On the Validity of Consensus

PIERRE CIVIT, Sorbonne University, France
SETH GILBERT, NUS Singapore, Singapore
RACHID GUERRAOUI, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
JOVAN KOMATOVIC, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
MANUEL VIDIGUEIRA, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

The Byzantine consensus problem involves n processes, out of which $t < n$ could be faulty and behave arbitrarily. Three properties characterize consensus: (1) termination, requiring correct (non-faulty) processes to eventually reach a decision, (2) agreement, preventing them from deciding different values, and (3) validity, precluding "unreasonable" decisions. But, what is a reasonable decision? Strong validity, a classical property, stipulates that, if all correct processes propose the same value, only that value can be decided. Weak validity, another established property, stipulates that, if all processes are correct and they propose the same value, that value must be decided. The space of possible validity properties is vast. However, their impact on consensus remains unclear.

This paper addresses the question of which validity properties allow Byzantine consensus to be solvable with partial synchrony, and at what cost. First, we determine necessary and sufficient conditions for a validity property to make the consensus problem solvable; we say that such validity properties are *solvable*. Notably, we prove that, if $n \leq 3t$, all solvable validity properties are *trivial* (there exists an always-admissible decision). Furthermore, we show that, with any non-trivial (and solvable) validity property, consensus requires $\Omega(t^2)$ messages. This extends the seminal Dolev-Reischuk bound, originally proven for strong validity, to *all* non-trivial validity properties. Lastly, we give a general Byzantine consensus algorithm, we call Universal, for *any* solvable (and non-trivial) validity property. Importantly, Universal incurs $O(n^2)$ message complexity. Thus, together with our lower bound, Universal implies a fundamental result in partial synchrony: with $t \in \Omega(n)$, the message complexity of all (non-trivial) consensus variants is $\Theta(n^2)$.

1 INTRODUCTION

Consensus [50] is the cornerstone of state machine replication (SMR) [1, 8, 9, 21, 46, 47, 57, 61, 77], as well as various distributed protocols [13, 37, 39, 40]. Recently, it has received a lot of attention with the advent of blockchain systems [5, 6, 17, 26, 28, 38, 55]. The consensus problem is posed in a system of n processes, out of which $t < n$ can be *faulty*, and the rest are *correct*. Each correct process proposes a value, and consensus enables correct processes to decide on a common value. In this paper, we consider Byzantine [50] consensus, where faulty processes can behave arbitrarily. While the exact definition of the problem might vary, two properties are always present: (1) *termination*, requiring correct processes to eventually decide, and (2) *agreement*, preventing them from deciding different values. It is not hard to devise an algorithm that satisfies only these two properties: every correct process decides the same, predetermined value. However, this algorithm is vacuous. To preclude such trivial solutions and render consensus meaningful, an additional property is required: *validity*, defining which decisions are admissible.

The many faces of validity. The literature contains many flavors of validity [4, 23, 24, 28, 36, 45, 59, 73, 74, 78]. One of the most studied properties is *Strong Validity* [4, 23, 28, 45], stipulating that, if all correct processes propose the same value, only that value can be decided. Another common property is *Weak Validity* [23, 24, 78], affirming that, if all processes are correct and propose the same value, that value must be decided. In fact, many other variants of the property have been considered [36, 59, 73, 74]. While validity may appear as an inconspicuous property, its exact definition has a big impact on consensus algorithms. For example, the seminal Dolev-Reischuk bound [30] states that any solution to consensus with *Strong Validity* incurs a quadratic number of messages; it was recently proven that the bound is tight [23, 52, 62]. In contrast, while there have been several improvements to the performance of consensus with *Weak Validity* over the last 40 years [23, 52, 78], the (tight) lower bound on message complexity remains unknown. (Although the bound is conjectured to be the same as for *Strong Validity*, this has yet to be formally proven.) Many other fundamental questions remain unanswered:

- What does it take for a specific validity property to make consensus solvable?
- What are the (best) upper and lower bounds on the message complexity of consensus with any specific validity property?
- Is there a hierarchy of validity properties (e.g., a "strongest" validity property)?

To the best of our knowledge, no in-depth study of the validity property has ever been conducted, despite its importance [2].

Contributions. We propose a precise mathematical formalism for the analysis of validity properties. We define a validity property as a mapping from assignments of proposals into admissible decisions. Although simple, our formalism enables us to determine the exact impact of validity on the solvability and complexity of consensus in the classical partially synchronous model [32], and answer the aforementioned open questions. Namely, we provide the following contributions:

- We classify all validity properties into solvable and unsolvable ones. (If a validity property makes consensus solvable, we say that the property itself is *solvable*.) Specifically, for $n \leq 3t$, we show that only *trivial* validity properties (for which there exists an always-admissible decision) are solvable. In the case of $n > 3t$, we define the *similarity condition*, which we prove to be necessary and sufficient for a validity property to be solvable.
- We prove that all non-trivial (and solvable) validity properties require $\Omega(t^2)$ exchanged messages. This result extends the Dolev-Reischuk bound [30], proven only for *Strong Validity*, to *all* "reasonable" validity properties.
- Finally, we present Universal, a general consensus algorithm for all solvable (and non-trivial) validity properties. Importantly, assuming a threshold signature scheme, Universal exchanges $O(n^2)$ messages. Thus, together with our lower bound, Universal implies a fundamental result in partial synchrony: given $t \in \Omega(n)$, all (non-trivial) consensus variants have $\Theta(n^2)$ message complexity. Figure 1 summarizes our findings.

Technical overview. In our formalism, we use the notion of *input configuration* that denotes an assignment of proposals to correct processes. For example, $\big[(P_1, v), (P_2, v), (P_3, v)\big]$ represents an input configuration by which (1) only processes P_1, P_2, and P_3 are correct, and (2) processes P_1, P_2, and P_3 propose v. First, we define a similarity relation between input configurations: two input configurations are similar if and only if (1) they have (at least) one process in common, and (2) for every common process, the process's proposal is identical in both input configurations. For example, an input configuration $c = \big[(P_1, 0), (P_2, 1)\big]$ is similar to $\big[(P_1, 0), (P_3, 0)\big]$, but not to $\big[(P_1, 0), (P_2, 0)\big]$. We observe that all similar input configurations must have an admissible value in common; we call this *canonical similarity*. Let us illustrate why a common admissible value must exist. Consider the aforementioned similar input configurations $c = \big[(P_1, 0), (P_2, 1)\big]$ and $c' = \big[(P_1, 0), (P_3, 0)\big]$. If there is no common admissible value for c and c', consensus cannot be solved: process P_1 cannot distinguish (1) an execution in which P_2 is correct, and P_3 is faulty and silent, from (2) an execution in which P_2 is faulty, but behaves correctly, and P_3 is correct, but slow. Thus, P_1 cannot conclude whether it needs to decide an admissible value for c or for c'. Canonical similarity is a critical

