

# As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy!

Pierre Civit<sup>a,\*</sup>, Seth Gilbert<sup>b</sup>, Vincent Gramoli<sup>c,d</sup>, Rachid Guerraoui<sup>d</sup>, Jovan Komatovic<sup>d,\*\*</sup>

<sup>a</sup> Sorbonne Université, CNRS, LIP6, 4 place Jussieu, Paris, 75005, France

<sup>b</sup> NUS Singapore, 21 Lower Kent Ridge Road, 119077, Singapore

<sup>c</sup> University of Sydney, 1 Cleveland St, Darlingtown NSW, 2008, Australia

<sup>d</sup> École Polytechnique Fédérale de Lausanne (EPFL), Rte Cantonale, Lausanne, 1015, Switzerland

## ARTICLE INFO

### Article history:

Received 31 August 2022

Received in revised form 22 June 2023

Accepted 17 July 2023

Available online 1 August 2023

### Keywords:

Distributed consensus

Accountability

Fault detection

Byzantine fault tolerance

## ABSTRACT

In a non-synchronous system with  $n$  processes, no  $t_0$ -resilient (deterministic or probabilistic) Byzantine consensus protocol can prevent a disagreement among correct processes if the number of faulty processes is  $\geq n - 2t_0$ . Therefore, the community defined the *accountable* Byzantine consensus problem: the problem of (i) solving Byzantine consensus whenever possible (e.g., when the number of faulty processes does not exceed  $t_0$ ), and (ii) allowing correct processes to obtain proofs of culpability of  $n - 2t_0$  faulty processes whenever a disagreement occurs. This paper presents *ABC*, a simple yet efficient transformation of any non-synchronous  $t_0$ -resilient (deterministic or probabilistic) Byzantine consensus protocol into its accountable counterpart. In the common case (up to  $t_0$  faults), *ABC* introduces an additive overhead of two communication rounds and  $O(n^2)$  exchanged bits. Whenever they disagree, correct processes detect culprits by exchanging  $O(n^3)$  messages, which we prove optimal. Lastly, *ABC* is not limited to Byzantine consensus: *ABC* provides accountability for other essential distributed problems (e.g., reliable and consistent broadcast).

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Byzantine consensus [40] is a fundamental problem of distributed computing. It plays a major role in state machine replication (SMR) [1,6,45,52,42,20], particular cryptographic protocols [9,34], and blockchain systems [3,14,26,33]. In brief, Byzantine consensus enables processes to agree on a common value despite Byzantine (arbitrary) failures. Concretely, the problem is defined among  $n$  processes, out of which some processes can misbehave in an arbitrary manner (these processes can crash, send different messages to different processes, fail to send some messages, etc.); processes that misbehave are said to be *faulty*, whereas non-faulty processes are said to be *correct*. The following interface is exposed:

- **input** propose( $v$ ): A process proposes a value  $v$ ; the cardinality of the values processes can be proposed can be arbitrary

(i.e., we consider multivalued consensus). Each correct process proposes exactly one value.

- **output** decide( $v'$ ): A process decides a value  $v'$ . Each correct process decides at most one value (i.e., decisions are irrevocable).

The following properties characterize the problem:

- **Agreement:** No two correct processes decide different values.
- **Validity:** If all correct processes propose the same value  $v$ , then no correct process decides a value  $v' \neq v$ .
- **Deterministic termination:** Every correct process eventually decides.
- **Probabilistic termination:** Every correct process eventually decides with probability 1.

**Definition 1** (Byzantine consensus protocol). A protocol is a  $t_0$ -resilient deterministic (resp., probabilistic) Byzantine consensus protocol if it satisfies agreement, validity and deterministic (resp., probabilistic) termination while tolerating up to  $t_0$  faulty processes.

In this paper, we are particularly interested in *non-synchronous* Byzantine consensus protocols: protocols that operate in an en-

\* Corresponding author.

\*\* Principal corresponding author.

E-mail addresses: [pierre.civit@lip6.fr](mailto:pierre.civit@lip6.fr) (P. Civit), [seth.gilbert@comp.nus.edu.sg](mailto:seth.gilbert@comp.nus.edu.sg) (S. Gilbert), [vincent.gramoli@sydney.edu.au](mailto:vincent.gramoli@sydney.edu.au) (V. Gramoli), [rachid.guerraoui@epfl.ch](mailto:rachid.guerraoui@epfl.ch) (R. Guerraoui), [jovan.komatovic@epfl.ch](mailto:jovan.komatovic@epfl.ch) (J. Komatovic).

environment without (permanent) timely communication. Dwork, Lynch and Stockmeyer proved that non-synchronous Byzantine consensus cannot be solved with  $n/3$  (or more) faulty processes [31]. By adapting their technique, we prove another negative result: the safety of Byzantine consensus can always be compromised in severely corrupted systems. (The following theorem is proven in Appendix A.)

**Theorem 1** (*Unavoidable disagreement*). *For any non-synchronous  $t_0$ -resilient (deterministic or probabilistic) Byzantine consensus protocol among  $n$  processes, there exists an execution with  $t \geq n - 2t_0$  faulty processes in which correct processes disagree (i.e., decide different values).*

A direct consequence of Theorem 1 is that no blockchain based on a non-synchronous Byzantine consensus protocol can prevent its divergence if the system is overly corrupted. Real-life consequences of such unlucky scenarios can be substantial. For example, people might lose valuable assets due to a fork created in a blockchain – such an attack is called *double-spending*.

While disagreements (and, thus, double-spending attacks) are unavoidable in severely corrupted systems (by Theorem 1), can we at least *detect* faulty processes which are responsible for disagreements? Such a detection would naturally stimulate processes to behave correctly, thus increasing the security of the entire system. Luckily, Civit et al. [24] answered this question affirmatively by introducing *accountability* to Byzantine consensus protocols. Namely, they defined the accountable Byzantine consensus problem: the problem of (i) solving Byzantine consensus whenever possible (e.g., when the number of faulty processes does not exceed some predefined threshold), and (ii) allowing correct processes to obtain proofs of culpability of (some) faulty processes whenever a disagreement occurs.

**Definition 2** (*Accountable Byzantine consensus protocol*). A protocol is a  $t_0$ -resilient deterministic (resp., probabilistic) accountable Byzantine consensus protocol if it satisfies the following two properties:

- **Byzantine consensus solvability:** In all executions with up to  $t_0$  faults, the protocol solves the Byzantine consensus problem, i.e., it satisfies agreement, validity and deterministic (resp., probabilistic) termination.
- **Accountability:** If two correct processes decide different values, every correct process eventually irrefutably detects (at least)  $n - 2t_0$  faulty processes and obtains a proof of culpability of each detected process. A proof of culpability of a process can be independently verified by a third party, and it is impossible to produce such a proof for a correct process.

Informally, a protocol is an accountable Byzantine consensus protocol if (1) it solves the Byzantine consensus problem when the system is not overly corrupted (Byzantine consensus solvability), and (2) it allows each correct process to detect (at least)  $n - 2t_0$  culprits whenever a disagreement occurs (accountability). Moreover, detection of culprits in the case of a disagreement implies an attainment of their culpability proofs. Importantly, no proof of culpability of a correct process can ever be obtained. Note that, if there are more than  $t_0$  faulty processes, correct processes might never decide and accountability is not provided in this case. In other words, accountability is guaranteed *only* in the case of a disagreement.

### 1.1. Contributions

In this paper, we present the following contributions:

1. We introduce a *generic* and *simple* transformation –  $\mathcal{ABC}$  – that maps any non-synchronous  $t_0$ -resilient (deterministic or probabilistic) Byzantine consensus protocol into its accountable counterpart. Additionally, our transformation is *efficient*: in the common case (i.e., in all executions with up to  $t_0$  faulty processes),  $\mathcal{ABC}$  introduces an additive overhead of (1) two all-to-all communication rounds, and (2)  $O(n^2)$  exchanged bits. In the case of a disagreement, correct processes achieve accountability by exchanging  $O(n^3)$  “accountability-specific” messages; we refer to this metric as the *accountability complexity*. Our transformation relies on (1) a public-key infrastructure [43,19], and (2) a threshold signature scheme [49].  $\mathcal{ABC}$  owes its simplicity and efficiency to an observation that the simple composition presented in Algorithm 1 solves the Byzantine consensus problem in a non-synchronous environment. Indeed, if the number of faults does not exceed  $t_0$ , all correct processes eventually decide the same value from Byzantine consensus (line 2). Therefore, all correct processes eventually receive  $n - t_0$  matching CONFIRM messages (line 4), and decide (line 5). The critical mechanism illustrated in Algorithm 1 is that faulty processes *must* send conflicting CONFIRM messages in order to cause a disagreement. Hence, whenever correct processes disagree, an exchange of received CONFIRM messages is sufficient for obtaining accountability.

### Algorithm 1 Intuition behind $\mathcal{ABC}$ transformation.

---

```

1: function propose( $v$ ) do
2:    $v' \leftarrow bc.propose(v)$   $\triangleright bc$  is any non-synchronous  $t_0$ -resilient Byzantine
     consensus protocol
3:   broadcast [CONFIRM,  $v'$ ]
4:   wait for [CONFIRM,  $v'$ ] from  $n - t_0$  processes
5:   return  $v'$ 

```

---

2. We prove a lower bound on the accountability complexity in a non-synchronous environment: any accountable Byzantine consensus protocol incurs  $\Omega(n^3)$  accountability complexity, with  $t_0 \in \Omega(n)$ . As a consequence,  $\mathcal{ABC}$  suffices for achieving optimal accountability complexity.
3. We show that the applicability of  $\mathcal{ABC}$  is not limited to Byzantine consensus. Specifically, we define a class of *easily-accountable agreement tasks*, and demonstrate that generalized  $\mathcal{ABC}$  transformation provides accountability for such tasks. Important distributed problems, such as Byzantine reliable and consistent broadcast [13,17], fall into the class of easily-accountable agreement tasks.

### 1.2. Related work

The work on accountability in distributed systems was pioneered in [36]: PeerReview, a generic accountability layer for distributed systems, was presented. Importantly, PeerReview does not allow correct processes to irrefutably detect faulty processes in non-synchronous environments: faulty processes might be *suspected forever* (i.e., processes strongly “believe” that the accused process is faulty, but no definitive proof is obtained), yet never conclusively detected. Hence, PeerReview does *not* suffice for accountability in non-synchronous Byzantine consensus. The formal study of Byzantine failures in the context of accountability was initiated by Haerberlen and Kuznetsov [37].

Recently, with the expansion of blockchain systems, the interest in accountable distributed protocols resurfaced once again. Polygraph [23], the first accountable Byzantine consensus protocol, was introduced by Civit et al. The Polygraph protocol is based on the DBFT consensus protocol [26] used in blockchains [27], tolerates up to  $n$  faulty processes in achieving accountability, and has the communication complexity of  $O(n^4)$  in the common case. It is

**Table 1**  
Overview of the main properties of existing accountable Byzantine consensus protocols.

Base Consensus Protocol	Communication Complexity of the Base Consensus Protocol	Communication Complexity of the Accountable Counterpart in the Common Case	Accountability Threshold	Paper
PBFT	$O(n^4)$	$O(n^4)$	$2n/3$	Sheng et al. [48]
HotStuff	$O(n^3)$	$O(n^3)$	$2n/3$	Sheng et al. [48]
Binary DBFT	$O(n^3)$	$O(n^4)$	$n$	Civid et al. [23]
Multivalued DBFT	$O(n^4)$	$O(n^4)$	$n$	Civid et al. [23]
Any	$X$	$X \cdot O(n^2)$ (or $X \cdot O(n)$ )	$n$	Civid et al. [25]
Any	$X$	$X$	$n$	<b>this paper</b>

worth observing that Polygraph worsens the communication complexity of (the binary-value version of) the DBFT base protocol by an  $O(n)$  multiplicative factor. Casper [16] is another system designed around the goal of obtaining accountability in blockchains, while Trap [47] combines accountability with game theory to increase the Byzantine fault tolerance of blockchains.

Recently, the possibility of obtaining accountability in protocols based on PBFT [20] was investigated [48]. Specifically, accountable variants of PBFT [20] and HotStuff [53] were presented; however, these protocols allow for accountability only if up to  $2n/3$  processes are faulty, which implies that their “accountability threshold” is lower than the one of Polygraph. A generic transformation  $\tau_{scr}$  of any protocol solving a Byzantine decision task into its accountable equivalent was introduced in [25]. Thus,  $\tau_{scr}$  suffices for obtaining accountability in any Byzantine consensus protocol. The drawback of the proposed transformation is its cost:  $\tau_{scr}$  worsens the communication complexity of a Byzantine consensus protocol by a multiplicative factor of  $O(n^2)$  (or  $O(n)$  in “broadcast-based” protocols<sup>1</sup>). The commonality between the discussed prior work is employing sophisticated mechanisms for achieving accountability. Indeed, the prior work achieves accountability with the help of non-trivial modifications applied to the base consensus protocol.

In contrast, we take a fundamentally different approach that allows us to treat the base consensus protocol as a “closed box”, thus obtaining simpler and more efficient accountable Byzantine consensus protocols. Table 1 compares accountable Byzantine consensus protocols obtained by  $\mathcal{ABC}$  with the existing alternatives.

**Roadmap** We present the necessary preliminaries in § 2. We devote § 3 to our  $\mathcal{ABC}$  transformation. In § 4, we prove a cubic lower bound on the accountability complexity. We define easily-accountable agreement tasks, and prove the applicability of generalized  $\mathcal{ABC}$  to such tasks in § 5. We conclude the paper in § 6. Optional appendix includes (1) a proof of Theorem 1 (Appendix A), and (2) a detailed specification of the cryptographic primitives we use (Appendix B).

## 2. Preliminaries

**System model** We consider a system  $\Psi = \{P_1, \dots, P_n\}$  of  $n$  processes that communicate by exchanging messages through a point-to-point authenticated network. Concretely, a message is a sequence of bits whose semantics is application-specific. The system is *non-synchronous*: there is no known bound that always holds on message delays and relative speed of the processes. Non-synchronous systems include:

- asynchronous systems, where the bound does not exist; and

- partially synchronous systems [31], where the bound holds only after some unknown Global Stabilization Time (GST).

Each process is assigned its *local protocol*. A local protocol of a process defines the steps to be taken during a run of the system. The collection of all local protocols is a *distributed protocol* (or simply a *protocol*).

Some processes might be *faulty*: these processes may arbitrarily deviate from their local protocol (e.g., by crashing, failing to receive or send messages, sending different messages to different processes, performing arbitrary state transitions). That is, this paper considers the Byzantine failure model [40]. If a process is not faulty, the process is *correct*. We assume that the communication is *reliable*: a message sent by a correct process to a correct process is eventually received.

An *execution* is a single run of the system. We denote by  $t$  the actual number of faulty processes in an execution.

**Best-effort broadcast** Throughout the entire paper, we extensively rely on the best-effort broadcast primitive [17]. This primitive exposes the following interface:

- **input** broadcast( $m$ ): A process broadcasts a message  $m$ ; each correct process can broadcast its messages arbitrarily many times.
- **output** deliver( $s, m'$ ): A process delivers a message  $m'$  with sender  $s$ .

The following properties are satisfied:

- **Validity**: If a correct process broadcasts a message  $m$ , then every correct process eventually delivers  $m$ .
- **Integrity**: If a correct process delivers a message  $m$  with sender  $s$  and  $s$  is correct, then  $s$  has previously broadcast  $m$ .

Best-effort broadcast can easily be implemented in an asynchronous environment assuming a reliable point-to-point network: the sender sends its message to every process. In the rest of the paper, we exclusively rely on the best-effort broadcast primitive for protocol designs. Hence, whenever we say that “a process broadcasts a message”, we mean that the process broadcasts the message using the best-effort broadcast primitive.

**Cryptographic primitives** This paragraph outlines the cryptographic primitives we use throughout the paper. For the completeness, these primitives are formally treated in Appendix B.

We assume a *public-key infrastructure* (PKI): each process is associated with its public/private key pair that is used to sign messages and verify signatures of other processes. Crucially, the following holds:

- If a correct process  $P$  signs a message  $m$ , every correct process successfully verifies that  $m$  was signed by  $P$ .

