Polygraph: Accountable Byzantine Agreement*

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

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

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

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

I. Introduction

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

Accountability. What if, instead of *preventing* bad behavior by a party that controls too much power, we guarantee accountability, i.e., we can provide irrefutable evidence of the bad behavior and the identifier of the perpetrator of those illegal actions? Much in the way we prevent crime in the real world, we can prevent bad blockchain behavior: if the attacker has strictly less than $1/3$ of the network under their control then consensus is reached, otherwise we record sufficient information to catch the third of the network that is criminal and take remedial actions. Accountability has been increasingly discussed as a desirable property in blockchains to slash stake of cheating peers [8], [33]. The problem is to avoid suspecting correct peers while provably identifying cheating ones.

Algorithm	Msgs	Bits	Account.
PBFT [10]	$O(n^3)$	$O(\kappa \cdot n^4)$	✗
Tendermint [6]	$O(n^3)$	$O(\kappa \cdot n^3)$	✗
HotStuff [36]	$O(n^2)$	$O(\kappa \cdot n^2)$	✗
HotStuff w/o thres. sig. [22]	$O(n^2)$	$O(\kappa \cdot n^3)$	✗
DBFT binary consensus [15]	$O(n^3)$	$O(n^3)$	✗
DBFT multivalue consensus [15]	$O(n^4)$	$O(n^4)$	✗
Polygraph (Sect. V)	$O(n^3)$	$O(\kappa \cdot n^4)$	✓
Multivalue Polygraph (Sect. VI)	$O(n^4)$	$O(\kappa \cdot n^4)$	✓

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

Why is it a hard problem? As far as we know, there is no generic way of getting definitive evidence of the guilt of processes (or nodes) for all systems. Previous work introduced a method to transform any distributed system into one that is accountable but it only guarantees that faulty processes will be suspected forever if the network is partially synchronous [19]. We thus narrow down the problem to *accountable Byzantine agreement*, specifically reaching agreement when there are fewer than $n/3$ Byzantine participants or detecting at least $n/3$ Byzantine participants in case of a disagreement. Most partially synchronous Byzantine consensus protocols, like PBFT [10], Tendermint [7] or HotStuff [36], already collect forms of cryptographic evidence like signatures or certificates to guarantee agreement upon a decision. So one might think of simply recording the quorum certificates containing honest processes signatures that attest the decision to detect $n/3$ Byzantine processes in case of disagreement. In fact, we show that justifications should contain at least $\Omega(\kappa \cdot n^2)$ bits (where κ is the security parameter of the signature scheme) for a simple piggybacking extension to make any of these algorithms accountable (see Theorem IV.3).

Results. In this paper, we propose *Polygraph*, the first accountable Byzantine agreement solution. The idea is to offer accountability guarantees to the participants of the service. Intuitively, one cannot hold n servers accountable to sepa-

* A brief announcement of an earlier version of this paper appeared in DISC'20 [14] and was presented at the non-archiving workshop VDS'19 [13].

rate clients (distinct from the servers) that interact with the blockchain when more than $t \geq 2n/3$ of the servers are Byzantine. The reason is that the coalition is sufficiently large to rewrite the blockchain and prevent a client from distinguishing the response of honest servers from the response of malicious servers [12]. This was confirmed by concurrent research to ours that showed it is impossible to hold servers accountable to separate clients for any number of Byzantine participants [34].

Our solution, called *Polygraph*, ensures that in a symmetric system where all n participants are peers that take part as clients and servers in the accountable Byzantine consensus, then accountability is ensured for **any number $t \leq n$ of Byzantine participants**. Note that the problem is trivial when $t > n - 2$ as no disagreement among correct processes is possible, but otherwise Polygraph guarantees all honest participants undeniably detect at least $n/3$ Byzantine participants responsible for disagreement. Because it is resilient to any number of failures, Polygraph is particularly interesting for peer-to-peer blockchain networks. In particular, it allows to hold all peers accountable to other peers, which is appealing for consortium blockchains and shard chains [18].