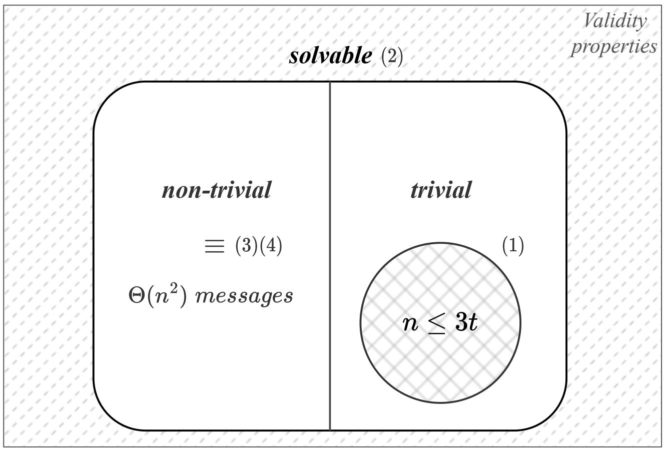

Fig. 1. Illustration of our results: (1) with $n \leq 3t$, all solvable validity properties are trivial; (2) the exact set of solvable validity properties (as determined by our necessary and sufficient conditions); (3) all non-trivial (and solvable) validity properties require $\Omega(t^2)$ exchanged messages; (4) for any non-trivial (and solvable) validity property, there exists a consensus algorithm with $O(n^2)$ message complexity.

intermediate result that we use extensively throughout the paper (even if it does not directly imply any of our results).

In our proof of triviality with $n \leq 3t$, we intertwine the classical partitioning argument [51] with our canonical similarity result. Namely, we show that, for any input configuration, there exists an execution in which the same value x is decided, making x an always-admissible value. For our lower bound, while following the idea of the original proof [3, 30], we rely on canonical similarity to prove the bound for all solvable and non-trivial validity properties. Finally, we design Universal by relying on vector consensus [27, 31, 69, 76], a problem in which processes agree on the proposals of $n - t$ processes: when a correct process decides a vector *vec* of $n - t$ proposals (from vector consensus), it decides (from Universal) the common admissible value for all input configurations similar to *vec*. For example, consider an execution which corresponds to an input configuration c. If, in this execution, a correct process decides a vector *vec* from vector consensus, it is guaranteed that *vec* is similar to c (the proposals of correct processes are identical in c and in *vec*). Hence, deciding (from Universal) the common admissible value for all input configurations similar to *vec* guarantees that the decided value is admissible according to c.

Roadmap. We provide an overview of related work in §2. In §3, we specify the system model (§3.1), define the consensus problem (§3.2), describe our formalism for validity properties (§3.3), and present canonical similarity (§3.4). We define the necessary conditions for the solvability of validity properties in §4. In §5, we prove a quadratic lower bound on message complexity for all non-trivial (and solvable) validity properties (§5.1), and introduce Universal, a general consensus algorithm for any solvable (and non-trivial) validity property (§5.2). We conclude the paper in §6. The appendix contains (1) detailed proofs of our results, (2) omitted algorithms, and (3) a proposal for how to extend our formalism to accommodate for blockchain-specific validity properties.

2 RELATED WORK

Solvability of consensus. The consensus problem has been thoroughly investigated in a variety of system settings and failure models. It has been known (for long) that consensus can be solved in a synchronous setting, both with crash [19, 56, 70] and arbitrary failures [4, 48, 62, 70, 72]. In an asynchronous environment, however, consensus cannot be solved deterministically even if a single process can fail, and it does so only by crashing; this is the seminal FLP impossibility result [35].

A traditional way of circumventing the FLP impossibility result is *randomization* [7, 10, 11, 33, 54], where termination of consensus is not ensured deterministically. Another well-established approach to bypass the FLP impossibility is to strengthen the communication model with *partial synchrony* [32]: communication is asynchronous until some unknown time, and then it becomes synchronous. The last couple of decades have produced many partially synchronous consensus algorithms [18, 21, 23, 24, 28, 32, 49, 52, 56, 78].

Another line of research has consisted in weakening the definition of consensus to make it deterministically solvable under asynchrony. In the *condition-based* approach [64], the specification of consensus is relaxed to require termination only if the assignment of proposals satisfies some predetermined conditions. The efficiency of this elegant approach has been studied further in [66], then extended to the synchronous setting [65, 79], as well as to the k-set agreement problem [41, 67].

Solvability of general decision problems. A distributed decision problem has been defined in [25, 44, 60] as a mapping from input assignments to admissible decisions. Our validity formalism is of the same nature, and it is inspired by the aforementioned specification of decision problems.

The solvability of decision problems has been thoroughly studied in asynchronous, crash-prone settings. It was shown in [63] that the FLP impossibility result [35] can be extended to many decision problems. In [15], the authors defined necessary and sufficient conditions for a decision problem to be solvable with a single crash failure. The asynchronous solvability of problems in which crash failures occur at the very beginning of an execution was studied in [75]. Necessary and sufficient conditions for a decision problem to be solvable in a randomized manner were given in [22]. The topology-based approach on studying the solvability of decision problems in asynchrony has proven to be extremely effective, both for crash [42, 43, 71] and arbitrary failures [42, 60]. Our results follow the same spirit as many of these approaches; however, we study the deterministic solvability and complexity of all consensus variants in a partially synchronous environment.

Validity of consensus. Various validity properties have been associated with the consensus problem (beyond the aforementioned *Strong Validity* and *Weak Validity*). *Correct-Proposal Validity* [36, 73] states that a value decided by a correct process must have been proposed by a correct process. *Median Validity* [74] is another validity property proposed in the context of synchronous consensus, requiring the decision to be close to the median of the proposals of correct processes. *Interval Validity* [59], on the other hand, requires the decision to be close to the k-th smallest proposal of correct processes. The advent of blockchain technologies has resurged the concept of *External Validity* [18, 20, 78]. This property requires the decided value to satisfy a predetermined predicate, typically asserting whether the decided value follows the rules of a blockchain system (e.g., no double-spending). (For pedagogical purposes, we considered a simple formalism to express basic validity properties and derive our results. To express *External Validity*, which is out of the scope of the paper, we propose an extension of our formalism in Appendix D.)