<sup>1</sup> The  $\tau_{scr}$  transformation [25] modifies each protocol by reliably-broadcasting [13, 17] its messages. Therefore, if the original protocol exclusively sends its messages to all processes (i.e., no unicasts are present),  $\tau_{scr}$  introduces an overhead of an  $O(n)$  multiplicative factor.

- If a message  $m$  is accompanied by a signature of a correct process  $P$ , then  $P$  has indeed signed  $m$ .

A message  $m$  signed by the PKI private key of a process  $P_i$  is denoted by  $m_{\sigma_i}$ .

Additionally, we assume a  $(k, n)$ -dual threshold signature scheme [49], where  $k$  is a parameter of the scheme. In this scheme, each process holds a distinct private key, and there exists a single public key. Each process  $P_i$  can use its private key to produce a partial signature of a message  $m$  by invoking  $\text{ShareSign}_i(m)$ . Moreover, a partial signature  $tsignature$  of a message  $m$  produced by process  $P_i$  could be verified by  $\text{ShareVerify}_i(m, tsignature)$  (as is the case for the PKI). Finally, a set  $S = \{tsignature_1, tsignature_2, \dots, tsignature_k\}$  of partial signatures, where  $|S| = k$  and, for each  $tsignature_i \in S$ ,  $tsignature_i = \text{ShareSign}_i(m)$ , could be combined into a single digital signature by invoking  $\text{Combine}(S)$ . A combined digital signature  $tcombined$  of message  $m$  could be verified by  $\text{Verify}(m, tcombined)$ ; if  $\text{Verify}(m, tcombined) = \top$  (true), then  $tcombined$  was obtained by combining  $k$  partial signatures of  $m$ . For obtaining the threshold signature scheme, we can either rely on (1) a trusted setup, or (2) a distributed key generation protocol [28,2]. In the latter case, the best known solution [28] incurs  $O(n^3)$  (expected) communication cost; in this paper, we assume the cost is amortized (i.e., the threshold signature scheme is setup only once for a sequence of consensus instances).

Crucially, we assume that the PKI private key of a correct process is *never* revealed (irrespective of the number of faulty processes in the system). Therefore, if a message  $m$  is signed by the PKI private key of a process  $P_i$  and  $P_i$  is correct, then the message  $m$  was certainly sent by  $P_i$ . Conversely, if the number of faulty processes exceeds  $n - k$ , the threshold private key of a process can be revealed, and a partial signature of a correct process might be forged.<sup>2</sup> In other words, we assume a *computationally-bounded adversary* for which the following holds:

- The adversary cannot forge a PKI signature of a correct process.
- If  $t \leq n - k$ , the adversary cannot forge a partial signature of a correct process.

For the full details, refer to Appendix B.

**Why cryptography?** As accountability implies a proof of culpability which can be independently verified by a third party, cryptography is irreplaceable in the accountable protocols. Indeed, cryptography is able to provide unforgeable association between any message and its sender. If such association is non-existent, one cannot verify that a proof of culpability is genuine, i.e., not manufactured by a faulty process whose goal is to falsely accuse correct processes.

**Proof of culpability** In order for correct processes to satisfy accountability (Definition 2), they need to obtain a proof of culpability of each detected process. Let us formally define a proof of culpability of a process.

**Definition 3 (Proof of culpability).** A set  $\Sigma_i$  of messages properly signed by the PKI private key of a process  $P_i$  is a *proof of culpability* of  $P_i$  if and only if there does not exist an execution such that (1)  $P_i$  is correct, and (2)  $P_i$  sends all the messages from the  $\Sigma_i$  set.

<sup>2</sup> The fact that the threshold private key of a correct process can be revealed if the system is overly corrupted allows us to not assume a trusted setup in obtaining the threshold signature scheme, i.e., the scheme might be obtained via a distributed key generation protocol (e.g., [2,28]).

Intuitively, a proof of culpability of a process is a set of messages which are properly signed (by the PKI private key) such that, if the process is correct, the process could not have sent all of those messages. Therefore, a proof of culpability undeniably proves that the concerned process is faulty (and this proof can be verified by a third party). A proof of culpability includes only messages signed by the PKI private key as the private PKI key of a correct process is *never* revealed (as opposed to the threshold private key of a correct process which might be revealed if the number of faulty processes exceeds  $n - k$ ; see the paragraph “Cryptographic primitives”). Thus, a proof of culpability of a correct process can never be obtained.

Let us give an example of a proof of culpability. Suppose that a correct process  $P_i$  never sends two conflicting PROPOSAL messages in some protocol. Hence, a set  $\Sigma_i = \{[\text{PROPOSAL}, v]_{\sigma_i}, [\text{PROPOSAL}, v' \neq v]_{\sigma_i}\}$  is a proof of culpability of  $P_i$  as  $\Sigma_i$  demonstrates that  $P_i$  has sent two conflicting PROPOSAL messages.

Lastly, if a correct process detects a process  $P_i$  and accompanies a proof of culpability  $\Sigma_i$  of  $P_i$  to the detection, we say that the process detects  $P_i$  using  $\Sigma_i$ .

**Communication complexity** In this work, as in many in distributed computing [51,4,22], we care about the number of exchanged bits of information. To this end we define a *word*: a word contains a constant number of signatures and values. Each message contains at least a single word.

For deterministic protocols, we are interested in the worst-case communication complexity.

**Definition 4 (Worst-case communication complexity).** The *worst-case communication complexity* of a deterministic protocol is the maximum number of words sent in messages by correct processes across all possible executions. If the protocol operates in the partially synchronous model, the worst-case communication complexity considers only the messages sent after GST.

We underline that Definition 4 only counts words sent *after* GST as it is known that, prior to GST, an unbounded number of words can be exchanged in any deterministic partially synchronous consensus protocol [32,51]. For probabilistic protocols, we are interested in the expected communication complexity.

**Definition 5 (Expected communication complexity).** The *expected communication complexity* of a probabilistic protocol is the maximal expected number of words sent in messages by correct processes across all possible executions, over all possible adversaries.<sup>3</sup> If the protocol operates in the partially synchronous model, the expected communication complexity considers only the messages sent after GST.

**Accountability complexity** The *accountability complexity* is a novel complexity metric designed for measuring the accountability-specific performance of protocols. We define the accountability complexity since the communication complexity is not a suitable metric for measuring the performance of accountable Byzantine consensus protocols in the degraded case (i.e., when the number of faults exceeds a predefined threshold). For example, Polygraph [24] and accountable variants of PBFT and Hotstuff [48] suffer from the infinite worst-case communication complexity in the degraded case: Byzantine processes force correct processes to constantly execute “one more round”, thus constructing an infinite execution where correct processes never decide.

<sup>3</sup> An adversary defines a probability distribution over executions of the algorithm [39].



In order to define the accountability complexity, we first define the accountability-specific messages.

**Definition 6** (*Accountability-specific message*). We denote by  $\mathcal{P}$  the set of all “used” culpability proofs across all executions of an accountable Byzantine consensus protocol. Formally:

$\mathcal{P} = \{ \text{a proof of culpability } \Sigma_i \text{ of } P_i \mid$   
 there exists an execution in which a correct  
 process detects  $P_i$  using  $\Sigma_i \}.$

A message  $m$  is an *accountability-specific* message if and only if there exists a proof of culpability  $\Sigma \in \mathcal{P}$  such that  $m \in \Sigma$ .

Intuitively, a message is accountability-specific if it is used (in any execution) by a correct process to detect a faulty process. For example, if a correct process detects a faulty process  $P_i$  using  $\Sigma_i$  in some execution, then all messages that belong to  $\Sigma_i$  are accountability-specific messages.

Finally, we are ready to formally define the accountability complexity.

**Definition 7** (*Accountability complexity*). The *accountability complexity* of an accountable Byzantine consensus protocol is the maximum number of accountability-specific messages sent by correct processes across all executions with at least two correct processes.

Intuitively, the accountability complexity represents the number of messages correct processes exchange with the goal of achieving accountability.

### 3. ABC transformation

This section presents *ABC*, our transformation that enables any non-synchronous (deterministic or probabilistic) Byzantine consensus protocol to obtain accountability. We first introduce the *accountable confirmer* problem, and give its asynchronous implementation (§ 3.1). Then, we construct our *ABC* transformation around the accountable confirmer (§ 3.2). Finally, we discuss the applicability and limitations of *ABC* (§ 3.3).

#### 3.1. Accountable confirmer

The accountable confirmer problem is a distributed problem defined among  $n$  processes, out of which some can be faulty (i.e., Byzantine). It exposes the following interface:

- **input** `submit( $v$ )`: A process submits a value  $v$ . Each correct process submits exactly one value.
- **output** `confirm( $v'$ )`: A process confirms a value  $v'$ . Each correct process confirms at most one value.
- **output** `detect( $F$ ,  $proof$ )`: A process detects processes from the set  $F$  such that  $proof$  contains a proof of culpability of each process included in  $F$ . Each correct process triggers `detect( $\cdot$ ,  $\cdot$ )` at most once.

A  $t_0^{ac}$ -resilient accountable confirmer protocol satisfies the following properties:

- **Terminating convergence**: If (1) the number of faulty processes does not exceed  $t_0^{ac}$ , and (2) all correct processes submit the

same value, then that value is eventually confirmed by all correct processes.<sup>4</sup>

- **Validity**: The value confirmed by a correct process was submitted by a correct process.
- **Accountability**: If two correct processes confirm different values, every correct process eventually irrefutably detects (at least)  $n - 2t_0^{ac}$  faulty processes and obtains a proof of culpability of each detected process.

We give an asynchronous implementation of the accountable confirmer problem in Algorithm 2. Importantly, our implementation (Algorithm 2) works under *any* computationally-bounded adversary.

*Intuition behind Algorithm 2* Consider the following algorithm (described in prose). (1) Once a correct process submits its value, it broadcasts a signed (by the PKI private key) message containing the submitted value. (2) The process waits for  $n - t_0^{ac}$  messages containing the same value. (3) Once this happens, the process confirms the value, and broadcasts the received  $n - t_0^{ac}$  messages to all processes in the system.

This simple algorithm ensures terminating convergence since, when there are up to  $t_0^{ac}$  faults and all correct processes submit the same value, all correct processes eventually receive  $n - t_0^{ac}$  messages containing the submitted value; thus, all correct processes confirm the value. As for the accountability property, if two correct processes disagree, every correct process eventually receives two conflicting sets of  $n - t_0^{ac}$  messages. Every process whose messages belong to both sets is faulty as no correct process submits multiple values.

*Description of Algorithm 2* The actual implementation (Algorithm 2) of a  $t_0^{ac}$ -resilient accountable confirmer protocol builds upon the presented intuition. We emphasize that Algorithm 2 implements a  $t_0^{ac}$ -resilient accountable confirmer for any  $t_0^{ac} \leq \lfloor n/3 \rfloor - 1$ . It takes advantage of a  $(k, n)$ -dual threshold signature scheme (see § 2, paragraph “Cryptographic primitives”), where  $k = n - t_0^{ac}$ , in order to achieve quadratic communication complexity in the common case (i.e., in all executions with up to  $t_0^{ac}$  faults). Note that an implementation which completely follows the presented intuition would suffer from a cubic communication complexity in the common case as each correct process would rebroadcast  $O(n)$  messages after confirming its value.

Each process initially broadcasts the value it submitted in a `SUBMIT` message (line 19): the `SUBMIT` message contains the value and a partial signature of the value. Moreover, the entire message is signed by the PKI private key of the sender. Once a process receives such a `SUBMIT` message, the process (1) checks whether the message is properly signed (line 7), (2) verifies the partial signature (line 21), and (3) checks whether the received value is equal to its submitted value (line 21). If all checks pass, the process stores the received partial signature (line 23) and the entire message (line 24). Once a process stores partial signatures from (at least)  $n - t_0^{ac}$  processes (line 26), the process confirms its submitted value (line 28) and informs other processes about its confirmation by combining the received partial signatures into a *light certificate* (line 29). The role of threshold signatures in our implementation is to allow every light certificate to contain a *single* signature (rather than  $n - t_0^{ac}$  signatures), thus obtaining a quadratic overall communication complexity if  $t \leq t_0^{ac}$ .

Once a process receives two conflicting light certificates (line 34), the process concludes that correct processes might have con-

<sup>4</sup> Note that it is *not* guaranteed that any correct process confirms a value if correct processes submit different values (even if the number of faulty processes does not exceed  $t_0^{ac}$ ).

firmed different values. If the process has already confirmed its value, the process broadcasts the set of (at least)  $n - t_0^{ac}$  properly signed  $[SUBMIT, v, *]$  messages (line 35), where  $v$  is the value confirmed (and submitted) by the process; such a set of messages is a *full certificate* for value  $v$ . Finally, once a process receives two conflicting full certificates (line 40), the process obtains proof of culpability of (at least)  $n - 2t_0^{ac}$  faulty processes (line 48), which ensures accountability. Indeed, each full certificate contains  $n - t_0^{ac}$  properly signed messages: every process whose messages belong to the conflicting full certificates is faulty and these messages represent a proof of its misbehavior. (Recall that no faulty process ever obtains the PKI private key of a correct process.)

#### Definitions for Algorithm 2.

- 1) A combined digital signature  $tsig$  is a *valid light certificate* for value  $v$  if and only if  $Verify(v, tsig) = T$ .
- 2) A set  $S$  of properly signed  $[SUBMIT, v, *]_{\sigma_v}$  messages is a *valid full certificate* for value  $v$  if and only if:
  - a)  $|S| \geq n - t_0^{ac}$
  - b) Each message  $m$  is sent (i.e., signed) by a distinct process.
- 3) Let  $tsig_v$  be a valid light certificate for value  $v$  and let  $tsig_{v'}$  be a valid light certificate for value  $v'$ .  $tsig_v$  *conflicts* with  $tsig_{v'}$  if and only if  $v \neq v'$ .
- 4) Let  $S_v$  be a valid full certificate for value  $v$  and let  $S_{v'}$  be a valid full certificate for value  $v'$ .  $S_v$  *conflicts* with  $S_{v'}$  if and only if  $v \neq v'$ .
- 5) Let  $(m_1, m_2)$  be a pair of messages properly signed by some process  $P_i$ .  $(m_1, m_2)$  is a *proof of culpability* of  $P_i$  if and only if:
  - a)  $m_1 = [SUBMIT, v, share_1]_{\sigma_i}$ , and
  - b)  $m_2 = [SUBMIT, v', share_2]_{\sigma_i}$ , and
  - c)  $v \neq v'$ .

**Theorem 2.** Algorithm 2 is an asynchronous  $t_0^{ac}$ -resilient accountable confirmer protocol safe under any computationally-bounded adversary, where  $t_0^{ac} \leq \lceil n/3 \rceil - 1$ , with:

- $O(n^2)$  worst-case communication complexity in the common case (i.e., when  $t \leq t_0^{ac}$ ), and
- $O(n^3)$  SUBMIT messages being sent by correct processes.

**Proof.** We start by proving the terminating convergence and validity properties. If  $t \leq t_0^{ac}$  and all correct processes submit the same value  $v$ , the rule at line 26 eventually triggers at every correct process. Since every correct process confirms only the value it has submitted (line 28), terminating convergence and validity are satisfied by Algorithm 2.

Next, let us prove accountability. Let a correct process  $P_i$  confirm a value  $v$  and let another correct process  $P_j$  confirm a value  $v' \neq v$ . The rule at line 34 is eventually triggered at each correct process that confirms a value. Once the rule is triggered at  $P_i$  and  $P_j$ , these processes broadcast their full certificates (line 35). Eventually, the rule at line 40 is triggered at each correct process, which ensures accountability. Indeed, every process whose SUBMIT messages belong to both conflicting full certificates is detected (line 42 - line 46); moreover, such a process is indeed faulty since no correct process submits different values, i.e., no correct process ever sends different SUBMIT messages.

Finally, we prove the claimed complexity:

- If  $t \leq t_0^{ac}$ , the communication complexity of the algorithm is quadratic because (1) light certificates are sent only once and they contain a single threshold signature, and (2) no correct process broadcasts a full certificate as no two conflicting light certificates can be produced.
- Each correct process sends  $n$  SUBMIT messages at line 19. Moreover, each correct process includes (at most)  $n$  SUBMIT messages in each FULL-CERTIFICATE message it sends (line 35).