We also show that Polygraph is optimal in that stronger forms of accountability are impossible. For example, we cannot guarantee agreement when $t > n/3$, even if we are willing to tolerate a failure of liveness (Theorem IV.1); and processes cannot detect even one guilty participant within a fixed time limit (e.g., prior to decision), since (intuitively) that would enable processes to determine guilt before deciding in a way that leads to disagreement. Nor can we guarantee detection of more than $n/3$ malicious users, since it takes only $n/3$ malicious users to cause disagreement and additional malicious users could simply stay mute to not be detected.

Finally, we show that Polygraph is efficient. First, its communication complexity is $O(\kappa \cdot n^4)$ bits, where n is the number of participants and κ is the security parameter of its signature scheme. This complexity is comparable to the communication complexity of state-of-the-art consensus algorithms as depicted in Table I because Polygraph simply needs to exchange signed messages received within at most the two latest previous asynchronous rounds. In particular, both the binary and the multivalue versions of Polygraph share the same asymptotic complexity as PBFT, which does not offer accountability. Second, we evaluate the performance of Polygraph in a blockchain application that thus becomes accountable. We deploy this blockchain application on 80 machines across continents and compare its performance to the Red Belly Blockchain [16]. Even though it presents some overheads compared to this non-accountable baseline, our accountable blockchain still exceeds 10,000 TPS at 80 nodes. This high performance can be attributed to the reasonable complexity of the multivalue variant of Polygraph depicted in Table I.

Roadmap. The background is given in Section II. The model and the accountable Byzantine consensus problem are presented in Section III, and impossibility results are given in Section IV. Section V describes the Polygraph protocol, which solves the accountable binary Byzantine consensus problem while Section VI generalizes the algorithm to arbitrary values. Section VII analyses empirically the Polygraph protocol in a geodistributed blockchain. blockchain. Section VIII makes general observations and Section IX concludes. The full proofs are deferred to our companion technical report [12].

II. Background and Related Work

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

A. PeerReview

Haeberlen, Kuznetsov, and Druschel [19] pioneered the idea of accountability in distributed systems. They developed a system called *PeerReview* that implemented accountability as an add-on feature for any distributed system. Each process in the system records messages in tamper-evident logs; an authenticator can challenge a process, retrieve its logs, and simulate the original protocol to ensure that the process behaved correctly. They show that in doing so, you can always identify at least one malicious process (if some process acts in a detectably malicious way). Their technique is quite powerful, given its general applicability which can be used in any (deterministic) distributed system!

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

B. 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 [8] have proposed a blockchain protocol, Casper, that provides this type of accountability guarantee. Validators try to agree on (or "finalize") a branch of k hundreds of consecutive blocks, by gathering signatures for this branch or "link" from validators jointly owning at least $2n/3$ of the deposited stake. If a validator signs multiple links at the same height, Casper uses its signatures as proofs to slash its deposited stake. This is very similar in intent to Polygraph's notion of identifying $n/3$ malicious users when there is disagreement. Like most blockchain protocols, however, Casper implicitly assumes some synchronous underlying (overlay) network and allows the blockchain to fork into a tree until

some branch is finalized. To guarantee "plausible liveness" or that Casper does not block when not enough signatures are collected to finalize a link, validators are always allowed to sign links that overlap but extend links they already signed. However, this does not guarantee that consensus terminates.

The longlasting blockchain [33] builds upon our companion technical report [12] to recover from forks by excluding guilty participants and compensating transient losses with the deposit of guilty participants. It recovers from $f = \lceil 2n/3 \rceil - 1$ failures as long as there are less than $\min(n/3, n - f)$ processes experiencing benign (e.g., crash) failures. Accountability in the context of blockchain fairness was raised by Herlihy and Moir in a keynote address [21], and the idea of "accountable Byzantine fault tolerance" has been discussed [6]. The goal in the latter case is to suggest a broadcast after the consensus in order to detect a fault by matching pre-vote and pre-commit messages of the same validator in Tendermint, but the algorithm is not detailed. Holding n servers accountable to separate clients in the Tendermint consensus algorithm [4], [7] as well as HotStuff [36] was shown possible when $t < 2n/3$ in recent research [34] but could not be achieved when $t \geq 2n/3$, which confirms our previous observations [12].