In interactive consistency [12, 34, 58], correct processes agree on the proposals of all correct processes. Given that the problem is impossible in a non-synchronous setting, a weaker variant has been considered: vector consensus [27, 31, 69, 76]. Here, processes need to agree on a vector of proposals which does not necessarily include the proposals of *all* correct processes. Interactive consistency and vector consensus can be seen as specific consensus problems with a validity property requiring that, if a decided vector contains a proposal v of a correct process, that correct process has indeed proposed value v. The design of UNIVERSAL, our general consensus algorithm for any solvable (and non-trivial) validity property, demonstrates that any non-trivial flavor of consensus, which is solvable in partial synchrony, can be solved using vector consensus (see §5.2).

3 PRELIMINARIES

In this section, we present the computational model (§3.1), recall the consensus problem (§3.2), formally define validity properties (§3.3), and introduce canonical similarity (§3.4).

3.1 Computational Model

Processes. We consider a system $\Pi = \{P_1, P_2, ..., P_n\}$ of n processes. At most t ($0 < t < n$) processes can be *faulty*: these processes can exhibit arbitrary behavior. A non-faulty process is said to be *correct*. Processes communicate by exchanging messages over an authenticated point-to-point network. The communication network is *reliable*: if a correct process sends a message to a correct process, the message is eventually received.

Executions. Given an algorithm $\mathcal{A}$, $\mathit{execs}(\mathcal{A})$ denotes the set of all executions of $\mathcal{A}$. Furthermore, $\mathit{Corr}_{\mathcal{A}}(\mathcal{E})$ denotes the set of correct processes in an execution $\mathcal{E} \in \mathit{execs}(\mathcal{A})$. We say that an execution $\mathcal{E} \in \mathit{execs}(\mathcal{A})$ is *canonical* if and only if no faulty process takes any computational step in $\mathcal{E}$; note that faulty processes do not send any message in a canonical execution.

Partial synchrony. We consider the standard partially synchronous model [32]. For every execution of the system, there exists a Global Stabilization Time (GST) and a positive duration δ such that message delays are bounded by δ after GST. GST is not known to processes, whereas δ is. We assume that all correct processes start executing their local algorithm before or at GST.

Cryptographic primitives. In one variant of the Universal algorithm, we assume a (k, n)-threshold signature scheme [53] (see §5.2). In fact, this variant relies on a closed-box consensus algorithm which internally utilizes threshold signatures. In a threshold signature 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; we denote by $\langle m \rangle_{\sigma_i}$ a message (partially) signed by the process P_i. Moreover, a signature can be verified by other processes. Finally, a set of signatures for a message m from k (the threshold) distinct processes can be combined into a single threshold signature for m, which proves that k processes have signed m.

Message complexity. Let $\mathcal{A}$ be any algorithm and let $\mathcal{E} \in \mathit{execs}(\mathcal{A})$ be any execution of $\mathcal{A}$. The message complexity of $\mathcal{E}$ is the number of messages sent by correct processes during $[\text{GST}, \infty]$.

The *message complexity* of $\mathcal{A}$ is defined as

$$\max_{\mathcal{E} \in \mathit{execs}(\mathcal{A})} \bigg\{ \text{message complexity of } \mathcal{E} \bigg\}.$$

3.2 Consensus

We denote by $\mathcal{V}_I$ the set of values processes can propose, and by $\mathcal{V}_O$ the set of values processes can decide. The consensus[1] problem exposes the following interface:

- **request** propose($v \in \mathcal{V}_I$): a process proposes a value v.
- **indication** decide($v' \in \mathcal{V}_O$): a process decides a value v'.

A correct process proposes and decides at most once. Consensus requires the following properties:

- *Termination:* Every correct process eventually decides.
- *Agreement:* No two correct processes decide different values.

[1]Throughout the entire paper, we use "consensus" and "Byzantine consensus" interchangeably.

If the consensus problem was completely defined by *Termination* and *Agreement*, a trivial solution would exist: processes decide on a default value. Therefore, the specification of consensus additionally includes a validity property, which connects the proposals of correct processes to admissible decisions, precluding the aforementioned trivial solutions.

3.3 Validity

In a nutshell, our specification of a validity property includes a set of assignments of proposals to correct processes, and, for each such assignment, a corresponding set of admissible decisions.

We start by defining a *process-proposal* pair as a pair (P, v), where (1) $P \in \Pi$ is a process, and (2) $v \in \mathcal{V}_I$ is a proposal. Given a process-proposal pair $pp = (P, v)$, $\mathsf{proposal}(pp) = v$ denotes the proposal associated with pp.

An *input configuration* is a tuple $\left[pp_1, pp_2, ..., pp_x\right]$ of x process-proposal pairs, where (1) $n - t \leq x \leq n$, and (2) every process-proposal pair is associated with a distinct process. Intuitively, an input configuration represents an assignment of proposals to correct processes. For example, an input configuration $\left[(P_1, v), (P_2, v), (P_3, v), (P_4, v), (P_5, v)\right]$ describes an execution in which (1) only processes P_1, P_2, P_3, P_4 and P_5 are correct, and (2) all of them propose the same value v.

We denote by $\mathcal{I}$ the set of all input configurations. Furthermore, for every $x \in [n - t, n]$, $\mathcal{I}_x \subset \mathcal{I}$ denotes the set of input configurations with *exactly* x process-proposal pairs. For every input configuration $c \in \mathcal{I}$, we denote by $c[i]$ the process-proposal pair associated with process P_i; if such a process-proposal pair does not exist, $c[i] = \bot$. Finally, $\pi(c) = \{P_i \in \Pi \mid c[i] \neq \bot\}$ denotes the set of all processes included in c.

Given (1) an execution $\mathcal{E}$ of an algorithm $\mathcal{A}$, where $\mathcal{A}$ exposes the $\mathsf{propose}(\cdot)/\mathsf{decide}(\cdot)$ interface, and (2) an input configuration $c \in \mathcal{I}$, we say that $\mathcal{E}$ *corresponds* to c if and only if (1) $\pi(c) = Corr_{\mathcal{A}}(\mathcal{E})$, and (2) for every process $P_i \in Corr_{\mathcal{A}}(\mathcal{E})$, P_i's proposal in $\mathcal{E}$ is $\mathsf{proposal}(c[i])$. We denote by $\mathsf{corresponding}(\mathcal{E}) = c$ the input configuration to which $\mathcal{E}$ corresponds.

Finally, we define a validity property val as a function $val : \mathcal{I} \to 2^{\mathcal{V}_O}$ such that, for every input configuration $c \in \mathcal{I}$, $val(c) \neq \emptyset$. An algorithm $\mathcal{A}$, where $\mathcal{A}$ exposes the $\mathsf{propose}(\cdot)/\mathsf{decide}(\cdot)$ interface, *satisfies* a validity property val if and only if, in every execution $\mathcal{E} \in execs(\mathcal{A})$, no correct process decides a value $v' \notin val\big(\mathsf{corresponding}(\mathcal{E})\big)$. That is, an algorithm satisfies a validity property if and only if correct processes decide only admissible values.