Therefore, each correct process sends (at most)  $O(n) + O(n^2) = O(n^2)$  SUBMIT messages, which implies that (at most)  $n \cdot O(n^2) = O(n^3)$  SUBMIT messages are sent by all correct processes.

The theorem holds.  $\square$

#### 3.2. $ABC$ : Byzantine consensus + accountable confirmer = accountable Byzantine consensus

We now present our  $ABC$  transformation (Algorithm 3), the main contribution of our work.  $ABC$  is built on the observation that any non-synchronous (deterministic or probabilistic) Byzantine consensus protocol paired with the accountable confirmer solves the accountable Byzantine consensus problem. Specifically, we prove that Algorithm 3 solves the accountable Byzantine consensus problem, which implies that  $ABC$  indeed enables non-synchronous Byzantine consensus protocols to obtain accountability.

The following theorem proves that the  $ABC$  transformation (Algorithm 3) is correct.

**Theorem 3 (Correctness of  $ABC$ ).** Let  $bc$  be a non-synchronous  $t_0$ -resilient deterministic (resp., probabilistic) Byzantine consensus protocol, where  $t_0 \leq \lceil n/3 \rceil - 1$ . Let  $abc$  be a protocol obtained by applying  $ABC$  (Algorithm 3) to  $bc$ . Then,  $abc$  is a non-synchronous  $t_0$ -resilient deterministic (resp., probabilistic) accountable Byzantine consensus protocol which tolerates the same computationally-bounded adversary as  $bc$ .

**Proof.** Consider an execution where  $t \leq t_0$ . Let  $bc$  be a deterministic (resp., probabilistic) Byzantine consensus protocol. All correct processes eventually decide (resp., decide with probability 1) the same value  $v$  from Byzantine consensus at line 9 by deterministic (resp., probabilistic) termination and agreement of Byzantine consensus. Moreover, if all correct processes have proposed the same value (line 7), then the proposed value is indeed  $v$  (ensured by validity of Byzantine consensus). Terminating convergence of accountable confirmer ensures that all correct processes eventually confirm  $v$  (line 11) and decide from accountable Byzantine consensus (line 12). Hence, Algorithm 3 satisfies deterministic (resp., probabilistic) termination, agreement and validity.

If correct processes disagree (i.e., decide different values at line 12), then these processes have confirmed different values from the accountable confirmer (line 11). Thus, every correct process detects (and obtains proofs of culpability of)  $n - 2t_0$  processes at line 13 (by the accountability property of the accountable confirmer). Finally, as Algorithm 2 solves the accountable confirmer problem under any computationally-bounded adversary (by Theorem 2),  $abc$  is safe under the same computationally-bounded adversary as  $bc$ .  $\square$

Theorem 3 shows that Algorithm 3 is an (asynchronous) implementation of an accountable Byzantine consensus protocol. Therefore, any Byzantine consensus protocol can be transformed into an accountable one by “inserting” that protocol into the composition presented by Algorithm 3 (line 5). Importantly, the upper bound on tolerated Byzantine processes for deterministic and probabilistic protocols is  $\lceil n/3 \rceil - 1$  [31,12], which implies that the  $ABC$  transformation is applicable to Byzantine consensus protocols with every possible resilience. Furthermore, we note that  $ABC$  provides maximal resilience against disagreement: if  $ABC$  is applied to a  $t_0$ -resilient Byzantine consensus protocol, a disagreement in the resulting protocol cannot occur with less than  $n - 2t_0$  faulty processes.

**Algorithm 2**  $t_0^{ac}$ -Resilient Accountable Confirmer: Code for process  $P_i$ .

---

```

1: Implements:
2:    $t_0^{ac}$ -Resilient Accountable Confirmer, instance  $ac$ 
3: Uses:
4:   Best-Effort Broadcast [17], instance  $beb$ 
5:    $(k, n)$ -Threshold Signature Scheme, where  $k = n - t_0^{ac}$ 
6: Rules:
7:   1) Any SUBMIT message that is not properly signed is discarded.
8:   2) Rules at lines 26, 34 and 40 are activated at most once.
9: upon event  $\langle ac, \text{Init} \rangle$  do
10:   $value_i \leftarrow \perp$ 
11:   $confirmed_i \leftarrow \text{false}$ 
12:   $from_i \leftarrow \emptyset$ 
13:   $lightCertificate_i \leftarrow \emptyset$ 
14:   $fullCertificate_i \leftarrow \emptyset$ 
15:   $obtainedLightCertificates_i \leftarrow \emptyset$ 
16:   $obtainedFullCertificates_i \leftarrow \emptyset$ 
17: upon event  $\langle ac, \text{Submit} | v \rangle$  do
18:   $value_i \leftarrow v$ 
19:  trigger  $\langle beb, \text{Broadcast} | [\text{SUBMIT}, v, \text{ShareSign}_i(v)]_{\sigma_i} \rangle$ 
20: upon event  $\langle beb, \text{Deliver} | P_j, [\text{SUBMIT}, value, share]_{\sigma_j} \rangle$  do
21:  if  $\text{ShareVerify}_j(value, share) = \top$  and  $value = value_i$  and  $P_j \notin from_i$  then
22:     $from_i \leftarrow from_i \cup \{P_j\}$ 
23:     $lightCertificate_i \leftarrow lightCertificate_i \cup \{share\}$ 
24:     $fullCertificate_i \leftarrow fullCertificate_i \cup \{[\text{SUBMIT}, value, share]_{\sigma_j}\}$ 
25:  end if
26: upon  $|from_i| \geq n - t_0^{ac}$  do
27:   $confirmed_i \leftarrow \text{true}$ 
28:  trigger  $\langle ac, \text{Confirm} | value_i \rangle$ 
29:  trigger  $\langle beb, \text{Broadcast} | [\text{LIGHT-CERTIFICATE}, value_i, \text{Combine}(lightCertificate_i)] \rangle$ 
30: upon event  $\langle beb, \text{Deliver} | P_j, [\text{LIGHT-CERTIFICATE}, value_j, lightCertificate_j] \rangle$  do
31:  if  $lightCertificate_j$  is a valid light certificate for  $value_j$  then
32:     $obtainedLightCertificates_i \leftarrow obtainedLightCertificates_i \cup \{[\text{LIGHT-CERTIFICATE}, value_j, lightCertificate_j]\}$ 
33:  end if
34: upon  $certificate_1, certificate_2 \in obtainedLightCertificates_i$  where  $certificate_1$  conflicts with  $certificate_2$  and  $confirmed_i = \text{true}$  do
35:  trigger  $\langle beb, \text{Broadcast} | [\text{FULL-CERTIFICATE}, value_i, fullCertificate_i] \rangle$ 
36: upon event  $\langle beb, \text{Deliver} | P_j, [\text{FULL-CERTIFICATE}, value_j, fullCertificate_j] \rangle$  do
37:  if  $fullCertificate_j$  is a valid full certificate for  $value_j$  then
38:     $obtainedFullCertificates_i \leftarrow obtainedFullCertificates_i \cup \{fullCertificate_j\}$ 
39:  end if
40: upon  $certificate_1, certificate_2 \in obtainedFullCertificates_i$  where  $certificate_1$  conflicts with  $certificate_2$  do
41:   $proof = \emptyset$ 
42:  for each process  $P_i$  such that  $P_i$ 's messages belong to both  $certificate_1$  and  $certificate_2$ :
43:     $m_1 \leftarrow$  the SUBMIT message signed by  $P_i$  which belongs to  $certificate_1$ 
44:     $m_2 \leftarrow$  the SUBMIT message signed by  $P_i$  which belongs to  $certificate_2$ 
45:     $\Sigma_i \leftarrow (m_1, m_2)$ 
46:     $proof = proof \cup \{\Sigma_i\}$ 
47:   $F \leftarrow$  the set of processes detected using  $proof$ 
48:  trigger  $\langle ac, \text{Detect} | F, proof \rangle$ 

```

---

▷  $P_i$  submits a value

▷  $P_i$  confirms a value

▷  $P_i$  detects faulty processes

---

**Algorithm 3**  $\mathcal{ABC}$  Transformation: Code for process  $P_i$ .

---

```

1: Implements:
2:    $t_0$ -Resilient Accountable Byzantine Consensus, instance  $abc$ 
3: Uses:
4:   ▷ Deterministic or probabilistic Byzantine consensus protocol to be transformed
5:    $t_0$ -Resilient Byzantine Consensus, instance  $bc$ 
6:    $t_0$ -Resilient Accountable Confirmer implemented by Algorithm 2, instance  $ac$ 
7: upon event  $\langle abc, \text{Propose} | proposal \rangle$  do
8:  trigger  $\langle bc, \text{Propose} | proposal \rangle$ 
9: upon event  $\langle bc, \text{Decide} | decision \rangle$  do
10:  trigger  $\langle ac, \text{Submit} | decision \rangle$ 
11: upon event  $\langle ac, \text{Confirm} | confirmation \rangle$  do
12:  trigger  $\langle abc, \text{Decide} | confirmation \rangle$ 
13: upon event  $\langle ac, \text{Detect} | F, proof \rangle$  do
14:  trigger  $\langle abc, \text{Detect} | F, proof \rangle$ 

```

---

▷ Proposal

▷ Decision

▷ Detection

---

Next, we show that  $\mathcal{ABC}$  does not worsen the communication complexity in the common case of sup-quadratic Byzantine consensus protocols and induces a cubic accountability complexity.

**Theorem 4.** Let  $bc$  be a  $t_0$ -resilient deterministic (resp., probabilistic) Byzantine consensus protocol, where  $t_0 \leq \lceil n/3 \rceil - 1$ , with the worst-case (resp., expected) communication complexity  $cc$  in the common case

(with up to  $t_0$  faults). Let  $abc$  be a protocol obtained by applying  $\mathcal{ABC}$  to  $bc$ . Then,  $abc$  has the worst-case (resp., expected) communication complexity  $\max(cc, O(n^2))$  in the common case and  $O(n^3)$  accountability complexity.

**Proof.** As the communication complexity of the accountable confirmer is  $O(n^2)$  in the common case (by Theorem 2), the

worst-case (resp., expected) communication complexity of  $abc$  is  $\max(cc, O(n^2))$  in the common case. Moreover, as the only accountability-specific messages sent by  $abc$  are the SUBMIT messages of the accountable confirmer and  $O(n^3)$  SUBMIT messages are sent by correct processes (by Theorem 2), the accountability complexity is  $O(n^3)$ .  $\square$

Lastly, we remark that  $ABC$  does not worsen the communication complexity of any (1) deterministic Byzantine consensus protocol, or (2) (possibly probabilistic) Byzantine consensus protocol safe under a strongly adaptive adversary. Namely, Dolev and Reischuk [29] proved that any deterministic Byzantine consensus protocol incurs a quadratic worst-case communication complexity in the common case. Similarly, Abraham et al. [5] showed that the quadratic lower bound for expected communication complexity holds against an adaptive adversary. However, we underline that  $ABC$  worsens the communication complexity of partially synchronous probabilistic Byzantine consensus protocols with subquadratic expected complexity (e.g., [46]).<sup>5</sup>

### 3.3. Discussion

In this subsection, we discuss  $ABC$ 's applicability to different variants of the consensus problem, as well as some of  $ABC$ 's limitations.

*ABC's applicability to different variants of Byzantine consensus* The (accountable) Byzantine consensus problem (as defined in § 1) specifies the validity property, which ensures that, if all correct processes propose the same value, then only that value could be decided by a correct process. In the literature, this is not the only variant of the validity property; the variant we use is traditionally called *strong validity*. Other most notable variants of the validity property include:

- *Weak Validity*: If all processes are correct and if a correct process decides value  $v$ , then  $v$  is proposed by a (correct) process [44,53,15].
- *External Validity*: A value decided by a correct process satisfies a predefined *valid* predicate [18].

Importantly, the correctness of  $ABC$  does not depend on a specific variant of the validity property: the “connection” between proposed values and the decided value is preserved by  $ABC$ . In other words, an accountable Byzantine consensus protocol which is a product of  $ABC$  satisfies the *same* validity property as the original consensus protocol.

*Limitations of ABC* We now list a few limitations of  $ABC$ :

1.  $ABC$  is not optimized for the best-case scenarios: It is possible to devise Byzantine consensus protocols that exhibit  $o(n^2)$  communication in some favorable scenarios. For instance, HotStuff [53] achieves only linear communication if no faulty processes exist and the execution is synchronous from the very beginning. However, this linear communication is lost when  $ABC$  is applied to HotStuff as processes “always” exchange the SUBMIT messages, which leads to the inevitable  $O(n^2)$  communication cost.
2.  $ABC$  (more precisely, our implementation of the accountable confirmer) uses threshold signatures [41] to obtain  $O(n^2)$  communication complexity in the common case. As we have

already mentioned in § 2, we do not need to assume a trusted setup to obtain a threshold signature scheme: an asynchronous distributed key generation (ADKG) protocol [2], executed on top of a PKI setup, provides a threshold signature scheme. Importantly, we are not aware of an ADKG protocol whose communication complexity is quadratic. Therefore,  $ABC$  introduces a quadratic overhead only if (1) a trusted setup is assumed, or (2) the communication cost of an ADKG protocol is amortized. An alternative is to use compressed  $\Sigma$  protocols [8], which allow us to obtain a *transparent* threshold signature scheme (without trusted setup) with a logarithmic overhead per threshold signature: communication overhead of  $ABC$  would be  $O(n^2 \cdot \log(n))$ .

In practice, it is worth considering multi-signatures [30,10] instead of the aforementioned threshold signatures. Multi-signatures have an accompanying bit-mask of  $n$  bits. In summary, if  $\kappa$  denotes the size of a signature (usually,  $\kappa = 256$ ), the communication overhead of  $ABC$  would be:

- $O(\kappa \cdot n^2 + n^3)$  in the case of multi-signatures;
- $O(\kappa \cdot n^2)$  in the case of threshold signatures (assuming a trusted setup or a cost-amortized ADKG); and
- $O(\kappa \cdot n^2 \cdot \log(n))$  in the case of compressed  $\Sigma$ -protocols.

A formal treatment of all of the aforementioned cryptographic primitives is available in Appendix B.

## 4. Lower bound on accountability complexity

In this section, we prove that any non-synchronous  $t_0$ -resilient accountable Byzantine consensus protocol incurs cubic accountability complexity (when  $t_0 \in \Omega(n)$ ). Throughout the entire subsection, we fix any non-synchronous  $t_0$ -resilient (deterministic or probabilistic) accountable Byzantine consensus protocol  $abc$ . Without loss of generality, we assume that  $n = 3t_0 + 1$ .

We prove the lower bound by showing that the accountability complexity of  $abc$  is  $\Omega(n^3)$ . Specifically, we prove that there exists an execution  $\mathcal{E}$  such that correct processes send  $\Omega(n^3)$  accountability-specific messages in  $\mathcal{E}$ .

*Execution  $\Lambda$*  First, we construct a specific (finite) execution  $\Lambda$ . We fix three disjoint groups of processes: (1) group  $A = \{a_1, a_2, \dots, a_{t_0}\}$ , where  $|A| = t_0$ , (2) group  $B = \{b_1, b_2, \dots, b_{t_0}, b_{t_0+1}\}$ , where  $|B| = t_0 + 1 = n - 2t_0$ , and (3) group  $C = \{c_1, c_2, \dots, c_{t_0}\}$ , where  $|C| = t_0$ . Throughout the entire subsection, we rely on the aforementioned groups.