C. 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., [27], [37], [38]). Yumerefendi and Chase [38] developed an accountable system for network storage, and Michalakis et al. [31] developed an accountable peer-to-peer content distribution network. The idea of accountability appeared less explicit in many earlier systems. For example, Aiyer et al. [3] proposed the BAR model for distributed systems, which relied on incentives to ensure good behavior; one key idea was in detecting and punishing bad behaviors. Finally, Intrusion Detection Systems (e.g., [17], [23], [29] provided heuristics and techniques for detecting malicious behaviors in a variety of different systems.

D. Failure detectors

There is a connection between accountability and failure detectors. A failure detector is designed to provide each process in the system with some advice, typically a list of processes that are faulty in some manner. However, failure detectors tend to have a different set of goals. They are used during an execution to help make progress, while accountability is usually about what can be determined *post hoc* after a problem occurs. They provide advice to a process, rather than proofs of culpability that can be shared. Most of the work in this area has focused on detecting crash failures (see, e.g., [11]). There has been some interesting work extending this idea to detecting Byzantine failures [19], [20], [24], [29]. Malkhi and Reiter [29] 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 [24] 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 [30] 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 [28] 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.

III. Model and Problem

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

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

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

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

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

Verification algorithm. A verification algorithm V takes as input the state of a process and returns a set G of undeniable guilty processes, that is, every process-id of G is tagged with

an unforgeable proof of culpability. (More formally, this means that for every computationally bounded adversary, for every execution in which a process p_j is honest, for every state s generated during the execution or constructed by Byzantine users, the probability that the verification algorithm returns a set containing p_j is negligible. In practice, this will reduce to the non-forgeability of signatures.)

Accountable Byzantine agreement. The problem of Byzantine Agreement, first introduced by Pease, Shostak, and Lamport [26], 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 [32], but our companion technical report [12] shows that our accountable binary Byzantine consensus protocol ensures even a stronger validity property (if $t \leq t_0$ and an honest process decides v, then some honest process proposed v).

IV. Impossibility Results

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

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

A couple of natural questions arise regarding accountable algorithms: Can we design an algorithm that always guarantees agreement, and simply fails to terminate if there are too many Byzantine users? If so, we would trivially get accountability! Can we design an algorithm that provides earlier evidence of Byzantine behavior, even before the decision is possible? If so, we could provide stronger guarantees than are provided in this paper. Alas, neither is possible. One can find the details of the following theorems in our companion technical report.

Theorem IV.1. *In a partially synchronous system, no algorithm solves both the Byzantine consensus problem when $t < n/3$ and the agreement and validity of the Byzantine consensus problem when $t > t_0$.*

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

Theorem IV.2. *Consider an algorithm that solves consensus when $t < n/3$. There is no swift verification algorithm when $t > t_0$.*

B. Classic PBFT-like algorithms

It is interesting to see that most partially synchronous consensus algorithms already collect forms of cryptographic evidence like signatures or certificates to guarantee agreement upon decision. While this is a characteristic of PBFT [10], this is the case of modern algorithms that build upon it including Tendermint [7] and HotStuff [36]. One could naturally be tempted to reuse these signatures and certificates to piggyback practical justifications in the existing messages of the original consensus algorithms to turn them into accountable consensus algorithms. Although threshold signatures do not convey enough information to identify which process is guilty, signatures are generally sufficient. We show below that, unfortunately, this transformation cannot work.