Weak Validity & Strong Validity in our formalism. To illustrate our formalism, we describe how it can be used to express these two properties. For both, $\mathcal{V}_I = \mathcal{V}_O$. *Weak Validity* can be expressed as:

$$val(c) = \begin{cases} \{v\}, & \text{if } (\pi(c) = \Pi) \land (\forall P_i \in \pi(c) : \mathsf{proposal}(c[i]) = v) \\ \mathcal{V}_O, & \text{otherwise} \end{cases}$$

whereas *Strong Validity* can be expressed as:

$$val(c) = \begin{cases} \{v\}, & \text{if } \forall P_i \in \pi(c) : \mathsf{proposal}(c[i]) = v \\ \mathcal{V}_O, & \text{otherwise} \end{cases}$$

Consensus algorithms. An algorithm $\mathcal{A}$ solves consensus with a validity property val if and only if the following holds:

- $\mathcal{A}$ exposes the $\mathsf{propose}(\cdot)/\mathsf{decide}(\cdot)$ interface, and
- $\mathcal{A}$ satisfies *Termination*, *Agreement* and the validity property *val*.

Lastly, we formally define the notion of a solvable validity property.

Definition 1 (Solvable validity property)**.** We say that a validity property *val* is *solvable* if and only if there exists an algorithm which solves consensus with *val*.

3.4 Canonical Similarity

In this subsection, we introduce *canonical similarity*, a crucial intermediate result. In order to do so, we first define an important relation between input configurations, that of *similarity*.

Similarity. We define the similarity relation ("$\sim$") between input configurations:

$$\forall c_1, c_2 \in \mathcal{I} : c_1 \sim c_2 \iff (\pi(c_1) \cap \pi(c_2) \neq \emptyset) \land (\forall P_j \in \pi(c_1) \cap \pi(c_2) : c_1[j] = c_2[j]).$$

In other words, c_1 is similar to c_2 if and only if (1) c_1 and c_2 have at least one process in common, and (2) for every common process, the process's proposal is identical in both input configurations. For example, $c = \big[(P_1, 0), (P_2, 1), (P_3, 0)\big]$ is similar to $\big[(P_1, 0), (P_3, 0)\big]$, whereas c is not similar to $\big[(P_1, 0), (P_2, 0), (P_3, 0)\big]$. Note that the similarity relation is symmetric (for every pair $c_1, c_2 \in \mathcal{I}$, $c_1 \sim c_2 \Leftrightarrow c_2 \sim c_1$) and reflexive (for every $c \in \mathcal{I}$, $c \sim c$).

For every input configuration $c \in \mathcal{I}$, we define its *similarity set*, denoted by $sim(c)$:

$$sim(c) = \{c' \in \mathcal{I} \mid c' \sim c\}.$$

The canonical similarity result. Let $\mathcal{A}$ be an algorithm which solves consensus with some validity property *val*. Our canonical similarity result states that $\mathcal{A}$, in any canonical execution which corresponds to some input configuration c, can only decide a value which is admissible for *all* input configurations similar to c. Informally, the reason is that correct processes cannot distinguish silent faulty processes from slow correct ones.

Lemma 1 (Canonical similarity)**.** Let *val* be any solvable validity property and let $\mathcal{A}$ be any algorithm which solves the consensus problem with *val*. Let $\mathcal{E} \in execs(\mathcal{A})$ be any canonical execution and let $\mathsf{corresponding}(\mathcal{E}) = c$, for some input configuration $c \in \mathcal{I}$. If a value $v' \in \mathcal{V}_O$ is decided by a correct process in $\mathcal{E}$, then $v' \in \bigcap_{c' \in sim(c)} val(c')$.

Proof. We prove the lemma by contradiction. Suppose that $v' \notin \bigcap_{c' \in sim(c)} val(c')$. Hence, there exists an input configuration $c' \in sim(c)$ such that $v' \notin val(c')$. Let $\mathcal{E}_P$ denote any infinite continuation of $\mathcal{E}$ such that $\mathsf{corresponding}(\mathcal{E}_P) = c$. Let P be any process such that $P \in \pi(c') \cap \pi(c)$; such a process exists as $c' \sim c$. As $\mathcal{A}$ satisfies *Termination* and *Agreement*, $\mathcal{E}_P$ is an infinite execution, and P is correct in $\mathcal{E}_P$, P decides v' in $\mathcal{E}_P$.

We construct another execution $\mathcal{E}' \in execs(\mathcal{A})$ such that $\mathsf{corresponding}(\mathcal{E}') = c'$:

(1) $\mathcal{E}'$ is identical to $\mathcal{E}_P$ until process P decides v'.
(2) All processes in $\Pi \setminus \pi(c')$ are faulty in $\mathcal{E}'$ (they behave correctly until P has decided), and all processes in $\pi(c')$ are correct.
(3) After P has decided, processes in $\pi(c') \setminus \pi(c)$ "wake up" with the proposals specified in c'.
(4) GST is set to after all processes in $\pi(c')$ have taken a computational step.

For every process $P_i \in \pi(c') \cap \pi(c)$, the proposal of P_i in $\mathcal{E}'$ is $\mathsf{proposal}(c'[i])$; recall that $c'[i] = c[i]$ as $c' \sim c$. Moreover, for every process $P_j \in \pi(c') \setminus \pi(c)$, the proposal of P_j in $\mathcal{E}'$ is $\mathsf{proposal}(c'[j])$ (due to the step 3 of the construction). Hence, $\mathsf{corresponding}(\mathcal{E}') = c'$. Furthermore, process P, which is correct in $\mathcal{E}'$, decides a value $v' \notin val(c')$ (due to the step 1 of the construction). Thus, we reach a contradiction with the fact that $\mathcal{A}$ satisfies *val*, which proves the lemma. □

4 NECESSARY SOLVABILITY CONDITIONS

We give in this section necessary conditions for the solvability of validity properties. We start by focusing on the case of $n \leq 3t$: we prove that, if $n \leq 3t$, all solvable validity properties are trivial (§4.1). Then, we consider the case of $n > 3t$: we formally define the similarity condition, and prove its necessity for solvable validity properties (§4.2).