Since  $abc$  solves Byzantine consensus if there are up to  $t_0$  faults, the following two infinite executions exist:

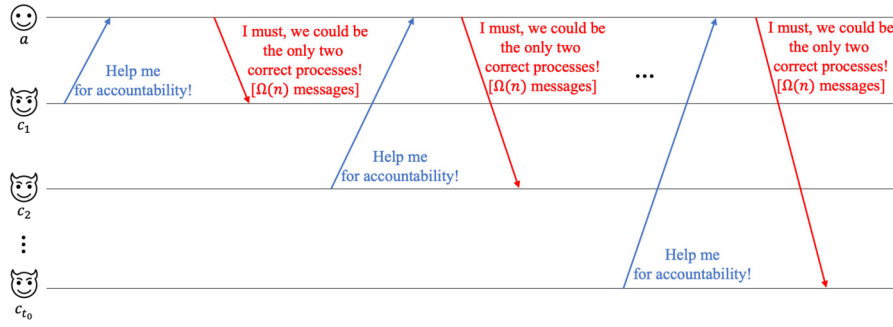
1.  $e_1$ : All processes from the group  $C$  are faulty, and these processes are silent throughout the entire execution (i.e., they send no messages). Moreover, all processes from the  $A \cup B$  set propose the same value  $v$ . Since there are  $t_0$  faulty processes (as  $|C| = t_0$ ),  $abc$  ensures that all processes from the  $A \cup B$  set eventually decide the same value  $v$  (by Byzantine consensus solvability of  $abc$ ) by some global time  $t_1$ .
2.  $e_2$ : All processes from the group  $A$  are faulty, and these processes are silent throughout the entire execution. Moreover, all processes from the  $B \cup C$  set propose the same value  $v' \neq v$ . As there are  $t_0$  faulty processes (since  $|A| = t_0$ ),  $abc$  ensures that all processes from the  $B \cup C$  set eventually decide the value  $v' \neq v$  by some global time  $t_2$ .

The existence of the executions  $e_1$  and  $e_2$  allows us to devise another infinite execution  $e$ , where:

- Processes from the group  $A$  and processes from the group  $C$  are correct, whereas processes from the group  $B$  are faulty.

<sup>5</sup> We emphasize that [46] assumes a static adversary, thus allowing itself to circumvent the quadratic lower bound [5].



Fig. 1. Intuition behind the execution  $e_a$ .

Moreover, all processes from the group A propose  $v$ , and all processes from the group C propose  $v' \neq v$ .

- The processes from the group B behave towards the processes from the group A as in execution  $e_1$ , and the processes from the group B behave towards the processes from the group C as in  $e_2$ . Moreover, if an event  $\epsilon$  (e.g., reception of a message, sending of a message, local computation) occurs at global time  $t_\epsilon$  in  $e_1$  or  $e_2$ , then  $\epsilon$  occurs at the same time  $t_\epsilon$  in execution  $e$ .
- All messages between processes from the group A and the group C are delayed until after time  $T_0 = \max(t_1, t_2)$ .

Importantly, execution  $e$  is indistinguishable from execution  $e_1$  to processes from the group A, which implies that all processes from the group A decide value  $v$  by time  $t_1$ . Similarly, all processes from the group C decide value  $v' \neq v$  by time  $t_2$ . Thus, correct processes disagree in  $e$ .

We denote by  $\Lambda$  the prefix of execution  $e$  until time  $T_0 = \max(t_1, t_2)$ .<sup>6</sup> Note that the following holds for  $\Lambda$ :

- All processes from the group A decide  $v$  in  $\Lambda$ .
- All processes from the group C decide  $v' \neq v$  in  $\Lambda$ .
- No message is exchanged between any two processes ( $a \in A, c \in C$ ).

The first intermediate result we prove is that no correct process  $p \in A \cup C$  can obtain a proof of culpability of any process from the messages it received in  $\Lambda$ . Informally, the reason is that all processes seem correct in the eyes of a correct process; recall that the groups A and C do not communicate with each other in  $\Lambda$ .

**Lemma 1.** Consider a process  $p \in A \cup C$ . Given the messages  $p$  receives in  $\Lambda$ ,  $p$  is unable to construct a proof of culpability of any process.

**Proof.** Without loss of generality, let  $p \in A$ . Recall that  $\Lambda$  is indistinguishable from  $e_1$  until time  $T_0$  to  $p$ . Assume that, by the means of contradiction,  $p$  obtains a proof of culpability of some process from the messages received in  $\Lambda$ .

We now construct an infinite execution  $e_1^*$  by relying on  $e_1$ :

1. All processes are correct in  $e_1^*$ .
2. All messages sent by the processes from the group C to any process from the groups A or B are delayed until after time  $T_0$ .
3. The execution  $e_1^*$  unfolds in the exact same way as  $e_1$  until time  $T_0$ .

Due to the construction of  $e_1^*$ ,  $p$  cannot distinguish  $e_1^*$  (until time  $T_0$ ) from  $\Lambda$ . Thus,  $p$  obtains a proof of culpability of some process in  $e_1^*$ . However, this is impossible as all processes are correct.  $\square$

*Intuition* Now that we have designed the finite execution  $\Lambda$ , we can present the intuition behind the construction of  $\mathcal{E}$ . Let us fix any process  $a \in A$ .

First, note that there exists a continuation  $e_a^1$  of  $\Lambda$  in which (1) only  $a$  and  $c_1 \in C$  are correct, and (2) these two processes do not receive messages from any other process after  $T_0$ . Importantly, all processes in  $A \setminus \{a\}$  are silent after  $T_0$ ; note that their behavior is correct, except that sent messages which are not received by  $T_0$  are omitted (this will play an important role in the conclusion of the proof intuition). As Lemma 1 proves,  $c_1$  is unable to build any proof of culpability given the messages it has received in  $\Lambda$ . As accountability must be satisfied in  $e_a^1$ ,  $a$  must help  $c_1$  in obtaining (at least)  $t_0 + 1$  proofs of culpability. Therefore,  $a$  must send (at least)  $t_0 + 1 = \Omega(n)$  accountability-specific messages to  $c_1$ . Let  $T_1$  denote the time by which  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_1$ .

However, if  $a$  receives a message from  $c_2 \in C$  after  $T_1$ ,  $a$  must help  $c_2$  in satisfying accountability. Indeed,  $a$  cannot rely on  $c_1$  helping  $c_2$  as  $c_1$  might be faulty. Thus,  $a$  needs to send  $\Omega(n)$  accountability-specific messages to  $c_2$ . Following the same logic, we construct a finite execution  $e_a$  in which  $a$  sends  $\Omega(n)$  accountability-specific messages to each process  $c \in C$ ; hence,  $a$  sends  $\Omega(n^2)$  accountability-specific messages in  $e_a$ . We denote by  $T_a$  the time by which  $a$  sends  $\Omega(n^2)$  messages in  $e_a$ . Fig. 1 gives a visual depiction of the intuition behind the design of  $e_a$ .

At this point, for every process  $a \in A$ , we have an execution  $e_a$  in which (1)  $a$  sends  $\Omega(n^2)$  accountability-specific messages, and (2) all other processes from the group A are silent. Therefore, we can “merge” all of these executions into  $\mathcal{E}$  in the following manner:

1. Only processes from the group A are correct. All other processes (i.e.,  $B \cup C$ ) are faulty.
2. Message between processes from the group A which are not received by  $T_0$  (i.e., in  $\Lambda$ ) are delayed until after  $\max(T_{a_1}, T_{a_2}, \dots, T_{a_{t_0}})$ .
3. Each process  $c \in C$  behaves towards each process  $a \in A$  as it does in  $e_a$ .

As no process  $a \in A$  can distinguish  $\mathcal{E}$  from  $e_a$  until  $T_a$ ,  $a$  sends  $\Omega(n^2)$  messages in  $\mathcal{E}$ . Thus,  $\Omega(n^3)$  accountability-specific messages are sent in  $\mathcal{E}$ , which suffices for proving the lower bound.

*Construction of  $\mathcal{E}$  (part 1):* In the first part of the construction, we build an execution  $e_a$  in which (1) only a fixed process  $a \in A$  is correct, and (2)  $a$  sends (at least)  $t_0 + 1 \in \Omega(n)$  accountability-specific messages to each process from the group C. Thus,  $a$  sends  $a$

<sup>6</sup> Recall that  $e$  is an infinite execution. On the other hand,  $\Lambda$  is a finite execution.








	$T_0$	$T_1$
 $a$ <ul style="list-style-type: none"> <li>decides <math>v</math></li> <li>obtains no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>receives messages only from <math>c_1</math> (and itself)</li> <li>sends <math>\Omega(n)</math> accountability-specific messages to <math>c_1</math> by <math>T_1</math></li> </ul>	
 $c_1$ <ul style="list-style-type: none"> <li>decides <math>v'</math></li> <li>obtains no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>receives messages only from <math>a</math> (and itself)</li> <li>achieves accountability</li> </ul>	
 $c_2$ <ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	
 $c_3$ <ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	
 $\vdots$		
 $c_{t_0}$ <ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	
 all other processes	<ul style="list-style-type: none"> <li>silent</li> </ul>	
$\pi_a^1$		

Fig. 2. Execution  $e_a^1$ .

quadratic number of accountability-specific messages in  $e_a$ . As constructing  $e_a$  is non-trivial, we construct  $e_a$  incrementally (through a sequence of steps).

**Step 1:** Construction of an infinite execution  $e_a^1$  in which  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_1$ .

We construct  $e_a^1$  by relying on the previously described execution  $\Lambda$ . Specifically, we construct  $e_a^1$  as follows:

- Only processes  $a$  and  $c_1$  are correct.
- We construct the prefix  $\pi_a^1$  of  $e_a^1$  until time  $T_0$  in the following way:
  - Let  $\pi_a^1$  be  $\Lambda$ .
  - For every message  $m$  such that (1)  $m$  is sent by a process  $s$  in  $\pi_a^1$  with  $s \neq a$  and  $s \neq c_1$ , and (2)  $m$  is not received in  $\pi_a^1$  (i.e.,  $\Lambda$ ), the sending event of  $m$  is removed from  $\pi_a^1$ . In other words, each message which is sent by a process other than  $a$  and  $c_1$  and not received by  $T_0$  is removed from  $\pi_a^1$ . This step ensures that processes can only receive messages from  $a$  and  $c_1$  after  $T_0$ .
- Each process  $p \in \Psi \setminus \{a, c_1\}$  is silent after time  $T_0$ , i.e., it does not send any message after time  $T_0$ .

Fig. 2 depicts the devised execution  $e_a^1$ .

Let us analyze  $e_a^1$  (the summary of the analysis can be seen in Fig. 2):

- No process receives any message from a process  $s$  after time  $T_0$ , where  $s \neq a$  and  $s \neq c_1$  (step 2b). In other words, if any process receives a message after time  $T_0$ , then the message was sent either by  $a$  or  $c_1$ .
- Until time  $T_0$ ,  $e_a^1$  is indistinguishable from  $\Lambda$  to both  $a$  and  $c_1$  (step 2a and step 2b). Thus,  $a$  decides  $v$  in  $e_a^1$ , whereas  $c$  decides  $v' \neq v$  in  $e_a^1$ . Moreover,  $c_1$  does not obtain any proof of culpability until time  $T_0$  in  $e_a^1$  (due to Lemma 1).
- For every process  $q \in (A \cup C) \setminus \{a, c_1\}$ ,  $q$  does not behave correctly until time  $T_0$  solely because it omits some messages (step 2b).

Finally, we prove that  $a$  sends (at least)  $t_0 + 1 \in \Omega(n)$  accountability-specific messages in  $e_a^1$ . Intuitively,  $a$  does so to ensure that

$c_1$  is able to obtain a proof of culpability of  $t_0 + 1 = n - 2t_0$  processes in  $e_a^1$  (to satisfy accountability).

**Lemma 2.** Process  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_1$  in  $e_a^1$ .

**Proof.** Given the messages  $c_1$  receives until time  $T_0$  in  $e_a^1$ ,  $c_1$  is not able to construct a proof of culpability of any process (by Lemma 1). However, as  $abc$  satisfies accountability,  $c_1$  eventually obtains proofs of culpability of  $t_0 + 1$  processes. Hence,  $c_1$  obtains the proofs after time  $T_0$ . As  $c_1$  only receives messages from  $a$  (and itself) after time  $T_0$ ,  $c_1$  must have incorporated (at least)  $t_0 + 1$  messages sent by  $a$  into the obtained proofs of culpabilities; thus, all of these messages are accountability-specific. Therefore,  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_1$  in  $e_a^1$ .  $\square$

We denote by  $T_1$  the first time such that (1)  $T_1 > T_0$ , and (2)  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_1$  by time  $T_1$  in  $e_a^1$ .

**Step 2:** Construction of an infinite execution  $e_a^2$  in which  $a$  sends  $\Omega(n)$  accountability-specific messages to both  $c_1$  and  $c_2$ .

This step of the construction is purely demonstrative. Namely, we show how to construct  $e_a^2$  by relying on  $e_a^1$ . In the next step of the construction, we will generalize the construction from  $e_a^i$  (in which  $a$  sends  $\Omega(n)$  accountability-specific messages to each process in  $\{a_1, \dots, a_i\}$ ) to  $e_a^{i+1}$  (in which  $a$  sends  $\Omega(n)$  accountability-specific messages to each process in  $\{a_1, \dots, a_i, a_{i+1}\}$ ).







We construct  $e_a^2$  in the following way:

- Only processes  $a$  and  $c_2$  are correct.
- We construct the prefix  $\pi_a^2$  of  $e_a^2$  until time  $T_1$  in the following manner:
  - Let  $\pi_a^2$  be the prefix of  $e_a^1$  until time  $T_1$ .
  - We correct the behavior of  $c_2$  until time  $T_0$  by inserting all the messages omitted in  $e_a^1$  (i.e.,  $c_2$  behaves exactly as it behaves in  $\Lambda$ ). After time  $T_0$ , the behavior of  $c_2$  is correct (as  $c_2$  is correct in  $e_a^2$ ).
  - For every message  $m$  sent by  $c_1$  in  $\pi_a^2$  such that (1)  $m$  is sent to a process  $r$  with  $r \neq a$  and  $r \neq c_1$ , and (2)  $m$  is not received by time  $T_0$ , the sending event of  $m$  is removed from  $\pi_a^2$ . In other words, only  $a$  (and  $c_1$ ) receive messages from  $c_1$  after  $T_0$ .
  - For every message  $m$  sent by  $c_1$  in  $\pi_a^2$  such that (1)  $m$  is sent to  $a$ , and (2)  $m$  is not received by time  $T_1$ , the sending event of  $m$  is removed from  $\pi_a^2$ . This step of the construction ensures that  $a$  only receives messages from  $c_1$  until time  $T_1$ ; after  $T_1$ ,  $a$  receives no messages from  $c_1$ .
- Process  $c_2$  does not receive any message from any other process between times  $T_0$  and  $T_1$ .
- Each process  $p \in \Psi \setminus \{a, c_2\}$  is silent after time  $T_1$ , i.e., it does not send any message after time  $T_1$ . Recall that processes in  $\Psi \setminus \{a, c_1, c_2\}$  are silent after time  $T_0$  (due to the construction of  $e_a^1$ ).

Fig. 3 depicts  $e_a^2$ .

The following holds for  $e_a^2$  (summarized in Fig. 3):

- After  $T_0$ , only  $a$  and  $c_1$  receive messages from  $c_1$  (step 2c). Moreover, even  $a$  stops receiving messages from  $c_1$  after time  $T_1$  (step 2d).
- After time  $T_0$ ,  $c_2$  only receives messages from  $a$  and itself (step 2c). Furthermore, between  $T_0$  and  $T_1$ ,  $c_2$  only receives messages from itself (step 3).
- Until time  $T_0$ ,  $e_a^2$  is indistinguishable from  $\Lambda$  to both  $a$  and  $c_2$  (step 2a). Thus,  $a$  decides  $v$ , whereas  $c_2$  decides  $v' \neq v$ .

	$T_0$	$T_1$	$T_2$
 $a$	<ul style="list-style-type: none"> <li>decides <math>v</math></li> <li>obtains no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>receives messages only from <math>c_1</math> (and itself)</li> <li>sends <math>\Omega(n)</math> accountability-specific messages to <math>c_1</math> by <math>T_1</math></li> </ul>	<ul style="list-style-type: none"> <li>receives messages only from <math>c_2</math> (and itself)</li> <li>sends <math>\Omega(n)</math> accountability-specific messages to <math>c_2</math> by <math>T_2</math></li> </ul>
 $c_1$	<ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
 $c_2$	<ul style="list-style-type: none"> <li>decides <math>v'</math></li> <li>obtains no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>receives messages only from itself</li> </ul>	<ul style="list-style-type: none"> <li>receives messages only from <math>a</math> (and itself)</li> <li>achieves accountability</li> </ul>
 $c_3$	<ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
$\vdots$			
 $c_{t_0}$	<ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
 all other processes	<ul style="list-style-type: none"> <li>behaves correctly, except that some messages are omitted</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>

$\pi_a^2$

Fig. 3. Execution  $e_a^2$ . All modifications introduced to  $e_a^1$  (in order to obtain  $e_a^2$ ) are noted in red.