The intuition of the proof is split in the four following steps while the full proof is deferred to our technical report. First, we define the class of 'classic' or PBFT-like consensus algorithms and denote it $\mathcal{L}$. Second, we show that HotStuff [36], PBFT [10] and Tendermint [4] belong to this class $\mathcal{L}$. (As we need to remove threshold signatures, we adopt the version of HotStuff without threshold signatures [22].) Third, we define a practical extension of PBFT-like consensus algorithms that piggybacks messages. By 'practical' we mean that piggybacked messages have a bounded staleness to prevent the justification communication to be superlinear (e.g., quadratic) in n. Finally, we prove that there exist executions leading to disagreement with different sets of Byzantine processes that correct processes cannot distinguish.

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

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

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

Theorem IV.3. *HotStuff, PBFT and Tendermint as well as all their t_0-bounded extensions are not accountable.*

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

This result (Theorem IV.3) simply shows that piggybacking t_0-bounded justifications is insufficient to make PBFT-like algorithms accountable, however, it does not mean that they cannot be transformed into an accountable algorithm. First, one could probably make PBFT-like algorithms accountable with a longer justification, exchanging $\Omega(\kappa \cdot n^2)$ times more bits, where κ is the security parameter of the signature scheme. This new extension would result in an accountable version of Tendermint, HotStuff and PBFT requiring between $\Omega(\kappa^2 \cdot n^5)$ and $\Omega(\kappa^2 \cdot n^6)$ bits. Second, transforming any of these algorithms into Polygraph (Section V) is a way of obtaining accountability with a lower complexity than the previous extension. Such a transformation would however be non-trivial because Polygraph relies on DBFT that differs from PBFT-like algorithms in various ways: every process participating in DBFT can propose a value, DBFT is signature-free and there is no view change in DBFT as there is no need to recover from a failed leader.

V. 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 [15] that was proved correct using the ByMC model checker [35] and that does not use the leader-based pattern mentioned in the proof of Theorem IV.3. Then, we focus on the key aspects that lead to accountability, specifically, the "ledgers" and "certificates." For the sake of simplicity, this section tackles the binary agreement, however, the generalization of this result to arbitrary values as well as the proof that the algorithm is correct can be found in the companion technical report.

As a notation, we indicate that a process p_i sends a message to every other process by: broadcast$(TAG, m) \rightarrow messages$, 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.)

A. 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

Algorithm 1 The Polygraph Protocol

1: bin-propose(v_i):
2: $est_i = v_i$
3: $r_i = 0$
4: $timeout_i = 0$
5: $ledger_i[0] = \emptyset$
6: **repeat:**
7: $r_i \leftarrow r_i + 1$; ▷ *increment the round number and the timeout*
8: $timeout_i \leftarrow timeout_i + 1$
9: $coord_i \leftarrow ((r_i - 1) \mod n) + 1$ ▷ *rotate the coordinator*
▷ Phase 1:
10: bv-broadcast(EST$[r_i], est_i, ledger_i[r_i - 1], i, bin_values_i[r_i]$) ▷ *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$[r_i], w) \rightarrow messages_i$
14: StartTimer($timeout_i$) ▷ *reset the timer*
15: **wait until** $(bin\_values_i[r_i] \neq \emptyset \wedge timer_i$ expired)
▷ Phase 2:
16: **if** (COORD$[r_i], w) \in messages_i$ from $p_{coord_i} \wedge w \in bin\_values_i[r_i]$) **then** ▷ *favor the coordinator*
17: $aux_i[r_i] \leftarrow \{w\}$
18: **else** $aux_i[r_i] \leftarrow bin\_values_i[r_i]$ ▷ *otherwise, use any value received*
19: $signature_i =$ sign$(aux_i[r_i], r_i, i)$ ▷ *sign the messages*
20: broadcast(ECHO$[r_i], aux_i[r_i], signature_i) \rightarrow messages_i$ ▷ *broadcast second phase message*
21: **wait until** $values_i =$ ComputeValues$(messages_i, bin_values_i[r_i], aux_i[r_i]) \neq \emptyset$
▷ Decision phase:
22: **if** $values_i = \{v\}$ **then** ▷ *if there is only one value, then adopt it*
23: $est_i \leftarrow v$
24: **if** $v = (r_i \mod 2)$ **then** ▷ *decide if value matches parity*
25: **if** no previous decision by p_i **then** decide(v)
26: **else**
27: $est_i \leftarrow (r_i \mod 2)$ ▷ *otherwise, adopt the current parity bit*
28: $ledger_i[r_i] =$ ComputeJustification$(values_i, est_i, r_i, bin_values_i[r_i], messages_i)$ ▷ *broadcast certificate*