4.1 Triviality of Solvable Validity Properties if $n \leq 3t$

Some validity properties, such as *Weak Validity* and *Strong Validity*, are known to be unsolvable for $n \leq 3t$ [32]. This seems to imply a split of validity properties depending on the resiliency threshold. We prove that such a split indeed exists for $n \leq 3t$, and, importantly, that it applies to *all* solvable validity properties. Implicitly, this means that there is no "useful" relaxation of the validity property that can tolerate $t > \lfloor n/3 \rfloor$ failures. Concretely, we prove the following theorem:

Theorem 1. *If a validity property is solvable with $n \leq 3t$, then the validity property is trivial. i.e., there exists a value $v' \in \mathcal{V}_O$ such that $v' \in \bigcap_{c \in \mathcal{I}} val(c)$.*

Before presenting the proof of the theorem, we introduce the *compatibility relation* between input configurations, which we use throughout this subsection.

Compatibility. We define the compatibility relation ("$\diamond$") between input configurations:

$$\forall c_1, c_2 \in \mathcal{I} : c_1 \diamond c_2 \iff (|\pi(c_1) \cap \pi(c_2)| \leq t) \land (\exists P \in \pi(c_1) \setminus \pi(c_2)) \land (\exists Q \in \pi(c_2) \setminus \pi(c_1)).$$

That is, c_1 is compatible with c_2 if and only if (1) there are at most t processes in common, (2) there exists a process which belongs to c_1 and does not belong to c_2, and (3) there exists a process which belongs to c_2 and does not belong to c_1. For example, when $n = 3$ and $t = 1$, $c = \big[(P_1, 0), (P_2, 0)\big]$ is compatible with $\big[(P_1, 1), (P_3, 1)\big]$, whereas c is not compatible with $\big[(P_1, 1), (P_2, 1), (P_3, 1)\big]$. Observe that the compatibility relation is symmetric and irreflexive.

Proof of Theorem 1. Throughout the rest of the subsection, we fix any validity property val which is solvable with $n \leq 3t$. Moreover:

- We assume that $n \leq 3t$.
- We fix any algorithm $\mathcal{A}$ which solves consensus with val.
- We fix any input configuration $base \in \mathcal{I}_{n-t}$ with exactly $n - t$ process-proposal pairs.
- We fix any infinite canonical execution $\mathcal{E}_{base} \in execs(\mathcal{A})$ such that $\text{corresponding}(\mathcal{E}_{base}) = base$. As $\mathcal{A}$ satisfies *Termination* and val, and $\mathcal{E}_{base}$ is infinite, some value $v_{base} \in val(base)$ is decided by a correct process in $\mathcal{E}_{base}$.

To prove Theorem 1, it suffices to prove that val is trivial. Our proof leverages our formalism, specifically the aforementioned compatibility relation and the established canonical similarity result (§3.4), and combines the formalism with the classical partitioning argument [50].

We start by showing that only v_{base} can be decided in any canonical execution which corresponds to any input configuration compatible with $base$. If a value different from v_{base} is decided, the adversary would be able to cause a disagreement by partitioning processes into two disagreeing groups. We delegate the formal proof of the following lemma to Appendix A.

Lemma 2. Let $c \in \mathcal{I}$ be any input configuration such that $c \diamond base$. Let $\mathcal{E}_c \in execs(\mathcal{A})$ be any canonical execution such that $\text{corresponding}(\mathcal{E}_c) = c$. If a value $v_c \in \mathcal{V}_O$ is decided by a correct process in $\mathcal{E}_c$, then $v_c = v_{base}$.

Observe that the proposals of an input configuration compatible with $base$ do not influence the decision: given an input configuration $c \in \mathcal{I}$, $c \diamond base$, only v_{base} can be decided in any canonical execution which corresponds to c, *irrespectively* of the proposals.

Next, we prove a direct consequence of Lemma 2: for every input configuration $c_n \in \mathcal{I}_n$, there exists an execution $\mathcal{E}_n$ such that (1) $\mathcal{E}_n$ corresponds to c_n, and (2) v_{base} is decided in $\mathcal{E}_n$. Below, we give a proof sketch of the claim; the formal proof can be found in Appendix A.

Lemma 3. For every input configuration $c_n \in \mathcal{I}_n$, there exists an execution $\mathcal{E}_n \in execs(\mathcal{A})$ such that (1) $\text{corresponding}(\mathcal{E}_n) = c_n$, and (2) v_{base} is decided in $\mathcal{E}_n$.

Proof Sketch. Fix any input configuration $c_n \in \mathcal{I}_n$. There exists an input configuration $c_{n-t} \in \mathcal{I}_{n-t}$ such that (1) for every process $P \in \pi(c_n) \cap \pi(c_{n-t})$, proposals of P in c_n and c_{n-t} are identical, and (2) $c_{n-t} \diamond base$. Hence, v_{base} is decided in any infinite canonical execution $\mathcal{E}_{n-t}$ which corresponds to c_{n-t} (by Lemma 2). Thus, by "waking up" processes in $\pi(c_n) \setminus \pi(c_{n-t})$ after v_{base} is decided in $\mathcal{E}_{n-t}$, we build an execution $\mathcal{E}_n$ such that (1) $\mathcal{E}_n$ corresponds to c_n, and (2) v_{base} is decided in $\mathcal{E}_n$. □

We are now ready to prove that *val* is trivial. We do so by showing that, for every input configuration $c \in \mathcal{I}$, $v_{base} \in val(c)$.

Lemma 4. Validity property *val* is trivial.

Proof. We fix any input configuration $c \in \mathcal{I}$. Let us distinguish two possible scenarios:

- Let $c \in \mathcal{I}_n$. There exists an execution $\mathcal{E}_c \in execs(\mathcal{A})$ such that (1) corresponding$(\mathcal{E}_c) = c$, and (2) v_{base} is decided in $\mathcal{E}_c$ (by Lemma 3). As $\mathcal{A}$ satisfies *val*, $v_{base} \in val(c)$.
- Let $c \notin \mathcal{I}_n$. We construct an input configuration $c_n \in \mathcal{I}_n$ in the following way:
  (1) Let $c_n \leftarrow c$.
  (2) For every process $P \notin \pi(c)$, $(P, \text{any proposal})$ is included in c_n.
  Due to the construction of c_n, $c_n \sim c$. By Lemma 3, there exists an execution $\mathcal{E}_n \in execs(\mathcal{A})$ such that (1) corresponding$(\mathcal{E}_n) = c_n$, and (2) v_{base} is decided in $\mathcal{E}_n$. Note that $\mathcal{E}_n$ is a canonical execution. Therefore, canonical similarity ensures that $v_{base} \in val(c)$ (Lemma 1).

In both possible cases, $v_{base} \in val(c)$. Thus, the theorem. □

Lemma 4 concludes the proof of Theorem 1, as Lemma 4 proves that *val*, *any* solvable validity property with $n \leq 3t$, is trivial. Figure 2 depicts the proof of Theorem 1. Since this subsection showed that no useful consensus variant exists when $n \leq 3t$, the rest of the paper focuses on the case of $n > 3t$.

