z, Z'. That is, m is pre-prepared at view v_0 but not prepared, then at each view between v_1 and v_k, m^{null} is attached to the NEW-VIEW message of each successive primary of W_6^k but is not committed. Finally the primary (which is in W_3^k) of view v_{k+1}, sends a NEW-VIEW message to pre-prepare m^{null} which does not contain V_1 a necessary set of VIEW-CHANGE messages for view v_1, which is supposed to justify the decision of m^{null}.

Lemma 27. *Scenario Z^k and Z'^k run with $t \leq t_0$. Every process from W_3^k commits m^{null} at view v_{k+1}*

Proof. The processes only follow the protocol and cannot expect more messages because they are supposed to decide despite possibly t_0 failed processes. $\square$

Scenarios A^k and A'^k are built as A and A'. In both, j cannot wait for more messages from $\{i\} \cup W_2 \cup W_6$ before deciding because there could be $t \leq t_0$ Byzantine processes. And because $Z^k \overset{W_3^k}{\sim} A^k$ and $Z'^k \overset{W_3^k}{\sim} A'^k$, j decides m^{null} to ensure agreement with W_3^k. At each view v_x between v_1 and v_{k+1}, j receives $t_0 + 1$ distinct VIEW-CHANGE messages $(v_x, \emptyset, \underline{V})VC$ from $W_3^k \cup W_4^k$ and t_0 distinct VIEW-CHANGE messages $(v_x, m, \underline{V})VC$ from $\{j\} \cup W_1 \cup W_5$.

Lemma 28. $A^k \overset{W_3}{\sim} Z^k$. $A'^k \overset{W_3}{\sim} Z'^k$. $A^k \overset{j}{\sim} A'^k$.

Proof. The state is true by contrustion, where the logs are the same. $\square$

Lemma 29. *In scenario A^k and A'^k, process j has to commit m^{null} without receiving additional messages.*

Proof. By construction, in scenario A^k at each view j received $n - t_0$ messages from only correct processes. Furthermore, $t \leq t_0$ which means j has to agree with correct processes from W_3^k, that is commit m^{null} due to Lemmas 27 and 28. This statement applies to every scenario that is indistinguishable from A^k from j's point of view. $\square$

Scenarios B^k and B'^k are built as B and B'. Because $A^k \overset{j}{\sim} A'^k \overset{j}{\sim} B^k \overset{j}{\sim} B'^k$, j commits m^{null} at view v_{k+1}.

Lemma 30. $A^k \overset{j}{\sim} B^k$. $A'^k \overset{j}{\sim} B'^k$. $B^k \overset{j}{\sim} B'^k$. $A^k \overset{W_3}{\sim} B^k$. $A'^k \overset{W_3}{\sim} B'^k$

Proof. The statement is true by construction because the logs are the same. □

Lemma 31. *In scenarios B^k and B'^k, process j has to commit m^{null} without receiving additional information whereas process i commits m, leading to a disagreement.*

Proof. This is true for A^k and A'^k because of Lemma 29. This is also true for B^k and B'^k because $A \overset{j}{\sim} B$. $A' \overset{j}{\sim} B'$ according to Lemma 30. □

Lemma 32. *The prolongation of B^k and B'^k, where the communication recovers between i and j so that $B^k \overset{i,j}{\sim} B'^k$. Furthermore the untouchable set $U^k = C^{B^k} \cup C^{B'^k}$ is such that $|U^k| \geq 3t_0 - 2k$.*

Proof. The proof is by construction, because $log^{B^k}_{i,j} = log^{B'^k}_{i,j}$. Moreover W_3^k is correct in B^k and B'^k, and W_1^k is correct in B^k while W_2^k is correct in B'^k, which means the associated untouchable set contains at least $W_1^k \cup W_2^k \cup W_3^k \cup \{i, j\}$, that is $|U^k| \geq 3t_0 - 2k$. (We can remark that the results still hold with the cooperation of $W_1^k \cup W_2^k$. Indeed after B^k, W_2^k send the log $log^{B'^k}_{W_2^k}$, while after B'^k, W_1^k send the logs $log^{B^k}_{W_1^k}$, so that the collected log are still the same.) □