- For every process  $q \in (A \cup C) \setminus \{a, c_1, c_2\}$ ,  $q$  does not behave correctly until time  $T_0$  solely because it omits some messages.

We conclude this step of the construction of  $e_a$  by proving that  $a$  sends  $\Omega(n)$  accountability-specific messages both to  $c_1$  and  $c_2$  in  $e_a^2$ .

**Lemma 3.** *Process  $a$  sends  $\Omega(n)$  accountability-specific messages both to  $c_1$  and  $c_2$  in  $e_a^2$ .*

**Proof.** First,  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_1$  in  $e_a^2$  as (1)  $e_a^2$  is indistinguishable from  $e_a^1$  to  $a$  until time  $T_1$  (due to the construction of  $e_a^2$ ), and (2)  $a$  sends  $\Omega(n)$  accountability-specific messages by  $T_1$  in  $e_a^1$  (by Lemma 2). Hence, it is left to prove that  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_2$  in  $e_a^2$ , as well.

As already noted,  $a$  and  $c_2$  disagree in  $e_a^2$ . Since  $abc$  satisfies accountability,  $c_2$  eventually obtains proofs of culpability of (at least)  $t_0 + 1$  processes. Moreover, due to Lemma 1, given the messages  $c_2$  receives until time  $T_0$ ,  $c_2$  is unable to form a proof of culpability of any process. Therefore,  $c_2$  obtains the culpability proofs after  $T_0$ . Given that  $c_2$  only receives messages from  $a$  (and itself) after  $T_0$ ,  $c_2$  must have “used”  $\Omega(n)$  messages received from  $a$  to form the culpability proofs (in order to satisfy accountability). Thus,  $a$  indeed sends  $\Omega(n)$  accountability-specific messages to  $c_2$  in  $e_a^2$ , which concludes the proof.  $\square$

We denote by  $T_2$  the first time such that (1)  $T_2 > T_1$ , and (2)  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_2$  by time  $T_2$  in  $e_a^2$ . Note that  $a$  sends  $\Omega(n)$  messages to both  $c_1$  and  $c_2$  by  $T_2$  in  $e_a^2$  as  $T_2 > T_1$ .

**Step 3:** Construction of an infinite execution  $e_a^{i+1}$  in which  $a$  sends  $\Omega(n)$  accountability-specific messages to each process in  $\{c_1, c_2, \dots, c_i, c_{i+1}\}$ , where  $i \in [1, t_0 - 1]$ .

We construct  $e_a^{i+1}$  from  $e_a^i$ . In order to do so, we describe the execution  $e_a^i$ :

- Property 1:** Only processes  $a$  and  $c_i$  are correct in  $e_a^i$ .
- Property 2:** Until time  $T_0$ ,  $e_a^i$  is indistinguishable from  $\Lambda$  to both  $a$  and  $c_i$ .








	$T_0$	$T_1$	$T_2 \dots T_{i-1}$	$T_i$
 $a$	<ul style="list-style-type: none"> <li>decides <math>v</math></li> <li>no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_1</math> <math>\Omega(n)</math> to <math>c_1</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_2</math> <math>\Omega(n)</math> to <math>c_2</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_i</math> <math>\Omega(n)</math> to <math>c_i</math></li> </ul>
 $c_1$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>make <math>a</math> send messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
 $c_2$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>make <math>a</math> send messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
$\vdots$				
 $c_{i-1}$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
 $c_i$	<ul style="list-style-type: none"> <li>decides <math>v'</math></li> <li>no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>only <math>c_i</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>c_i</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_i</math> accountability</li> </ul>
 $c_{i+1}$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
 all other processes	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>

Fig. 4. A summary of the execution  $e_a^i$ .

- Property 3:** For every process  $c \in \{c_{i+1}, c_{i+2}, \dots, c_{t_0}\}$ ,  $c$  behaves correctly in  $e_a^i$  until time  $T_0$  except that some messages are omitted.
- Property 4:** For every process  $f \in \Psi \setminus \{a, c_1, c_2, \dots, c_{i-1}, c_i\}$ , no process receives any message from  $f$  after  $T_0$ .
- Property 5:** For every process  $c \in \{c_1, c_2, \dots, c_{i-1}\}$ , only processes  $a$  and  $c$  receive any message from  $c$  after  $T_0$ .
- Property 6:** There exists a time  $T_i > T_0$  such that  $a$  has sent  $\Omega(n)$  messages to each process  $c \in \{c_1, c_2, \dots, c_{i-1}, c_i\}$ .
- Property 7:** For every  $j \in [1, i - 1]$ , there exists a time  $T_j$  such that, if a process  $a$  receives a message from  $c_j$  after  $T_0$ , the reception happens between  $T_{j-1}$  and  $T_j$  ( $T_j > T_{j-1}$ ).
- Property 8:** Process  $c_i$  does not receive any message from another process between  $T_0$  and  $T_{i-1}$ .
- Property 9:** If process  $a$  receives a message from  $c_i$  after  $T_0$ , the reception happens after  $T_{i-1}$ .

The execution  $e_a^i$  is summarized in Fig. 4. Observe that  $e_a^1$  and  $e_a^2$  satisfy the aforementioned properties.

Now, we construct  $e_a^{i+1}$  from  $e_a^i$  in the same way we constructed  $e_a^2$  from  $e_a^1$ :

	$T_0$	$T_1$	$T_2 \dots T_{i-1}$	$T_i$	$T_{i+1}$
$a$	<ul style="list-style-type: none"> <li>decides <math>v</math></li> <li>no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_1</math></li> <li><math>\Omega(n)</math> to <math>c_1</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_2</math></li> <li><math>\Omega(n)</math> to <math>c_2</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_i</math></li> <li><math>\Omega(n)</math> to <math>c_i</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_{i+1}</math></li> <li><math>\Omega(n)</math> to <math>c_{i+1}</math></li> </ul>
$c_1$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>make <math>a</math> send messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
$c_2$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>make <math>a</math> send messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
$\vdots$					
$c_{i-1}$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
$c_i$	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>make <math>a</math> send messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
$c_{i+1}$	<ul style="list-style-type: none"> <li>decides <math>v'</math></li> <li>no proof of culpability</li> </ul>	<ul style="list-style-type: none"> <li>only <math>c_{i+1}</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>c_{i+1}</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>c_{i+1}</math></li> </ul>	<ul style="list-style-type: none"> <li>only <math>a</math> and <math>c_{i+1}</math></li> <li>accountability</li> </ul>
all other processes	<ul style="list-style-type: none"> <li>omitted messages</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>	<ul style="list-style-type: none"> <li>silent</li> </ul>
	$\pi_a^{i+1}$				

Fig. 5. Execution  $e_a^{i+1}$ . All modifications introduced to  $e_a^i$  (in order to obtain  $e_a^{i+1}$ ) are noted in red.

- Only processes  $a$  and  $c_{i+1}$  are correct.
- We construct the prefix  $\pi_a^{i+1}$  of  $e_a^{i+1}$  until time  $T_i$  in the following manner:
  - Let  $\pi_a^{i+1}$  be the prefix of  $e_a^i$  until time  $T_i$ .
  - We correct the behavior of  $c_{i+1}$  until time  $T_0$  by inserting all the messages omitted in  $e_a^i$  (i.e.,  $c_{i+1}$  behaves exactly as it behaves in  $\Lambda$ ). After time  $T_0$ , the behavior of  $c_{i+1}$  is correct (as  $c_{i+1}$  is correct in  $e_a^{i+1}$ ).
  - For every message  $m$  sent by  $c_i$  in  $\pi_a^i$  such that (1)  $m$  is sent to a process  $r$  with  $r \neq a$  and  $r \neq c_i$ , and (2)  $m$  is not received by time  $T_0$ , the sending event of  $m$  is removed from  $\pi_a^i$ . In other words, only  $a$  (and  $c_i$ ) receive messages from  $c_i$  after  $T_0$ .
  - For every message  $m$  sent by  $c_i$  in  $\pi_a^i$  such that (1)  $m$  is sent to  $a$ , and (2)  $m$  is not received by time  $T_i$ , the sending event of  $m$  is removed from  $\pi_a^i$ . This step of the construction ensures that  $a$  only receives messages from  $c_i$  until time  $T_i$ ; after  $T_i$ ,  $a$  receives no messages from  $c_i$ .
- Process  $c_{i+1}$  does not receive any message from any other process between times  $T_0$  and  $T_i$ .
- Each process  $p \in \Psi \setminus \{a, c_{i+1}\}$  is silent after time  $T_i$ , i.e., it does not send any message after time  $T_1$ . Recall that processes in  $\Psi \setminus \{a, c_1, c_2, \dots, c_{i-1}, c_i, c_{i+1}\}$  are silent after time  $T_0$  (due to the property 4 of  $e_a^i$ ).

Fig. 5 depicts  $e_a^{i+1}$ .

First, let us prove that all nine introduced properties are preserved for  $e_a^{i+1}$ :

- **Property 1:** Only processes  $a$  and  $c_{i+1}$  are correct by construction of  $e_a^{i+1}$ .
- **Property 2:** For  $c_{i+1}$ , this property is satisfied due to the construction of  $e_a^{i+1}$  (step 2b). For  $a$ , the property is satisfied as (1)  $a$  cannot distinguish  $e_a^{i+1}$  from  $e_a^i$  until time  $T_i > T_0$ , and (2)  $a$  cannot distinguish  $e_a^{i+1}$  from  $\Lambda$  until time  $T_0$ .
- **Property 3:** This property is satisfied as it is satisfied for  $e_a^i$ .
- **Property 4:** This property is satisfied as it is satisfied for  $e_a^i$ .
- **Property 5:** For every process  $c \in \{c_1, c_2, \dots, c_{i-1}\}$ , the property is satisfied as it is satisfied for  $e_a^i$ . For  $c_i$ , the property holds due to the construction of  $e_a^{i+1}$  (step 2c).
- **Property 6:** We prove the property by proving Lemma 4.
- **Property 7:** For every process  $c \in \{c_1, c_2, \dots, c_{i-1}\}$ , the property holds as it holds for  $e_a^i$ . For  $c_i$ , if  $a$  receives a message from  $c_i$

after  $T_0$ , that happens after  $T_{i-1}$  (as  $e_a^{i+1}$  is indistinguishable from  $e_a^i$  until  $T_i > T_{i-1}$  to  $a$  and  $a$  receives messages from  $c_i$  only after  $T_{i-1}$  due to the property 9 of  $e_a^i$ ). Moreover,  $a$  receives no messages from  $c_i$  after time  $T_i$  due to the construction of  $e_a^{i+1}$  (step 2d).

- **Property 8:** This property is ensured due to the step 3 of the construction.
- **Property 9:** The property is ensured by construction. Namely,  $a$  cannot distinguish  $e_a^{i+1}$  from  $e_a^i$  until time  $T_i$  and  $a$  does not receive any message from  $c_{i+1}$  in  $e_a^i$  (as the property 4 holds for  $e_a^i$ ).

Finally, let us prove that  $a$  sends  $\Omega(n)$  accountability-specific messages to each process  $c \in \{c_1, c_2, \dots, c_i, c_{i+1}\}$  in  $e_a^{i+1}$ ; the following lemma ensures that the property 6 is satisfied for  $e_a^{i+1}$ .

**Lemma 4.** Process  $a$  sends  $\Omega(n)$  accountability-specific messages to each process  $c \in \{c_1, c_2, \dots, c_i, c_{i+1}\}$  in  $e_a^{i+1}$ .

**Proof.** First,  $a$  sends  $\Omega(n)$  accountability-specific messages to each process in the  $\{c_1, c_2, \dots, c_{i-1}, c_i\}$  set in  $e_a^{i+1}$  as (1)  $e_a^{i+1}$  is indistinguishable to  $a$  from  $e_a^i$  until time  $T_i$  (due to the construction of  $e_a^{i+1}$ ), and (2)  $a$  sends  $\Omega(n)$  accountability-specific messages to each process in  $\{c_1, c_2, \dots, c_i\}$  by  $T_i$  in  $e_a^i$  (by the property 6 of  $e_a^i$ ). Hence, it is left to prove that  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_{i+1}$  in  $e_a^{i+1}$ , as well.

Processes  $a$  and  $c_{i+1}$  disagree in  $e_a^{i+1}$ . Since  $abc$  satisfies accountability,  $c_{i+1}$  eventually obtains proofs of culpability of (at least)  $t_0 + 1$  processes. Moreover, due to Lemma 1, given the messages  $c_{i+1}$  receives until time  $T_0$ ,  $c_{i+1}$  is unable to form a proof of culpability of any process. Therefore,  $c_{i+1}$  obtains the culpability proofs after  $T_0$ . Given that  $c_{i+1}$  only receives messages from  $a$  (and itself) after  $T_0$  (due to the properties 4 and 5 of  $e_a^{i+1}$ ),  $c_{i+1}$  must have “used”  $\Omega(n)$  messages received from  $a$  to form the culpability proofs (to satisfy accountability). Thus,  $a$  indeed sends  $\Omega(n)$  accountability-specific messages to  $c_{i+1}$  in  $e_a^{i+1}$ , as well.  $\square$

We denote by  $T_{i+1}$  the first time such that (1)  $T_{i+1} > T_i$ , and (2)  $a$  sends  $\Omega(n)$  accountability-specific messages to  $c_{i+1}$  by time  $T_{i+1}$  in  $e_a^{i+1}$ . Note that  $a$  sends  $\Omega(n)$  messages to each process in the  $\{c_1, \dots, c_i, c_{i+1}\}$  set by  $T_{i+1}$ .



**Step 4:** Construction of a finite execution  $e_a$  in which (1)  $a$  sends  $\Omega(n^2)$  accountability-specific messages, and (2)  $a$  is the only correct process.

We construct  $e_a$  in the following manner:

1. Only process  $a$  is correct in  $e_a$ .
2. We build the prefix  $\pi_a$  of  $e_a$  until time  $T_{t_0}$  in the following manner, where  $T_{t_0}$  is the time specified explicitly in the construction of  $e_a^{t_0}$  (constructed by the generic transformation introduced in the previous step):
  - (a) Let  $\pi_a$  be the prefix of  $e_a^{t_0}$  until time  $T_{t_0}$ .
  - (b) For every message  $m$  sent by  $c_{t_0}$  in  $\pi_a$  such that (1)  $m$  is sent to a process  $r$  with  $r \neq a$  and  $r \neq c_{t_0}$ , and (2)  $m$  is not received by time  $T_0$ , the sending event of  $m$  is removed from  $\pi_a$ . In other words, only  $a$  (and  $c_{t_0}$ ) receive messages from  $c_{t_0}$  after  $T_0$ .
  - (c) For every message  $m$  sent by  $c_{t_0}$  in  $\pi_a$  such that (1)  $m$  is sent to  $a$ , and (2)  $m$  is not received by time  $T_{t_0}$ , the sending event of  $m$  is removed from  $\pi_a$ . This step of the construction ensures that  $a$  only receives messages from  $c_{t_0}$  until time  $T_{t_0}$ ; after  $T_{t_0}$ ,  $a$  receives no messages from  $c_{t_0}$ .
3. Each process  $p \in \Psi \setminus \{a\}$  is silent after time  $T_i$ , i.e., it does not send any message after time  $T_{t_0}$ . Recall that processes in  $\Psi \setminus \{a, c_1, c_2, \dots, c_{t_0}\}$  are silent after time  $T_0$ .

Since  $a$  cannot distinguish  $e_a^{t_0}$  from  $e_a$  until time  $T_{t_0}$ ,  $a$  sends  $\Omega(n^2)$  accountability-specific messages in  $e_a$ . Lastly, we associate  $e_a$  with  $T_a = T_{t_0}$ .