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

it is long enough.) If a process receives a message from the coordinator, then it chooses the coordinator's value to "echo", i.e., to rebroadcast to everyone in the second phase. Otherwise, it simply echoes all the messages received in the first phase.

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

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

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

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

B. Binary value broadcast

The protocol relies in Phase 1 on a reliable broadcast routine bv-broadcast proposed before [1], which is used to ensure validity, i.e., any estimate adopted (and later decided) must have been proposed by some honest process. Moreover, it guarantees that if every honest process begins a round with the same value, then that is the only possible estimate for

Algorithm 2 Helper Components

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

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

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

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; (ii) every message delivered to an honest process was broadcast by at least $t + 1$ processes. The proof details can be found in the companion technical report.

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.

C. 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).

D. Accountability

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

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

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

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

E. Proving culpability

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

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

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

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

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

F. Analysis of the Polygraph Protocol

We show in the companion technical report that the BV-broadcast routine provides the requisite properties. This then allows us to prove the main correctness theorem:

Theorem V.1. *The Polygraph Protocol is a correct Byzantine consensus protocol guaranteeing agreement, validity, and termination.*

Accountability follows from the fact that a disagreement leads every honest process to eventually receive a certificate and a ledger that conflict:

Theorem V.2. *The Polygraph Protocol is accountable.*

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

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

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

Note that in the good case (after τ_{GST} and when $t < t_0$) Polygraph reaches consensus in three message delays. Section VI presents the multivalue generalization of Polygraph.

VI. The Mulivalue Polygraph Protocol

In this section, we discuss how to generalize the binary consensus to ensure accountable Byzantine agreement for arbitrary values. We follow the approach from [15]: First, all n processes use a reliable broadcast service to send their proposed value to all the other n processes. Then, all the processes 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 that depend on an external validity condition [16].)

A. 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.

B. Reliable broadcast with accountability

We now describe the reliable broadcast service, which is a straightforward extension of the broadcast protocol proposed by Bracha [5]. 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 [5] 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.

C. Analysis of the Multivalue Polygraph Protocol

Algorithm 3 has a message complexity of $O(n^4)$ and a bit complexity of $O(\kappa \cdot n^5)$ because it executes one reliable broadcast and spawns n parallel instances of Polygraph and because of Lemma V.3. By combining messages of parallel binary consensus instances and compressing bv-broadcast messages using existing optimizations [9], [25], the Multivalue Polygraph Protocol achieves a communication complexity of $O(\kappa \cdot n^4)$ as depicted in our companion technical report [12].

Lemma VI.1 (Multivalue Polygraph Complexity)**.** *After τ_{GST}, the multivalue Polygraph protocol has message complexity $O(n^4)$ and communication complexity $O(\kappa \cdot n^4)$, where n is the number of participants and κ is the security parameter.*

VII. 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 [16] based on DBFT [15] and the "Accountable Red Belly Blockchain" based on Polygraph. The reason is that DBFT and Polygraph are both one-shot consensus protocols while a blockchain application allows for a more realistic comparison of the performance. To this end, we implemented the (naive) Multivalue Polygraph as described in Section VI to decide blocks. We then turn the Multivalue Polygraph Protocol into a state machine replication with the classic technique [2, Fig.4] by tagging messages with their consensus instance. Finally, we build a blockchain layer using