Theorem 9. *Let $PBFT^k$ be the algorithm that consists of PBFT where the VIEW-CHANGE messages of the last k views have to be propagated at each view. $PBFT^k$ cannot be made accountable.*

Proof. The proof comes from the conjunction of Lemmas 20, 31 and 32. Such a verification algorithm could at most try to prove guilty $W_4^k \cup W_5^k \cup W_6^k$ which is a constant set of nodes independent of n. (With our scenarios, only W_4^k can be proved guilty to have sent $commit(m)$ to process i at view v_0 and sent $(v_1, \emptyset)VC$ to process j at view v_1 which is a commission fault). It follows that no verification algorithm ensuring the detection of more than $O(k)$ (and a fortiori, the detection of $t_0 + 1$) Byzantine exists. And the result follows. □

view	r	$i:+$	$j:\times$	$W_1:\square$	$W_2:\triangle$	$W_3:\lozenge\lozenge\lozenge\lozenge$	$\omega_4:\clubsuit$	$\omega_5:\spadesuit\spadesuit$	$\omega_6:\heartsuit$	$\omega_7:\star$
v_0	nv	.	.	.	.	$(v_0,m,V_{-1},V_0)NV$	.	.	.	.
v_0	nv	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$
v_0	pre	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$
v_0	pre	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark \mid .$	.	$\checkmark$	$\checkmark \mid .$	$\checkmark \mid .$	$\checkmark \mid .$
v_0	com	$com(m)$	$com(m)$	$com(m)$	$com(m) \mid .$	.	$com(m)$	$com(m) \mid .$	$com(m) \mid .$	$com(m) \mid .$
v_0	com	$\checkmark\checkmark : \{+\times \square \triangle \clubsuit \spadesuit\spadesuit \heartsuit \star\}$	.	.	.	.	.	.	.	.
v_1	vc	$(v_1,m,V_0)VC$	$(v_1,m,V_0)VC$	$(v_1,m,V_0)VC$	$. \mid (v_1,\emptyset,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$
v_1	vc	.	$(v_1,\emptyset) : \{ \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \}$ $(v_1,m) : \{ \square \clubsuit \times \}$	.	.	.	.	.	$(v_1,\emptyset) : V_1 = \{ \triangle \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}\ V_0$	.
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1,V_0)NV$	.
v_1	nv	.	.	.	$. \mid (v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	.
v_1	pre	.	.	.	.	.	.	.	.	.
v_2	vc	$(v_2,m,V_1)VC$	$(v_2,m,V_1)VC$	$(v_2,m,V_1)VC$	$. \mid (v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_2)VC$	$(v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_1)VC$
v_2	vc	.	$(v_2,\emptyset) : \{ \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \}$ $(v_2,m) : \{ \square \clubsuit \times \}$	.	.	.	.	.	.	$(v_2,\emptyset) : V_2 = \{ \triangle \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}, V_1$
v_2	nv	.	.	.	.	.	.	.	.	$(v_2,\emptyset,V_1,V_2)NV$
v_2	nv	.	.	.	$. \mid (v_2,\emptyset,V_1,V_2)NV$	$(v_1,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$
v_2	pre	.	.	.	.	.	.	.	.	.
v_3	vc	.	$(v_3,m,V_2)VC$	$(v_3,m,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$
v_3	vc	.	.	.	.	$(v_3,\emptyset) : V_3 = \{ \triangle \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}, V_2$	.	.	.	.
v_3	nv	.	.	.	.	$(v_3,\emptyset,V_2,V_3)NV$	.	.	.	.
v_3	nv	.	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$
v_3	pre	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_3	pre	.	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
v_3	com	.	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_3	com	.	$\checkmark\checkmark : \{\times \square \lozenge\lozenge\lozenge\lozenge \clubsuit \spadesuit\spadesuit \}$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark : \{ \triangle \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark$
.	.	.	.	.	.	.	.	.	.	.