**Construction of  $\mathcal{E}$  (part 2):** The first part of the proof was devoted to constructing  $e_a$ , an execution in which a fixed correct process  $a \in A$  sends a quadratic number of accountability-specific messages. The second part “merges” all of these executions in order to obtain an execution with a cubic number of sent accountability-specific messages.

**Theorem 5.** *The accountability complexity of  $abc$ , where  $abc$  is a  $t_0$ -resilient (deterministic or probabilistic) accountable Byzantine consensus protocol and  $t_0 \in \Omega(n)$ , is  $\Omega(n^3)$ .*

**Proof.** We prove the theorem by constructing an execution  $\mathcal{E}$  with a cubic number of accountability-specific messages. Recall that, for each process  $a \in A$ , time  $T_a > T_0$  is associated with  $e_a$ ;  $a$  sends  $\Omega(n^2)$  accountability-specific messages by  $T_a$  in  $e_a$ .

We construct  $\mathcal{E}$  in the following manner. First, we “merge” executions  $e_a$  until time  $T_0$ , for every  $a \in A$ . More formally, we build a prefix  $\rho$  of  $\mathcal{E}$  until time  $T_0$  using the following construction:

1. Let  $\rho$  be the prefix of  $e_{a_1}$  until time  $T_0$ , where  $a_1 \in A$ .
2. For every process  $a \in A \setminus \{a_1\}$ :
  - (a) If there exists a message  $m$  sent by a process  $p \in \Psi$  in the prefix of  $e_a$  until time  $T_0$  such that  $m$  is not sent in  $\rho$ , add  $m$  to be sent in  $\rho$  at the exact same time as in  $e_a$ .

Intuitively, we create  $\rho$  as the “union” of the prefixes of  $e_a$  until time  $T_0$ , for every  $a \in A$ . Observe that all processes from the group  $A$  are correct in  $\rho$ .

After time  $T_0$ , we do the following:

1. Processes from the  $C$  set behave towards a process  $a$  as they do in  $e_a$ , for every  $a \in A$ .
2. Processes from the group  $B$  are silent, i.e., they do not send any messages.

3. Messages between processes from the group  $A$  that are not received by time  $T_0$  are delayed until after time  $T^* = \max(T_{a_1}, T_{a_2}, \dots, T_{a_{t_0}})$ .

Given that no process  $a \in A$  distinguishes  $\mathcal{E}$  from  $e_a$  until time  $T^* > T_a$ , each process  $a \in A$  sends a quadratic number of accountability-complexity messages in  $\mathcal{E}$ . Since  $|A| = t_0 \in \Omega(n)$ , the overall accountability complexity of execution  $\mathcal{E}$  is  $\Omega(n^3)$ , which concludes the proof.  $\square$

## 5. Generalized $ABC$ transformation

We have shown that  $ABC$  enables Byzantine consensus protocols to obtain accountability. This section generalizes our  $ABC$  transformation and defines its applicability. Namely, we specify a class of distributed computing problems named *easily-accountable agreement tasks*, and we prove that generalized  $ABC$  enables accountability in such tasks.

We introduce agreement tasks in § 5.1. Then, we define the class of easily-accountable agreement tasks (§ 5.2), and prove the correctness of generalized  $ABC$  transformation applied to such agreement tasks (§ 5.3).

### 5.1. Agreement tasks

Agreement tasks represent an abstraction of distributed input-output problems performed in a Byzantine environment. Specifically, each process has its *input value*. We assume that “ $\perp$ ” denotes the special input value of a process that specifies that the input value is non-existent. A process may eventually halt; if a process halts, it produces its *output value*. The “ $\perp$ ” output value of a process means that the process has not yet halted (and produced its output value). We denote by  $I_i$  (resp.,  $O_i$ ) the input (resp., output) value of process  $P_i$ . We note that some processes might never halt if permitted by the definition of an agreement task. We provide the formal explanation in the rest of the subsection.

An agreement task  $\mathcal{A}$  is parametrized with the upper bound  $t_{\mathcal{A}}$  on number of faulty processes that are tolerated. In other words, the specification of an agreement task assumes that no more than  $t_{\mathcal{A}}$  processes are faulty in any execution.

Any agreement task could be defined as a relation between input and output values of processes. Since we assume that processes might fail, we only care about input and output values of correct processes. Hence, an agreement task could be defined as a relation between input and output values of *correct* processes.

An *input configuration* of an agreement task  $\mathcal{A}$  is  $v_I = \{(P_i, I_i) \text{ with } P_i \text{ is correct}\}$ , where  $|v_I| \geq n - t_{\mathcal{A}}$ : an input configuration consists of input values of all correct processes. Similarly, an *output configuration* of an agreement task is  $v_O = \{(P_i, O_i) \text{ with } P_i \text{ is correct}\}$ , where  $|v_O| \geq n - t_{\mathcal{A}}$ : it contains output values of correct processes. We denote by  $\theta(v_O) = |\{O_i \mid (P_i, O_i) \in v_O \wedge O_i \neq \perp\}|$  the number of distinct non- $\perp$  values in the  $v_O$  output configuration.

Finally, we define an agreement task  $\mathcal{A}$  as tuple  $(\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$ , where:

- $\mathcal{I}$  denotes the set of all input configurations of  $\mathcal{A}$ .
- $\mathcal{O}$  denotes the set of all output configurations of  $\mathcal{A}$  such that  $\theta(v_O) \leq 1$ , for every  $v_O \in \mathcal{O}$ .
- $\Delta : \mathcal{I} \rightarrow 2^{\mathcal{O}}$ , where  $v_O \in \Delta(v_I)$  if and only if the output configuration  $v_O \in \mathcal{O}$  is valid given the input configuration  $v_I \in \mathcal{I}$ .
- $t_{\mathcal{A}} \leq \lfloor n/3 \rfloor - 1$  denotes the maximum number of faulty processes the task assumes.

As seen from the definition, correct processes that halt always output the same value in agreement tasks. Moreover, we define agreement tasks to tolerate less than  $n/3$  faults. Without loss of

generality, we assume that  $\Delta(v_I) \neq \emptyset$ , for every input configuration  $v_I \in \mathcal{I}$ . Moreover, for every  $v_O \in \mathcal{O}$ , there exists  $v_I \in \mathcal{I}$  such that  $v_O \in \Delta(v_I)$ .

We note that some problems that are traditionally considered as “agreement” problems do not fall into our classification of agreement tasks. For instance, Byzantine lattice agreement [50] or  $k$ -set agreement [21] is *not* agreement tasks per our definition since the number of distinct non- $\perp$  values that can be outputted is greater than 1.

**Solutions** The following definitions specify solutions of agreement tasks.

**Definition 8** (Solution of an agreement task). A protocol  $\Pi_{\mathcal{A}}$  deterministically (resp., probabilistically) solves an agreement task  $\mathcal{A} = (\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$  if and only if, in every execution with up to  $t_{\mathcal{A}}$  faults, there exists (resp., exists with probability 1) an unknown time  $T_D$  such that  $v_O \in \Delta(v_I)$ , where  $v_I \in \mathcal{I}$  denotes the input configuration that consists of input values of all correct processes and  $v_O \in \mathcal{O}$  denotes the output configuration that (1) consists of output values (potentially  $\perp$ ) of all correct processes, and (2) no correct process  $P_i$  with  $O_i = \perp$  updates its output value after  $T_D$ .

Lastly, we define accountable solutions of agreement tasks.

**Definition 9** (Accountable solution of an agreement task). A protocol  $\Pi_{\mathcal{A}}^{\text{acc}}$  deterministically (resp., probabilistically) solves an agreement task  $\mathcal{A} = (\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$  with accountability if and only if the following holds:

- **$\mathcal{A}$ -Solvability:**  $\Pi_{\mathcal{A}}^{\text{acc}}$  deterministically (resp., probabilistically) solves  $\mathcal{A}$ .
- **Accountability:** If two correct processes output different values, then every correct process eventually detects (at least)  $n - 2t_{\mathcal{A}}$  faulty processes and obtains a proof of culpability of each detected process.

## 5.2. Easily-accountable agreement tasks

Fix an agreement task  $\mathcal{A} = (\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$ . We say that  $\mathcal{A}$  is an *easily-accountable agreement task* if and only if one of the following conditions is satisfied:

1. “All-or-None-Decidability”: There does not exist  $v_O \in \mathcal{O}$  such that  $(P_i, O_i \neq \perp) \in v_O$  and  $(P_j, O_j = \perp) \in v_O$ ; or
2. “Partial-Decidability”: For every  $v_I \in \mathcal{I}$  such that there exists  $v_O \in \Delta(v_I)$ , where  $(P_i, O_i = v \neq \perp) \in v_O$  and  $(P_j, O_j = \perp) \in v_O$ , the following holds:

for every  $c \in \mathbb{P}(\{P_i \mid (P_i, I_i) \in v_I\})$ ,  $\exists v'_O \in \Delta(v_I)$ , where

$$\forall P_i \in c : (P_i, O_i = v) \in v'_O \text{ and}$$

$$\forall P_j \in \{P_k \mid (P_k, I_k) \in v_I\}$$

$$\setminus c : (P_j, O_j = \perp) \in v'_O.$$

“All-or-None-Decidability” characterizes all the problems in which either every process halts or none does. For instance, Byzantine consensus [40] and Byzantine reliable broadcast [17] satisfy “All-or-None-Decidability”.

On the other hand, some agreement tasks permit that some processes halt, whereas others do not. We say that these tasks satisfy “Partial-Decidability” if and only if it is allowed for any subset of correct processes to halt (and output a value). Note that “Partial-Decidability” covers the case where no correct process ever halts. Byzantine consistent broadcast [17] is the only agreement task we

are aware of that satisfies “Partial-Decidability” (in the case of a Byzantine sender). However, the significance of Byzantine consistent broadcast (e.g., for implementing cryptocurrencies [35]) motivated us to consider the “Partial-Decidability” property.

## 5.3. Correctness of generalized ABC transformation

We now prove the correctness of our generalized ABC transformation (Algorithm 4). First, we show that Algorithm 4 solves an easily-accountable agreement task  $\mathcal{A}$  if  $\mathcal{A}$  satisfies “All-or-None-Decidability”.

**Lemma 5.** Let  $\mathcal{A} = (\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$  be an easily-accountable agreement task that satisfies “All-or-None-Decidability”. Algorithm 4 deterministically (resp., probabilistically) solves  $\mathcal{A}$  if  $\Pi_{\mathcal{A}}$  (line 5) deterministically (resp., probabilistically) solves  $\mathcal{A}$ .

**Proof.** If no correct process ever outputs a value at line 9, then no correct process confirms any value from accountable confirmer (because no correct process submits any value to accountable confirmer at line 10). Hence, no correct process produces any output at line 12, which concludes the proof in this scenario.

Otherwise, each correct process eventually outputs a value at line 9. Moreover, all correct processes output the exact same value  $v$  (since  $\mathcal{A}$  is an agreement task). Therefore, all correct processes submit the same value  $v$  to accountable confirmer (line 10). By terminating convergence of accountable confirmer, all correct processes eventually confirm value  $v$  (line 11) and output it (line 12). Once this happens, the agreement task  $\mathcal{A}$  is solved, which concludes the lemma.  $\square$

Now, we prove that Algorithm 4 solves an easily-accountable agreement task  $\mathcal{A}$  if  $\mathcal{A}$  satisfies “Partial-Decidability”.

**Lemma 6.** Let  $\mathcal{A} = (\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$  be an easily-accountable agreement task that satisfies “Partial-Decidability”. Algorithm 4 deterministically (resp., probabilistically) solves  $\mathcal{A}$  if  $\Pi_{\mathcal{A}}$  (line 5) deterministically (resp., probabilistically) solves  $\mathcal{A}$ .

**Proof.** Let  $v_I$  denote a specific input configuration of  $\mathcal{A}$ . We consider two cases:

- If no or all correct processes output a value at line 9, the proof is identical to the proof of Lemma 5.
- Otherwise, there exists a correct process that outputs a value  $v$  at line 9 and another correct process that does not output any value at line 9. Since  $\mathcal{A}$  is an agreement task, any correct process that outputs a value at line 9 outputs the value  $v$ . Moreover, any correct process that outputs a value at line 12 outputs the value  $v$  (ensured by validity of accountable confirmer). Finally, once the system stabilizes at time  $T_D$  (the system stabilizes at time  $T_D$  if and only if no correct process  $P_i$  with  $O_i = \perp$  updates its output value after  $T_D$ ), the fact that any subset of correct processes could halt and that all halted processes output  $v$  implies that Algorithm 4 solves  $\mathcal{A}$ .

The lemma holds.  $\square$

Finally, we are ready to prove that Algorithm 4 solves  $\mathcal{A}$  with accountability, where  $\mathcal{A}$  is an easily-accountable agreement task, which means that generalized ABC is correct.

**Theorem 6.** Let  $\mathcal{A} = (\mathcal{I}, \mathcal{O}, \Delta, t_{\mathcal{A}})$  be an easily-accountable agreement task. Algorithm 4 deterministically (resp., probabilistically) solves  $\mathcal{A}$  with accountability if  $\Pi_{\mathcal{A}}$  (line 5) deterministically (resp., probabilistically) solves  $\mathcal{A}$ .

**Algorithm 4** Generalized  $\mathcal{ABC}$  Transformation - Code For Process  $P_i$ .

---

```

1: Implements:
2:   Agreement Task  $\mathcal{A}$  With Accountability, instance  $a \leftarrow \mathcal{A}$ 
3: Uses:
4:    $\triangleright$  Protocol to be transformed
5:   Protocol that (deterministically or probabilistically) solves agreement task  $\mathcal{A}$ , instance  $\Pi_{\mathcal{A}}$ 
6:    $t_{\mathcal{A}}$ -Resilient Accountable Confirmer, where  $t_{\mathcal{A}}$  is the resilience of  $\mathcal{A}$ , instance  $ac$ 
7: upon event  $\langle a \leftarrow \mathcal{A}, \text{Input} | \text{input} \rangle$  do  $\triangleright$  Input
8:   trigger  $\langle \Pi_{\mathcal{A}}, \text{Input} | \text{input} \rangle$ 
9: upon event  $\langle \Pi_{\mathcal{A}}, \text{Output} | \text{output} \rangle$  do
10:   trigger  $\langle ac, \text{Submit} | \text{output} \rangle$ 
11: upon event  $\langle ac, \text{Confirm} | \text{confirmation} \rangle$  do
12:   trigger  $\langle a \leftarrow \mathcal{A}, \text{Output} | \text{confirmation} \rangle$   $\triangleright$  Output
13: upon event  $\langle ac, \text{Detect} | F, \text{proof} \rangle$  do
14:   trigger  $\langle a \leftarrow \mathcal{A}, \text{Detect} | F, \text{proof} \rangle$   $\triangleright$  Detection

```

---

**Proof.** Algorithm 4 satisfies  $\mathcal{A}$ -solvability by Lemmas 5 and 6. Finally, Algorithm 4 ensures accountability since the accountable confirmer ensures detection of (at least)  $n - 2t_{\mathcal{A}}$  faulty processes whenever a disagreement occurs.  $\square$

## 6. Concluding remarks

We presented  $\mathcal{ABC}$ , a generic and simple transformation that allows non-synchronous (deterministic or probabilistic) Byzantine consensus protocols to obtain accountability. Besides its simplicity,  $\mathcal{ABC}$  is efficient: it introduces an additive overhead of only two all-to-all communication rounds and  $O(n^2)$  exchanged bits of information in the common case. Furthermore, we show that  $\mathcal{ABC}$  can easily be generalized to other agreement problems (e.g., Byzantine reliable broadcast, Byzantine consistent broadcast). Future work includes (1) designing similarly simple and efficient transformations for problems not covered by the generalized  $\mathcal{ABC}$  transformation, like Byzantine lattice and  $k$ -set agreement problems, and (2) circumventing the cubic accountability complexity bound using randomization techniques.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## Acknowledgments

This work is supported in part by the Australian Research Council Future Fellowship funding scheme (#180100496), the Ethereum Foundation, and by Singapore MOE Grant MOE-T2EP20122-0014.