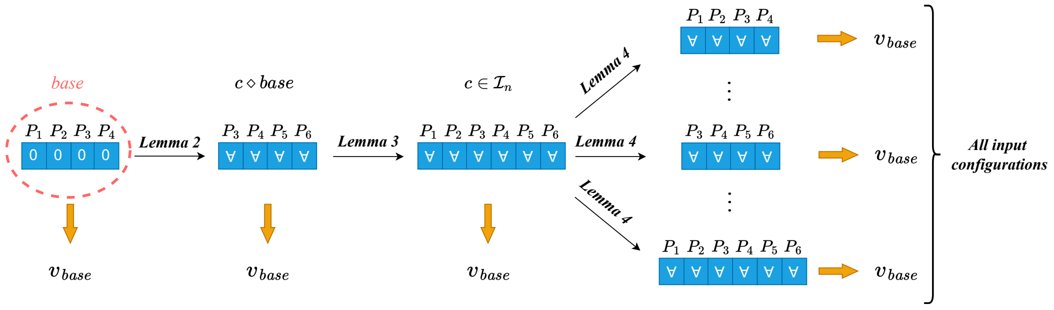

Fig. 2. Theorem 1: Overview of the proof in the case $n = 6$, $t = 2$, and $base = \big[(P_1, 0), (P_2, 0), (P_3, 0), (P_4, 0)\big]$.

4.2 Similarity Condition: Necessary Solvability Condition

This subsection defines the similarity condition, and proves its necessity for solvable properties.

Definition 2 (Similarity condition). A validity property *val* satisfies the *similarity condition* (C_S, in short) if and only if there exists a computable function $\Lambda : \mathcal{I}_{n-t} \to \mathcal{V}_O$ such that:

$$\forall c \in \mathcal{I}_{n-t} : \Lambda(c) \in \bigcap_{c' \in sim(c)} val(c').$$

C_S states that, for every input configuration $c \in \mathcal{I}_{n-t}$, there exists a computable function $\Lambda(c)$ which retrieves a common admissible decision among all input configurations similar to c. The necessity of C_S follows from the canonical similarity result: in any infinite canonical execution, a common admissible value must be decided (Lemma 1).

Theorem 2. *Any solvable validity property satisfies C_S.*

Proof. By the means of contradiction, let there exist a validity property *val* such that (1) *val* does not satisfy C_S, and (2) *val* is solvable. Let $\mathcal{A}$ be any algorithm which solves the Byzantine consensus problem with *val*. As *val* does not satisfy C_S, there does not exist a computable function $\Lambda : \mathcal{I}_{n-t} \to \mathcal{V}_O$ such that, for every input configuration $c \in \mathcal{I}_{n-t}$, $\Lambda(c) \in \bigcap_{c' \in sim(c)} val(c')$.

Fix any input configuration $c \in \mathcal{I}_{n-t}$ for which $\Lambda(c)$ is not defined or not computable. Let $\mathcal{E}_c \in execs(\mathcal{A})$ be an infinite canonical execution such that (1) $\text{corresponding}(\mathcal{E}_c) = c$, (2) the system is synchronous from the very beginning (GST $= 0$), and (3) message delays are exactly δ. In other words, $\mathcal{E}_c$ unfolds in a "lock-step" manner. As $\mathcal{A}$ satisfies *Termination* and $\mathcal{E}_c$ is an infinite execution, some value $v_c \in \mathcal{V}_O$ is decided by a correct process in $\mathcal{E}_c$. By canonical similarity (Lemma 1), $v_c \in \bigcap_{c' \in sim(c)} val(c')$. Hence, $\Lambda(c)$ is defined ($\Lambda(c) = v_c$) and computable (due to the construction of $\mathcal{E}_c$). Thus, we reach a contradiction with the fact that $\Lambda(c)$ is not defined or not computable, which concludes the proof. □

Notice that, for proving the necessity of C_S (Theorem 2), we do not rely on the $n > 3t$ assumption. Hence, C_S is necessary for *all* solvable validity properties (irrespectively of the resiliency threshold). However, as proven in §4.1, C_S is not sufficient when $n \leq 3t$.[2] (Observe that any trivial validity property satisfies C_S.)

Similarity condition (example). Consider *Correct-Proposal Validity* [36, 73], a validity property which states that any decided value must have been proposed by a correct process. Consensus with *Correct-Proposal Validity* is also known as *strong* consensus [36, 73]. Strong consensus assumes $\mathcal{V}_I = \mathcal{V}_O$. It was shown that, in partial synchrony, strong consensus cannot be solved if $n \leq (|\mathcal{V}_I| + 1)t$ [36]. We now present an alternative proof. Namely, we show that *Correct-Proposal Validity* does not satisfy C_S if $n \leq (|\mathcal{V}_I| + 1)t$, which makes it unsolvable by Theorem 2.

Let $|\mathcal{V}_I| = |\mathcal{V}_O| = m$ and $n = (m+1)t$. We define an input configuration $base \in \mathcal{I}_{n-t}$ such that (1) $|\pi(base)| = n - t = mt$, and (2) every value $v \in \mathcal{V}_I$ is the proposal of exactly t processes in *base*. Therefore, all values are admissible for *base*. Next, for each value $v \in \mathcal{V}_I$, we design an input configuration $c_{\not\ni v}$ such that (1) v is not admissible for $c_{\not\ni v}$, and (2) $c_{\not\ni v} \sim base$:

(1) Let $c_{\not\ni v} \leftarrow base$.
(2) We remove from $c_{\not\ni v}$ all process-proposal pairs pp with $\text{proposal}(pp) = v$.
(3) For every process $P \notin \pi(base)$, we add (P, v') to $c_{\not\ni v}$, for any value $v' \neq v$.

Hence, *Correct-Proposal Validity* does not satisfy C_S when $n \leq (m+1)t$ as, for every $v \in \mathcal{V}_O$, there exists an input configuration $c_{\not\ni v} \sim base$ for which v is not admissible. Thus, *Correct-Proposal Validity* is unsolvable if $n \leq (m+1)t$ (by Theorem 2).

5 LOWER BOUND & GENERAL ALGORITHM

This section is devoted to the cost of solving consensus. Specifically, we first show that any non-trivial and solvable validity property requires $\Omega(t^2)$ messages to be exchanged (§5.1). Then, we present Universal, a general algorithm which, if $n > 3t$, solves consensus with any validity property which satisfies C_S (§5.2). Thus, Universal proves the sufficiency of C_S when $n > 3t$.