Algorithm 3 The (Naive) Multivalue Polygraph Protocol

1: gen-propose(v_i):
2: RB-broadcast(EST, $\langle v_i, i \rangle$) $\rightarrow$ $messages_i$ ▷ *reliable broadcast value to all*
3:
4: **repeat:** ▷ *when you recieve a value from p_k, begin consensus instance k with a proposal of 1*
5: **if** $\exists\ v, k\ :\ (\text{EST}, \langle v, k \rangle) \in messages_i$ **then**
6: **if** BIN-CONSENSUS$[k]$ not yet invoked **then**
7: BIN-CONSENSUS$[k]$.bin-propose$(1) \rightarrow bin\text{-}decisions[k]_i$
8: **until** $\exists k : bin\text{-}decisions[k] = 1$ ▷ *wait until the first decision*
9:
10: **for all** k such that BIN-CONSENSUS$[k]$ not yet invoked **do** ▷ *begin consensus on the remaining instances*
11: BIN-CONSENSUS$[k]$.bin-propose$(0) \rightarrow bin\text{-}decisions[k]_i$ ▷ *for these, propose 0*
12: **wait until for all** k, $bin\text{-}decisions[k] \neq \perp$ ▷ *wait until all the instances decide*
13:
14: $j = \min\{k\ :\ bin\text{-}decisions[k] = 1\}$ ▷ *choose the smallest instance that decides 1*
15: **wait until** $\exists\ v\ :\ (\text{EST}, \langle v, j \rangle) \in messages_i$ ▷ *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, $v, \cdot, j$) and not having sent a READY message:
9: Set ℓ_i to be one of the (valid) ledgers received (READY, $v\ell, j$).
10: broadcast(READY, v, ℓ, j) ▷ *send* READY *message for v to all.*
11: **upon** receiving $n - t_0$ distinct messages (READY, $v, \cdot, j$) and not having delivered a message from j:
12: Let ℓ be one of the (valid) ledgers received (READY, v, ℓ, j).
13: deliver(v, j) ▷ *send* READY *message for v to all*

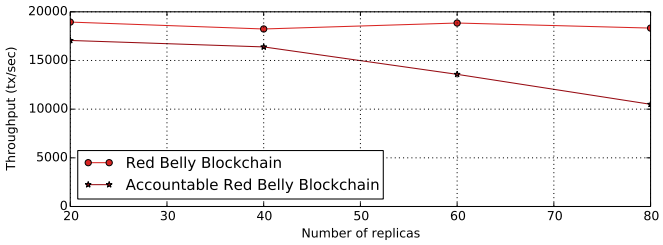

Fig. 1: The overhead of accountability in the Red Belly Blockchain with 400 byte transactions cryptographically verified with ECDSA signatures and parameters secp256k1 when deployed on 80 replicas geo-distributed in Frankfurt, Ireland, London, N. California and N. Virginia.

the Red Belly Blockchain UTXO model with signed Bitcoin-style transaction requests. We implemented Polygraph using the RSA 2048 bits signature scheme to authenticate messages.

We deployed both blockchains on up to $n = 80$ c4.xlarge AWS virtual machines located in 5 availability zones on two continents: Frankfurt, Ireland, London, North California and North Virginia. All machines issue transactions, insert transactions in their memory pool, propose blocks of 10,000 transactions, verify transaction signatures (and account integrity, and run their respective consensus algorithm with $t = t_0 = \lceil \frac{n}{3} \rceil - 1$, before storing decided blocks to non-volatile storage. Red Belly Blockchain commits tens of thousands of Bitcoin TPS on hundreds of geo-distributed processes [16].

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

VIII. Discussion

The accountability cost induced by algorithm designs. The fact that Polygraph achieves accountability efficiently while piggybacking justifications is insufficient to obtain accountable PBFT-like algorithms seems to indicate some interesting

aspects of the specific design of the DBFT algorithm [15]. In DBFT, only the messages that are sent within the current round and the two preceding rounds are sufficient to detect the guilty processes. Extending PBFT-like algorithms with piggybacking showed (Theorem IV.3) that the proof of guilt had to be built with a chain of messages that goes back to a view, as far as $\Omega(n)$ views away, where a process has prepared a value that it did not propagate. It seems that the cost of this extension is inherently induced by the view-change design common to PBFT-like algorithms.