## Appendix A. Formal proof of Theorem 1

**Theorem 1** (Unavoidable disagreement; restated). *For any non-synchronous  $t_0$ -resilient (deterministic or probabilistic) Byzantine consensus protocol among  $n$  processes, there exists an execution with  $t \geq n - 2t_0$  faulty processes in which correct processes disagree (i.e., decide different values).*

**Proof.** As we need to have at least two correct processes (in order for a disagreement to occur),  $t \leq n - 2$ . Let  $B$  denote the group of faulty processes;  $|B| = t \geq n - 2t_0$ . We denote by  $A$  a group of any  $\lceil \frac{n-t}{2} \rceil$  correct processes and by  $C$  the group of “other”  $\lfloor \frac{n-t}{2} \rfloor$  correct processes. Note that (1)  $A \cap C = \emptyset$ , (2)  $A \neq \emptyset$  (as  $n - t \geq 2$ ), and (3)  $C \neq \emptyset$  (as  $n - t \geq 2$ ).

Consider the following two executions:

- Execution  $e_1$ : In this execution, processes in  $A \cup B$  are correct, whereas processes in  $C$  are faulty and silent (i.e., they do not send any messages). All correct processes propose the same value  $v$ . The number of faulty processes in  $e_1$  is  $|C| = \lfloor \frac{n-t}{2} \rfloor \leq t_0$ . Due to the fact that the Byzantine consensus protocol is  $t_0$ -resilient and the number of faulty processes is  $\leq t_0$ , all correct processes decide  $v$  by some time  $T_1$ .
- Execution  $e_2$ : In this execution, processes in  $B \cup C$  are correct, whereas processes in  $A$  are faulty and silent. All correct processes propose the same value  $v'$ . The number of faulty processes in  $e_2$  is  $|A| \leq t_0$ . Hence, all correct processes decide  $v'$  by some time  $T_2$ .

Now, we build an execution  $e_3$ :

1. Processes in  $A \cup C$  are correct, whereas processes in  $B$  are faulty.
2. Processes in  $A$  propose  $v$ , whereas processes in  $C$  propose  $v'$ .
3. Processes in  $B$  behave (1) towards processes in  $A$  as in  $e_1$ , and (2) towards processes in  $C$  as in  $e_2$ . Moreover, all messages between groups  $A$  and  $C$  are delayed until after  $\max(T_1, T_2)$ .

Until time  $T_1$ , processes in  $A$  cannot distinguish  $e_3$  from  $e_1$ ; thus, processes in  $A$  decide  $v$  in  $e_3$ . Similarly, until time  $T_2$ , processes in  $C$  cannot distinguish  $e_3$  from  $e_2$ ; thus, processes in  $C$  decide  $v'$  in  $e_3$ . Therefore, a disagreement occurs in  $e_3$ , and there are  $n - 2t_0 \leq t \leq n - 2$  faulty processes in  $e_3$ , which concludes the proof.  $\square$

## Appendix B. Cryptographic primitives: formal overview

This subsection recalls the formal definitions of the cryptographic schemes we rely upon for constructing  $\mathcal{ABC}$ .

A family of real numbers  $(x_k)_{k \in \mathbb{N}} \in \mathbb{R}^{\mathbb{N}}$  is said to *belong to*  $\text{poly}(k)$  if there exists  $c \in \mathbb{N}$  such that  $x_k = O(k^c)$ . A family of real numbers  $(x_k)_{k \in \mathbb{N}} \in \mathbb{R}^{\mathbb{N}}$  is said to be *negligible*, denoted by  $(x_k)_{k \in \mathbb{N}} \in \text{neg}(k)$ , if for every  $c \in \mathbb{N}$ ,  $x_k = o(\frac{1}{k^c})$ . A probabilistic Turing Machine is  $k$ -bounded for some  $k \in \mathbb{N}$  if (1) it can be described with  $k$  bits, assuming a standard bit-string representation, and (2) it halts after  $k$  transitions.

The cryptographic schemes and their properties are defined with respect to a security parameter  $\kappa \in \mathbb{N}$ . A (local) protocol is said to be *efficient* if its complexity belongs to  $\text{poly}(\kappa)$ . A real number  $x$  (which, traditionally, describes a probability) is said to be *negligible* if it is parametrized with the security parameter  $\kappa$  and  $x \in \text{neg}(\kappa)$ . An adversary is said to be *polynomially-bounded* if it is parametrized with the security parameter  $\kappa$  and is  $\text{poly}(\kappa)$ -bounded. A property of a cryptographic scheme is said to *hold* if it cannot be violated by a polynomially-bounded adversary with more than a negligible probability.

Throughout the section, we denote by  $\text{String} \triangleq \{0, 1\}^*$  the set of all strings.

### B.1. Digital signatures

A *digital signature scheme* is a tuple of efficient local protocols  $(\text{Gen}, \text{Sign}, \text{Verify})$ , with:

- **Gen**: a probabilistic algorithm that takes the security number  $\kappa$  as the input, and randomly selects a pair  $(sk_i, pk_i)$  composed of a secret (i.e., private) key  $sk_i$  and a public (i.e., verification) key  $pk_i$ ; the bit-representations of  $sk_i$  and  $pk_i$  are of size of (at most)  $\kappa$  bits.
- **Sign** $(m, sk_i)$ : a (potentially probabilistic) algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a private key  $sk_i$  as the input. The algorithm outputs a signature  $\sigma_i$  whose bit-representation has a size of (at most)  $\kappa$  bits.
- **Verify** $(m, pk_j, \sigma_j)$ : a (potentially probabilistic) algorithm that takes (1) a string  $m \in \text{String}$ , (2) the public key  $pk_j$  of a process  $P_j$ , and (3) a signature  $\sigma_j$  as the input. The algorithm outputs  $\top$  (true) or  $\perp$  (false) depending on whether  $\sigma_j$  is deemed as a valid signature.

The following properties hold:

- **Correctness**: If  $(sk_i, pk_i) \leftarrow \text{Gen}(\kappa)$ , then  $\text{Verify}(m, pk_i, \text{Sign}(m, sk_i))$  returns  $\top$ .
- **Unforgeability**: If  $\text{Verify}(m, pk_j, \sigma_j)$  returns  $\top$ , then (1)  $\sigma_j \leftarrow \text{Sign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.

This scheme is formalized by the functionality  $\mathcal{F}_{\text{SIG}}$  in the universally composable (UC) framework [19].

### B.2. Public key infrastructure (PKI)

An *ideal public key infrastructure* is a tuple  $(\text{Keys}, \text{Sign}, \text{Verify})$ , where  $\text{Keys} = ((SK = (sk_1, \dots, sk_n), PK = (pk_1, \dots, pk_n)))$ , with:

- **PK**: a vector of public (i.e., verification) keys stored by every correct process; each public key  $pk_j$  is associated with the process  $P_j$ .
- **SK**: a vector of secret (i.e., private) keys such that, for every correct process  $P_i$ ,  $P_i$  stores its secret key  $sk_i$  which corresponds to its public key  $pk_i$ ;  $sk_i$  is hidden from the adversary (i.e., the unforgeability property of digital signatures is satisfied; Appendix B.1).
- **Sign** $(m, sk_i)$ : a (potentially probabilistic) algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a private key  $sk_i$  as the input. The algorithm outputs a signature  $\sigma_i$ .
- **Verify** $(m, pk_j, \sigma_j)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , (2) the public key  $pk_j$  of a process  $P_j$ , and (3) a signature  $\sigma_j$  as the input. The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma_j$  is deemed as a valid signature.

The following properties hold:

- **Correctness**: For every  $i \in [1, n]$ ,  $\text{Verify}(m, pk_i, \text{Sign}(m, sk_i))$  returns  $\top$ .
- **Unforgeability**: If  $\text{Verify}(m, pk_j, \sigma_j)$  returns  $\top$ , then (1)  $\sigma_j \leftarrow \text{Sign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.

In our paper, we assume an established PKI, i.e., we are not concerned with how such an infrastructure is obtained. We emphasize that the definition above does not state how a verifier learns the public (i.e., verification) key of another process. The associated

ideal functionality, formalized by  $\mathcal{F}_{\text{CA}}$  in the UC model [19], corresponds to a “rudimentary certification authority that registers party identities together with public values provided by the registered party”. Traditionally, this functionality is emulated in the following manner: a process publicly announces its public key using the Byzantine reliable broadcast primitive [13,17]. However, without additional assumptions, the resiliency of Byzantine reliable broadcast is bounded by  $n/3$  (without previously established PKI), even in the synchronous setting [40], making it impossible for implementation in overly corrupted systems.

Assuming that each party can solve cryptographic puzzles only at a bounded rate, it is possible to (1) implement a setup phase to establish an ideal PKI assuming a (potentially very large) bound  $\Delta$  on message delays [38,7], and then (2) run asynchronous protocols on top of the established ideal PKI (in the main phase), without facing the dilemma between safety and efficiency due to the choice of  $\Delta$ . (A small  $\Delta$  would threaten the safety, while a large  $\Delta$  would increase the latency.)

### B.3. Threshold signature scheme

A *non-interactive  $(k, n)$ -dual threshold signature scheme* is a tuple of efficient local protocols  $(\text{Keys}, \text{ShareSign}, \text{ShareVerify}, \text{Verify}, \text{Combine})$ , where  $\text{Keys} = (PK, SK = (sk_1, \dots, sk_n), VK = (vk_1, \dots, vk_n))$ , with:

- **PK**: a public key store by correct process.
- **VK**: a vector of verification keys stored by every correct process.
- **SK**: a vector of private key shares such that, for every correct process  $P_i$ ,  $P_i$  stores its private key share  $sk_i$ ;  $sk_i$  is hidden from the adversary.
- **ShareSign** $(m, sk_i)$ : a (potentially probabilistic) algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a private key share  $sk_i$  as the input. The algorithm outputs a partial signature  $\sigma_i^P$  of (at most)  $\kappa$  bits.
- **ShareVerify** $(m, vk_j, \sigma_j^P)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , (2) the verification key  $vk_j$  of a process  $P_j$ , and (3) a partial signature  $\sigma_j^P$  as the input. The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma_j^P$  is deemed as a valid partial signature.
- **Combine** $(m, \{\sigma_i\}_{i \in S \wedge S \subseteq [1, n] \wedge |S|=k})$ : an algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a subset  $S$  of size  $|S| = k$  of partial signatures  $\{\sigma_i\}_{i \in S}$ . The algorithm outputs a threshold signature  $\sigma^T$ .
- **Verify** $(m, PK, \sigma^T)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , (2) the public key  $PK$ , and (3) a threshold signature  $\sigma^T$ . The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma^T$  is deemed as a valid threshold signature.

The following properties hold:

- **Correctness of partial signatures**: For every  $i \in [1, n]$ ,  $\text{ShareVerify}(m, vk_i, \text{ShareSign}(m, sk_i))$  returns  $\top$ .
- **Unforgeability of partial signatures**: If  $\text{ShareVerify}(m, vk_j, \sigma_j^P)$  returns  $\top$ , then (1)  $\sigma_j^P \leftarrow \text{ShareSign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.
- **Correctness of threshold signatures**:  $\text{Verify}(m, PK, \text{Combine}(\{\text{ShareSign}(m, sk_j)\}_{j \in J \wedge J \subseteq [1, n] \wedge |J|=k}))$  returns  $\top$ .
- **Unforgeability of threshold signatures**: If  $\text{Verify}(m, PK, \sigma^T)$  returns  $\top$ , then there exists a set  $J$ ,  $|J| = k$ , of partial signatures such that, for each  $\sigma_j^P \in J$ , (1)  $\sigma_j^P \leftarrow \text{ShareSign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.



Importantly, there exist dual threshold signature schemes with threshold signatures having  $\kappa$  bits (e.g., [49]). Note that, without a trusted setup, a  $(k, n)$ -dual threshold signature scheme can be obtained via a distributed key generation (DKG) protocol (e.g., [2]) if the number of faulty processes does not exceed  $n - k$ . Otherwise, no guarantees exist. For example, if the number of faults exceeds  $n - k$ , we can have an imperfect threshold signature scheme in which the aforementioned properties would not hold (e.g., faulty processes could forge the private key shares of correct processes and use them to sign statements on their behalf).

#### B.4. Threshold signature scheme in a transparent setup (via $\Sigma$ -compressed protocols)

A transparent non-interactive  $(k, n)$ -dual threshold signature scheme is a scheme whose specification is extremely similar to the specification presented in Appendix B.3. The only difference is that a transparent scheme does not include a common public key, which implies that a trusted setup nor a DKG protocol is necessary. Hence, obtaining a transparent scheme is strictly easier (in terms of necessary assumptions) than obtaining a non-transparent scheme (Appendix B.3). For the completeness, the full specification is given below.

A transparent non-interactive  $(k, n)$ -dual threshold signature scheme is tuple of efficient local protocols (Keys, ShareSign, ShareVerify, Verify, Aggregate), where  $\text{Keys} = (\text{SK} = (sk_1, \dots, sk_n), \text{VK} = (vk_1, \dots, vk_n))$ , with:

- **VK**: a vector of verification keys stored by every correct process.
- **SK**: a vector of private key shares such that, for every correct process  $P_i$ ,  $P_i$  stores its private key share  $sk_i$ ;  $sk_i$  is hidden from the adversary.
- **ShareSign** $(m, sk_i)$ : a (potentially probabilistic) algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a private key share  $sk_i$  as the input. The algorithm outputs a signature share  $\sigma_i^S$  of (at most)  $\kappa$  bits.
- **ShareVerify** $(m, vk_j, \sigma_j^S)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , (2) the verification key  $vk_j$  of a process  $P_j$ , and (3) a signature share  $\sigma_j^S$  as the input. The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma_j^S$  is deemed as a valid signature.
- **Aggregate** $(m, \{\sigma_i\}_{i \in S \wedge S \subseteq [1, n] \wedge |S|=k})$ : an algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a subset  $S$  of size  $|S| = k$  of signature shares  $\{\sigma_i\}_{i \in S}$ . The algorithm outputs an aggregate signature  $\sigma^A$ .
- **Verify** $(m, \sigma^A)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , and (2) an aggregate signature  $\sigma^A$ . The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma^A$  is deemed as a valid aggregate signature.

The following properties hold:

- **Correctness of signature shares**: For every  $i \in [1, n]$ , **ShareVerify** $(m, vk_i, \text{ShareSign}(m, sk_i))$  returns  $\top$ .
- **Unforgeability of signature shares**: If **ShareVerify** $(m, vk_j, \sigma_j^S)$  returns  $\top$ , then (1)  $\sigma_j^S \leftarrow \text{ShareSign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.
- **Correctness of aggregate signatures**: **Verify** $(m, \text{Aggregate}(\{\text{ShareSign}(m, sk_j)\}_{j \in J \wedge J \subseteq [1, n] \wedge |J|=k}))$  returns  $\top$ .
- **Unforgeability of aggregate signatures**: If **Verify** $(m, \sigma^A)$  returns  $\top$ , then there exists a set  $J$ ,  $|J| = k$ , of share signatures such that, for each  $\sigma_j^S \in J$ , (1)  $\sigma_j^S \leftarrow \text{ShareSign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.

Importantly, there exist transparent threshold signature schemes such that the aggregate signatures have a size of  $O(\kappa \log(n))$  bits (e.g., [8]). Thus, proving that a group of a linear number of processes signed a certain message requires  $O(\kappa \log(n))$  bits to be transmitted rather than  $O(\kappa)$  bits for the non-transparent threshold signature schemes (Appendix B.3).