[2]For example, *Weak Validity* satisfies C_S, but it is unsolvable with $n \leq 3t$ [32].

5.1 Lower Bound on Message Complexity

In this subsection, we prove the following theorem:

Theorem 3. *If an algorithm solves consensus with a non-trivial validity property, the message complexity of the algorithm is $\Omega(t^2)$.*

Theorem 3 extends the seminal Dolev-Reischuk bound [30], proven only for consensus algorithms with *Strong Validity*, to all non-trivial variants of consensus. To prove Theorem 3, we intertwine the idea of the original proof [30] with the canonical similarity result (§3.4).

Proof of Theorem 3. In our proof, we show that any algorithm which solves the Byzantine consensus problem with a non-trivial validity property has a synchronous execution (GST = 0) in which correct processes send $\geq (\frac{t}{2})^2$ messages. Hence, throughout the entire subsection, we fix a non-trivial and solvable validity property *val*. Moreover, we fix $\mathcal{A}$, an algorithm which solves the Byzantine consensus problem with *val*. As *val* is a non-trivial validity property, $n > 3t$ (§4.1).

Next, we define a specific infinite execution $\mathcal{E}_{base} \in execs(\mathcal{A})$ in the following manner:

(1) GST = 0. That is, the system is synchronous throughout the entire execution.
(2) All processes are separated into two groups: (1) group A, with $|A| = n - \lceil \frac{t}{2} \rceil$, and (2) group B, with $|B| = \lceil \frac{t}{2} \rceil$.
(3) All processes in the group A are correct, whereas all processes in the group B are faulty.
(4) We fix any value $v^* \in \mathcal{V}_I$. For every correct process $P_A \in A$, the proposal of P_A in $\mathcal{E}_{base}$ is v^*.
(5) For every faulty process $P_B \in B$, P_B behaves correctly in $\mathcal{E}_{base}$ with its proposal being v^*, except that (1) P_B ignores the first $\lceil \frac{t}{2} \rceil$ messages received from other processes, and (2) P_B omits sending messages to other processes in B.

To prove Theorem 3, it suffices to show that the message complexity of $\mathcal{E}_{base}$ is $\geq (\lceil \frac{t}{2} \rceil)^2$. By contradiction, let the correct processes (processes in A) send less than $(\lceil \frac{t}{2} \rceil)^2$ messages in $\mathcal{E}_{base}$.

The first step of our proof shows that, given that correct processes send less than $(\lceil \frac{t}{2} \rceil)^2$ messages in $\mathcal{E}_{base}$, there must exist a process $Q \in B$ which can correctly decide some value $v_Q \in \mathcal{V}_O$ without receiving *any* message from any other process. We prove this claim using the pigeonhole principle.

Lemma 5. There exist a value $v_Q \in \mathcal{V}_O$ and a process $Q \in B$ such that Q has a correct behavior β_Q in which (1) Q decides v_Q, and (2) Q does not receive any message from any other process.

Proof. By assumption, correct processes (i.e., processes in the group A) send less than $(\lceil \frac{t}{2} \rceil)^2$ messages in $\mathcal{E}_{base}$. Therefore, due to the pigeonhole principle, there exists a process $Q \in B$ which receives less than $\lceil \frac{t}{2} \rceil$ messages (from other processes) in $\mathcal{E}_{base}$. Recall that Q behaves correctly in $\mathcal{E}_{base}$ with its proposal being $v^* \in \mathcal{V}_I$, except that (1) Q ignores the first $\lceil \frac{t}{2} \rceil$ messages received from other processes, and (2) Q does not send any messages to other processes in the group B. We denote by S_Q the set of processes, not including Q, which send messages to Q in $\mathcal{E}_{base}$; $|S_Q| < \lceil \frac{t}{2} \rceil$.

Next, we construct an infinite execution $\mathcal{E}'_{base}$. Execution $\mathcal{E}'_{base}$ is identical to $\mathcal{E}_{base}$, except that we introduce the following modifications:

(1) Processes in $(A \cup \{Q\}) \setminus S_Q$ are correct; other processes are faulty. That is, we make Q correct in $\mathcal{E}'_{base}$, and we make all processes in S_Q faulty in $\mathcal{E}'_{base}$.
(2) Processes in $B \setminus \{Q\}$ behave exactly as in $\mathcal{E}_{base}$. Moreover, processes in S_Q behave exactly as in $\mathcal{E}_{base}$, except that they *do not* send any message to Q.

Due to the construction of $\mathcal{E}'_{base}$, process Q does not receive any message (from any other process) in $\mathcal{E}'_{base}$. As Q is correct in $\mathcal{E}'_{base}$ and $\mathcal{A}$ satisfies *Termination*, Q decides some value $v_Q \in \mathcal{V}_O$ in $\mathcal{E}'_{base}$. Thus, Q indeed has a correct behavior β_Q in which it decides $v_Q \in \mathcal{V}_O$ without having received messages from other processes. □

In the second step of our proof, we show that there exists an execution in which (1) Q is faulty and silent, and (2) other processes decide some value $v \neq v_Q$.

Lemma 6. There exists an execution $\mathcal{E}_v$ such that (1) Q is faulty and silent in $\mathcal{E}_v$, and (2) a value $v \neq v_Q$ is decided by a correct process.

Proof. As *val* is a non-trivial validity property, there exists an input configuration $c_{\not\ni v_Q} \in \mathcal{I}$ such that $v_Q \notin val(c_{\not\ni v_Q})$; recall that v_Q is the value that Q can correctly decide without having received any message from any other process (Lemma 5). We consider two possible cases:

- Let $Q \notin \pi(c_{\not\ni v_Q})$. Thus, $\mathcal{E}_v$ is any infinite canonical execution which corresponds to $c_{\not\ni v_Q}$. As $v_Q \notin val(c_{\not\ni v_Q})$, the value decided in $\mathcal{E}_v$ must be different from v_Q (as $\mathcal{A}$ satisfies *val*).
- Let $Q \in \pi(c_{\not\ni v_Q})$. We construct an input configuration $c_{\not\ni Q} \in \mathcal{I}$ such that $Q \notin \pi(c_{\not\ni Q})$:
  (1) Let $c_{\not\ni Q} \leftarrow c_{\not\ni v_Q}$.
  (2) We remove $(Q, \cdot)$ from $c_{\not\ni Q}$. That is, we remove Q's process-proposal pair from $c_{\not\ni Q}$.
  (3) If $|\pi(c_{\not\ni v_Q})| = n - t$, we add $(Z, \text{any value})$ to $c_{\not\ni Q}$, where Z is any process such that $Z \notin \pi(c_{\not\ni v_Q})$; note that such a process Z exists as $t > 0$.