The peer-to-peer and client-server settings. The fact that accountability cannot be offered to clients that are not playing the role of servers was both mentioned as the Zombie case [12] and [34]. Offering peer-to-peer accountability without such an assumption, like Polygraph offers, has interesting applications in consortium blockchains and shard chains [18] but also in replicated state machine (RSM). Involving clients as part of the protocol could thus strengthen this protocol accountability. For example, one can think of an RSM where a client c (resp. c') accepts a command only if it receives acknowledgments from a number k (resp. $k' \neq k$) of different servers. This RMS would thus offer different trade-offs between safety and liveness to these different clients c and c'.

IX. Conclusion

We introduced Polygraph, the first accountable Byzantine consensus algorithm. If $t < n/3$, it ensures consensus, otherwise it eventually detects users that cause disagreement. Thanks to its bounded justification size, Polygraph can be used to commit tens of thousands of blockchain transactions. We conjecture that the complexity of Multivalue Polygraph can be reduced to $O(n^3)$ messages and $O(\max(\kappa, n) \cdot n^3)$ bits by combining messages of parallel binary consensus instances and compressing bv-broadcast messages [9].

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

References

[1] Mostéfaoui A., Moumen H., and Raynal M. Signature-free asynchronous Byzantine consensus with $T < N/3$ and $O(N^2)$ messages. In *PODC*, pages 2–9, 2014.

[2] Abhinav Aggarwal and Yue Guo. A simple reduction from state machine replication to binary agreement in partially synchronous or asynchronous networks. *IACR Cryptology ePrint Archive*, 2018.

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

[4] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Dissecting tendermint. In *NETYS*, pages 166–182, 2019.

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

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

[7] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. Technical Report 1807.04938v3, arxiv, 2018.

[9] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *SRDS*, pages 191–202, 2005.

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

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

[11] 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.

[12] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2019:587, 2019.

[13] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine consensus. In *Workshop on Verification of Distributed Systems (VDS'19)*, Jun 2019. Unpublished work.

[14] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Brief announcement: Polygraph: Accountable byzantine agreement. In *DISC*, pages 45:1–45:3, 2020.

[15] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *NCA*, pages 1–8, 2018.

[16] Tyler Crain, Chris Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *S&P*, 2021.

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

[18] The eth2 upgrades. Accessed: 2020-12-12, https://ethereum.org/en/eth2/.

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

[20] Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *OPODIS*, pages 99–114, 2009.

[21] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *LICS*, pages 27–30, 2016.

[22] libhoststuff. Accessed: 2021-03-01 https://github.com/hot-stuff/libhotstuff.

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

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

[25] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *CCS*, pages 1751–1767, 2020.

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

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

[28] Jinyuan Li and David Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, pages 10–10, 2007.

[29] Dahlia Malkhi and Michael K. Reiter. Unreliable intrusion detection in distributed computations. In *CSFW*, pages 116–125, 1997.

[30] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *PODC*, pages 108–117, 2002.

[31] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *NSDI*, page 11, 2007.

[32] Roberto De Prisco, Dahlia Malkhi, and Michael K. Reiter. On k-set consensus problems in asynchronous systems. In *PODC*, pages 257–265, 1999.

[33] Alejandro Ranchal-Pedrosa and Vincent Gramoli. Blockchain is dead, long live blockchain! Technical Report 10541v2, arXiv, 2020.

[34] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. Technical Report 2010.06785, arXiv, 2020.

[35] Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain Byzantine fault tolerance. Technical Report 1909.07453v2, arXiv, 2019.

[36] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.

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

[38] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. *TOS*, 3(3):11:1–11:33, 2007.