#### B.5. Aggregate signatures

A non-interactive  $n$ -aggregate signature scheme is a tuple of efficient (local) protocols (Keys, ShareSign, ShareVerify, Verify, Aggregate), where  $\text{Keys} = (\text{SK} = (sk_1, \dots, sk_n), \text{VK} = (vk_1, \dots, vk_n))$ , with:

- **VK**: a vector of verification keys stored by every correct process.
- **SK**: a vector of private key shares such that, for every correct process  $P_i$ ,  $P_i$  stores its private key share  $sk_i$ ;  $sk_i$  is hidden from the adversary.
- **ShareSign** $(m, sk_i)$ : a (potentially probabilistic) algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a private key share  $sk_i$  as the input. The algorithm outputs a signature share  $\sigma_i^S$  of (at most)  $\kappa$  bits.
- **ShareVerify** $(m, vk_j, \sigma_j^S)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , (2) the verification key  $vk_j$  of a process  $P_j$ , and (3) a signature share  $\sigma_j^S$  as the input. The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma_j^S$  is deemed as a valid signature.
- **Aggregate** $(m, \{\sigma_i\}_{i \in S \wedge S \subseteq [1, n]})$ : an algorithm that takes (1) a string  $m \in \text{String}$ , and (2) a subset  $S$  of any size of signature shares  $\{\sigma_i\}_{i \in S}$ . The algorithm outputs an aggregate signature  $\sigma^A$ .
- **Verify** $(m, \sigma^A, B)$ : a deterministic algorithm that takes (1) a string  $m \in \text{String}$ , (2) an aggregate signature  $\sigma^A$ , and (3) a bit mask  $B \in \{0, 1\}^n$ . The algorithm outputs  $\top$  or  $\perp$  depending on whether  $\sigma^A$  is deemed as a valid aggregate signature with reference to  $B$ .

The following properties hold:

- **Correctness of signature shares**: For every  $i \in [1, n]$ , **ShareVerify** $(m, vk_i, \text{ShareSign}(m, sk_i))$  returns  $\top$ .
- **Unforgeability of signature shares**: If **ShareVerify** $(m, vk_j, \sigma_j^S)$  returns  $\top$ , then (1)  $\sigma_j^S \leftarrow \text{ShareSign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.
- **Correctness of aggregate signatures**: Consider any string  $m \in \text{String}$  and any bit-mask  $B \in \{0, 1\}^n$ . The following holds: **Verify** $(m, \text{Aggregate}(\{\text{ShareSign}(m, sk_j)\}_{B[j]=1}), B)$  returns  $\top$ .
- **Unforgeability of aggregate signatures**: If **Verify** $(m, \sigma^A, B)$  returns  $\top$ , then, for every  $j \in [1, n]$  such that  $B[j] = 1$ , (1)  $\sigma_j^S \leftarrow \text{ShareSign}(m, sk_j)$  has been executed by  $P_j$ , or (2)  $P_j$  is faulty.

There exist transparent aggregate signature schemes such that the aggregate signatures have a size of  $O(\kappa)$  bits (e.g., [11]). The interface of an aggregate signature scheme is similar to the interface of the non-transparent threshold signature schemes (Appendix B.3). We emphasize two differences:

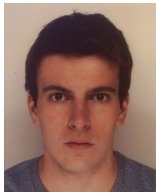
- **(Complexity)** An aggregate signature  $\sigma^A$  has to be associated with a bit-mask  $B$  of  $n$  bits (representing the subset of signers). Indeed, this bit-mask is an argument of the associated **Verify** protocol. Thus, proving that a group of a linear number of processes have signed a certain message requires  $O(\kappa + n)$  bits to be transmitted instead of  $O(\kappa)$  bits for the non-transparent threshold signature schemes (Appendix B.3).

- (Transparency) This scheme requires weaker assumptions for the setup (the same ones as the transparent threshold signature schemes defined in Appendix B.4). Indeed, the secret keys can be generated independently, and the correct processes have to agree on the associated verification keys (exactly as in a PKI).

## References

- [1] M. Abd-El-Malek, G.R. Ganger, G.R. Goodson, M.K. Reiter, J.J. Wylie, Fault-scalable Byzantine fault-tolerant services, in: A. Herbert, K.P. Birman (Eds.), Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23–26, 2005, ACM, 2005, pp. 59–74.
- [2] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, A. Tomescu, Reaching consensus for asynchronous distributed key generation, in: Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), 2021, pp. 363–373.
- [3] I. Abraham, D. Malkhi, K. Nayak, L. Ren, A. Spiegelman, Solida: a blockchain protocol based on reconfigurable Byzantine consensus, in: J. Aspnes, A. Bessani, P. Felber, J. Leitão (Eds.), 21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18–20, 2017, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 25.
- [4] I. Abraham, D. Malkhi, A. Spiegelman, Asymptotically optimal validated asynchronous Byzantine agreement, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC), 2019, pp. 337–346.
- [5] I. Abraham, T.-H.H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, E. Shi, Communication complexity of Byzantine agreement, revisited, in: P. Robinson, F. Ellen (Eds.), Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, ACM, New York, 2019, pp. 317–326.
- [6] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, D. Zage, Steward: scaling Byzantine fault-tolerant replication to wide area networks, IEEE Trans. Dependable Secure Comput. 7 (2010) 80–93, <https://doi.org/10.1109/TDSC.2008.53>.
- [7] M. Andrychowicz, S. Dziembowski, Pow-based distributed cryptography with no trusted setup, in: R. Gennaro, M. Robshaw (Eds.), Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 2015, Proceedings, Part II, Springer, 2015, pp. 379–399.
- [8] T. Attema, R. Cramer, M. Rambda, Compressed  $\Sigma$ -protocols for bilinear group arithmetic circuits and application to logarithmic transparent threshold signatures, in: M. Tibouchi, H. Wang (Eds.), Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part IV, Springer, 2021, pp. 526–556.
- [9] M. Ben-Or, S. Goldwasser, A. Wigderson, Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract), in: J. Simon (Ed.), Proceedings of the 20th Annual ACM Symposium on Theory of Computing, Chicago, Illinois, USA, May 2–4, 1988, ACM, 1988, pp. 1–10.
- [10] D. Boneh, M. Drijvers, G. Neven, Compact multi-signatures for smaller blockchains, in: Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II, 2018, pp. 435–464.
- [11] D. Boneh, C. Gentry, B. Lynn, H. Shacham, Aggregate and verifiably encrypted signatures from bilinear maps, in: E. Biham (Ed.), Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4–8, 2003, Proceedings, Springer, 2003, pp. 416–432.
- [12] G. Bracha, An asynchronous  $(n-1)/3$ -resilient consensus protocol, in: T. Kameda, J. Misra, J.G. Peters, N. Santoro (Eds.), Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27–29, 1984, ACM, 1984, pp. 154–162.
- [13] G. Bracha, Asynchronous Byzantine agreement protocols, Inf. Comput. 75 (1987) 130–143, [https://doi.org/10.1016/0890-5401\(87\)90054-X](https://doi.org/10.1016/0890-5401(87)90054-X).
- [14] E. Buchman, Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, Ph.D. thesis, 2016.
- [15] E. Buchman, J. Kwon, Z. Milosevic, The latest gossip on BFT consensus, Technical Report, arXiv:1807.04938, 2018.
- [16] V. Buterin, V. Griffith, Casper the friendly finality gadget, arXiv preprint, arXiv: 1710.09437, 2017.
- [17] C. Cachin, R. Guerraoui, L. Rodrigues, Introduction to Reliable and Secure Distributed Programming, Springer Science & Business Media, 2011.
- [18] C. Cachin, K. Kursawe, F. Petzold, V. Shoup, Secure and efficient asynchronous broadcast protocols, in: Proceedings of the Annual International Cryptology Conference (CRYPTO), Springer, 2001, pp. 524–541.
- [19] R. Canetti, Universally composable signature, certification, and authentication, in: 17th IEEE Computer Security Foundations Workshop, CSFW-17 2004, Pacific Grove, CA, USA, 28–30 June 2004, IEEE Computer Society, 2004, p. 219.
- [20] M. Castro, B. Liskov, Practical Byzantine fault tolerance, in: Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI), 1999, pp. 173–186.
- [21] S. Chaudhuri, More choices allow more faults: set consensus problems in totally asynchronous systems, Inf. Comput. 105 (1993) 132–158.
- [22] P. Civit, M.A. Dzulfikar, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, M. Vidigueira, Byzantine consensus is  $\Theta(n^2)$ : the Dolev-Reischuk bound is tight even in partial synchrony!, in: C. Scheidele (Ed.), 36th International Symposium on Distributed Computing, DISC 2022, Augusta, Georgia, USA, October 25–27, 2022, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 14.
- [23] P. Civit, S. Gilbert, V. Gramoli, Brief announcement: polygraph: accountable Byzantine agreement, in: Proceedings of the 34th International Symposium on Distributed Computing (DISC), 2020, 45.
- [24] P. Civit, S. Gilbert, V. Gramoli, Polygraph: accountable Byzantine agreement, in: Proceedings of the IEEE 41st International Conference on Distributed Computing Systems (ICDCS), 2021, pp. 403–413.
- [25] P. Civit, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, Z. Milosevic, A. Serendinschi, Crime and punishment in distributed Byzantine decision tasks, in: 42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 11–13, 2022, IEEE, 2022, <https://eprint.iacr.org/2022/121>.
- [26] T. Crain, V. Gramoli, M. Larrea, M. Raynal, DBFT: efficient leaderless Byzantine consensus and its application to blockchains, in: Proceedings of the IEEE 17th International Symposium on Network Computing and Applications (NCA), 2018, pp. 1–8.
- [27] T. Crain, C. Natoli, V. Gramoli, Red belly: a secure, fair and scalable open blockchain, in: Proceedings of the 42nd IEEE Symposium on Security and Privacy (SP), 2021, pp. 466–483.
- [28] S. Das, T. Yurek, Z. Xiang, A.K. Miller, L. Kokoris-Kogias, L. Ren, Practical asynchronous distributed key generation, in: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22–26, 2022, IEEE, 2022, pp. 2518–2534.
- [29] D. Dolev, R. Reischuk, Bounds on information exchange for Byzantine agreement, J. ACM 32 (1985) 191–204.
- [30] M. Drijvers, S. Gorbunov, G. Neven, H. Wee, Pixel: multi-signatures for consensus, in: S. Capkun, F. Roesner (Eds.), 29th USENIX Security Symposium, USENIX Security 2020, August 12–14, 2020, USENIX Association, 2020, pp. 2093–2110, <https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers>.
- [31] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (1988) 288–323.
- [32] M.J. Fischer, N.A. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (1985) 374–382, <https://doi.org/10.1145/3149.214121>.
- [33] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: scaling Byzantine agreements for cryptocurrencies, in: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28–31, 2017, ACM, 2017, pp. 51–68.
- [34] O. Goldreich, S. Micali, A. Wigderson, How to play any mental game or a completeness theorem for protocols with honest majority, in: A.V. Aho (Ed.), Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, ACM, 1987, pp. 218–229.
- [35] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlović, D.A. Seredinschi, The consensus number of a cryptocurrency, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC), 2019, pp. 307–316.
- [36] A. Haeberlen, P. Kuznetsov, P. Druschel, PeerReview: practical accountability for distributed systems, in: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP), ACM, 2007, pp. 175–188.
- [37] A. Haeberlen, P. Kuznetsov, The fault detection problem, in: Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS), Springer, 2009, pp. 99–114.
- [38] J. Katz, A. Miller, E. Shi, Pseudonymous secure computation from time-lock puzzles, IACR Cryptol. ePrint Arch., 857, <http://eprint.iacr.org/2014/857>, 2014.
- [39] R. Küsters, M. Tuengerthal, D. Rausch, The IITM model: a simple and expressive model for universal compossibility, J. Cryptol. 33 (2020) 1461–1584, <https://doi.org/10.1007/s00145-020-09352-1>.
- [40] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, in: Concurrency: the Works of Leslie Lamport, 2019, pp. 203–226.
- [41] B. Libert, M. Joye, M. Yung, Born and raised distributively: fully distributed non-interactive adaptively-secure threshold signatures with short shares, Theor. Comput. Sci. 645 (2016) 1–24, <https://doi.org/10.1016/j.tcs.2016.02.031>.
- [42] D. Malkhi, K. Nayak, L. Ren, Flexible Byzantine fault tolerance, in: L. Cavallaro, J. Kinder, X. Wang, J. Katz (Eds.), Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11–15, 2019, ACM, 2019, pp. 1041–1053.
- [43] U.M. Maurer, Modelling a public-key infrastructure, in: E. Bertino, H. Kurth, G. Martella, E. Montolivo (Eds.), Computer Security - ESORICS 96, 4th European Symposium on Research in Computer Security, Rome, Italy, September 25–27, 1996, Proceedings, Springer, 1996, pp. 325–350.

- [44] Z. Milosevic, M. Hutle, A. Schiper, Unifying Byzantine consensus algorithms with weak interactive consistency, in: *International Conference on Principles of Distributed Systems (OPODIS)*, Springer, 2009, pp. 300–314.
- [45] A. Momose, L. Ren, Multi-threshold Byzantine fault tolerance, in: Y. Kim, J. Kim, G. Vigna, E. Shi (Eds.), *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event, Republic of Korea, November 15 – 19, 2021, ACM, 2021, pp. 1686–1699.
- [46] O. Naor, I. Keidar, Expected linear round synchronization: the missing link for linear byzantine SMR, in: *34th International Symposium on Distributed Computing, DISC 2020, October 12–16, 2020, Virtual Conference, Schloss Dagstuhl – Leibniz-Zentrum für Informatik*, 2020, 26.
- [47] A.R. Pedrosa, V. Gramoli, Trap: the bait of rational players to solve Byzantine consensus, in: *Proceedings of the 17th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2022.
- [48] P. Sheng, G. Wang, K. Nayak, S. Kannan, P. Viswanath, BFT protocol forensics, in: *Computer and Communication Security (CCS)*, 2021.
- [49] V. Shoup, Practical threshold signatures, in: B. Preneel (Ed.), *Advances in Cryptology – EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques*, Bruges, Belgium, May 14–18, 2000, Proceeding, Springer, 2000, pp. 207–220.
- [50] L.F. de Souza, P. Kuznetsov, T. Rieutord, S. Tucci Piergiovanni, Accountability and reconfiguration: self-healing lattice agreement, in: *Proceedings of the 25th International Conference on Principles of Distributed Systems (OPODIS)*, 2021, 25.
- [51] A. Spiegelman, In search for an optimal authenticated Byzantine agreement, in: *Proceedings of the 35th International Symposium on Distributed Computing (DISC)*, 2021, 38.
- [52] G.S. Veronese, M. Correia, A.N. Bessani, L.C. Lung, P. Veríssimo, Efficient Byzantine fault-tolerance, *IEEE Trans. Comput.* 62 (2013) 16–30, <https://doi.org/10.1109/TC.2011.221>.
- [53] M. Yin, D. Malkhi, M.K. Reiter, G. Golan-Gueta, I. Abraham, HotStuff: BFT consensus with linearity and responsiveness, in: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, 2019, pp. 347–356.



**Pierre Civi** received the M.S. in Aerospace Engineering from the ISAE-Supaéro in 2019, and a Ph.D. in Computer Science from the Sorbonne University, France, in 2022. He has been visiting scholar at Georgia Tech, at the University of Sydney and at EPFL. His current research interests include distributed computing, secure multi-party computation, automata theory and formal verification.



**Seth Gilbert** is a Dean's Chair Associate Professor at the National University of Singapore. He received his PhD from MIT, and spent several years as a postdoc at EPFL. His work includes research on backoff protocols, dynamic graph algorithms, wireless networks, robust scheduling, and the occasional blockchain. In fact, Seth's research focuses on algorithmic issues of robustness and scalability, wherever they may arise.



**Vincent Gramoli** is the Founder of Redbelly Network and the head of the Concurrent Systems Research Group at the University of Sydney. In the past, Gramoli has been affiliated with INRIA, Cornell, CSIRO and EPFL. He received a Future Fellowship from the Australian Research Council, his PhD from Université de Rennes and his Habilitation from Sorbonne University. His expertise is in distributed computing and security, his book, *Blockchain Scalability and its Foundations in Distributed Systems*, is published by Springer and his blockchain scalability course is followed by more than five thousand students.



**Rachid Guerraoui** is professor in Computer Science at EPFL where he leads the Distributed Computing Laboratory. He worked in the past with École des Mines de Paris, CEA Saclay, HP Labs in Palo Alto and MIT. He has been elected ACM Fellow and Professor of the College de France. He was awarded a Senior ERC Grant and a Google Focused Award.



**Jovan Komatovic** is a PhD student in computer science at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. Previously, he obtained his Bachelor's and Master's degrees from University of Belgrade, Serbia. His scientific interests lie in theoretical distributed computing with a particular focus on Byzantine fault tolerance.