Table 7: Scenario B^2

view	r	$i:+$	$j:\times$	$W_1:\square$	$W_2:\triangle$	$W_3:\lozenge\lozenge\lozenge\lozenge$	$\omega_4:\clubsuit$	$\omega_5:\spadesuit\spadesuit$	$\omega_6:\heartsuit$	$\omega_7:\star$
v_0	nv	.	.	.	.	$(v_0,m,V_{-1},V_0)NV$	.	.	.	.
v_0	nv	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$	$(v_0,m,V_{-1},V_0)NV$
v_0	pre	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$	$pre(m)$
v_0	pre	$\checkmark$	$\checkmark$	$\checkmark \mid .$	$\checkmark$	.	$\checkmark$	$\checkmark \mid .$	$\checkmark \mid .$	$\checkmark \mid .$
v_0	com	$com(m)$	$com(m)$	$com(m) \mid .$	$com(m)$	.	$com(m)$	$com(m) \mid .$	$com(m) \mid .$	$com(m) \mid .$
v_0	com	$\checkmark\checkmark : \{+\times \square \triangle \clubsuit \spadesuit\spadesuit \heartsuit \star\}$	.	.	.	.	.	.	.	.
v_1	vc	$(v_1,m,V_0)VC$	$(v_1,m,V_0)VC$	$(v_1,m,V_0)VC \mid (v_2,\emptyset,,V_0)VC$	$(v_1,m,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$	$(v_1,\emptyset,,V_0)VC$
v_1	vc	.	$(v_1,\emptyset) : \{ \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \}$ $(v_1,m) : \{ \square \clubsuit \times \}$	.	.	.	.	.	$(v_1,\emptyset) : V_1 = \{ \square \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}\ V_0$	.
v_1	nv	.	.	.	.	.	.	.	$(v_1,\emptyset,V_1,V_0)NV$	.
v_1	nv	.	.	$. \mid (v_1,\emptyset,V_1,V_0)NV$	.	$(v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	$(v_1,\emptyset,V_1,V_0)NV$	.
v_1	pre	.	.	.	.	.	.	.	.	.
v_2	vc	$(v_2,m,V_1)VC$	$(v_2,m,V_1)VC$	$(v_2,m,V_1)VC \mid (v_2,\emptyset,V_1)VC$	$(v_2,m,V_1)VC$	$(v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_2)VC$	$(v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_1)VC$	$(v_2,\emptyset,V_1)VC$
v_2	vc	.	$(v_2,\emptyset) : \{ \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \}$ $(v_2,m) : \{ \square \clubsuit \times \}$	.	.	.	.	.	.	$(v_2,\emptyset) : V_2 = \{ \square \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}, V_1$
v_2	nv	.	.	.	.	.	.	.	.	$(v_2,\emptyset,V_1,V_2)NV$
v_2	nv	.	.	$. \mid (v_2,\emptyset,V_1,V_2)NV$	.	$(v_1,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$	$(v_2,\emptyset,V_1,V_2)NV$
v_2	pre	.	.	.	.	.	.	.	.	.
v_3	vc	.	$(v_3,m,V_2)VC$	$(v_3,m,V_2)VC \mid (v_3,\emptyset,V_2)VC$	.	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$	$(v_3,\emptyset,V_2)VC$
v_3	vc	.	.	.	.	$(v_3,\emptyset) : V_3 = \{ \square \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}, V_2$	.	.	.	.
v_3	nv	.	.	.	.	$(v_3,\emptyset,V_2,V_3)NV$	.	.	.	.
v_3	nv	.	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	.	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$	$(v_3,\emptyset,V_2,V_3)NV$
v_3	pre	.	$pre(m^{null})$	$pre(m^{null})$	.	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$	$pre(m^{null})$
v_3	pre	.	$\checkmark$	$\checkmark$	.	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark .$
v_3	com	.	$com(m^{null})$	$com(m^{null})$	.	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$	$com(m^{null})$
v_3	com	.	$\checkmark\checkmark : \{\times \square \lozenge\lozenge\lozenge\lozenge \clubsuit \spadesuit\spadesuit \}$	$\checkmark\checkmark$	.	$\checkmark\checkmark : \{ \square \lozenge\lozenge\lozenge\lozenge \spadesuit\spadesuit \heartsuit \star \}$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark$	$\checkmark\checkmark$
.	.	.	.	.	.	.	.	.	.	.

Table 8: Scenario B'^2