  Due to the construction of $c_{\not\ni Q}$, $c_{\not\ni Q} \sim c_{\not\ni v_Q}$. Indeed, (1) $\pi(c_{\not\ni Q}) \cap \pi(c_{\not\ni v_Q}) \neq \emptyset$ (as $n - t - 1 > 0$ when $n > 3t$ and $t > 0$), and (2) for every process $P \in \pi(c_{\not\ni Q}) \cap \pi(c_{\not\ni v_Q})$, the proposal of P is identical in $c_{\not\ni Q}$ and $c_{\not\ni v_Q}$ (by the step 1 of the construction).

  In this case, $\mathcal{E}_v$ is any infinite canonical execution such that $\text{corresponding}(\mathcal{E}_v) = c_{\not\ni Q}$. As $\mathcal{A}$ satisfies *Termination* and $\mathcal{E}_v$ is infinite, some value $v \in \mathcal{V}_O$ is decided by a correct process in $\mathcal{E}_v$. As $c_{\not\ni Q} \sim c_{\not\ni v_Q}$, $v \in val(c_{\not\ni v_Q})$ (by canonical similarity; Lemma 1). Finally, $v \neq v_Q$ as (1) $v \in val(c_{\not\ni v_Q})$, and (2) $v_Q \notin val(c_{\not\ni v_Q})$.

The lemma holds as its statement is true in both possible cases. □

As we have shown the existence of $\mathcal{E}_v$ (Lemma 6), we can "merge" $\mathcal{E}_v$ with the valid behavior β_Q in which Q decides v_Q without having received any message (Lemma 5). Hence, we can construct an execution in which $\mathcal{A}$ violates *Agreement*. Thus, correct processes must send at least $(\lceil \frac{t}{2} \rceil)^2 \in \Omega(t^2)$ messages in $\mathcal{E}_{base}$. The formal proof of the following lemma, from which the lower bound on message complexity (Theorem 3) follows directly, is given in Appendix B.

Lemma 7. The message complexity of $\mathcal{E}_{base}$ is at least $(\lceil \frac{t}{2} \rceil)^2$.

5.2 General Algorithm Universal: Similarity Condition is Sufficient if $n > 3t$

In this subsection, we prove that C_S is sufficient for a validity property to be solvable when $n > 3t$. That is, we prove the following theorem:

Theorem 4. *Let $n > 3t$, and let val be any validity property which satisfies C_S. Then, val is solvable. Moreover, assuming a threshold signature scheme, there exists an algorithm which solves Byzantine consensus with val, and has $O(n^2)$ message complexity.*

To prove Theorem 4, we present Universal, an algorithm which solves the Byzantine consensus problem with *any* validity property satisfying C_S, given that $n > 3t$. That is, Universal solves consensus with any solvable and non-trivial validity property. Notably, assuming a threshold signature scheme, Universal achieves $O(n^2)$ message complexity, making it optimal (when $t \in \Omega(n^2)$) according to our lower bound (§5.1).

To construct Universal, we rely on vector consensus [27, 31, 69, 76] (see §5.2.1), a problem which requires correct processes to agree on the proposals of $n - t$ processes. Specifically, when a correct process decides a vector *vec* of $n - t$ proposals (from vector consensus), it decides (from Universal) the common admissible value for all input configurations similar to *vec*, i.e., the process decides $\Lambda(vec)$. Note that the idea of solving consensus from vector consensus is not novel [14, 28, 68]. For some validity properties it is even natural, such as *Strong Validity* (choose the most common value) or *Weak Validity* (choose any value). However, thanks to the necessity of C_S (§4.2), *any* solvable consensus variant can reuse this simple algorithmic design.

In this subsection, we first recall vector consensus (§5.2.1). Then, we utilize vector consensus to construct Universal (§5.2.2). Throughout the entire subsection, $n > 3t$.

5.2.1 Vector Consensus. In essence, vector consensus allows each correct process to infer the proposals of $n - t$ (correct or faulty) processes. Formally, correct processes agree on input configurations (of vector consensus) with exactly $n - t$ process-proposal pairs: $\mathcal{V}_O = \mathcal{I}_{n-t}$.

Let us formally define *Vector Validity*, the validity property of vector consensus:

- *Vector Validity:* Let a correct process decide $vector \in \mathcal{V}_O$, which contains exactly $n - t$ process-proposal pairs, such that (1) (P, v) belongs to $vector$, for some process $P \in \Pi$ and some value $v \in \mathcal{V}_I$, and (2) P is a correct process. Then, P proposed v to vector consensus.

Intuitively, *Vector Validity* states that, if a correct process "concludes" that a value v was proposed by a correct process P, then P's proposal was indeed v.

We provide two implementations of vector consensus: (1) a non-authenticated implementation (without any cryptographic primitives), and (2) an authenticated implementation (with threshold signatures). Due to the lack of space, we give the pseudocode of the non-authenticated version in Appendix C.2. The pseudocode of an authenticated version is presented in Algorithm 1. This version relies on Quad, a Byzantine consensus algorithm recently introduced in [23]; we briefly discuss Quad below.

Quad. In essence, Quad is a partially-synchronous, "leader-based" Byzantine consensus algorithm, which achieves $O(n^2)$ message complexity. Internally, Quad relies on a threshold signature scheme. Formally, Quad is concerned with two sets: (1) $\mathcal{V}_{\text{Quad}}$, a set of values, and (2) $\mathcal{P}_{\text{Quad}}$, a set of proofs. In Quad, processes propose and decide value-proof pairs. There exists a function $\mathsf{verify} : \mathcal{V}_{\text{Quad}} \times \mathcal{P}_{\text{Quad}} \to \{\mathit{true}, \mathit{false}\}$. Importantly, $\mathcal{P}_{\text{Quad}}$ is not known a-priori: it is only assumed that, if a correct process proposes a pair $(v \in \mathcal{V}_{\text{Quad}}, \Sigma \in \mathcal{P}_{\text{Quad}})$, then $\mathsf{verify}(v, \Sigma) = \mathit{true}$. Quad guarantees the following: if a correct process decides a pair (v, Σ), then $\mathsf{verify}(v, \Sigma) = \mathit{true}$. In other words, correct processes decide only valid value-proof pairs. (See [23] for full details on Quad.)

In our authenticated implementation of vector consensus (Algorithm 1), we rely on a specific instance of Quad where (1) $\mathcal{V}_{\text{Quad}} = \mathcal{I}_{n-t}$ (processes propose to Quad input configurations of vector consensus), and (2) $\mathcal{P}_{\text{Quad}}$ is a set of $n - t$ proposal messages (sent by processes in vector consensus). Finally, given an input configuration $c \in \mathcal{V}_{\text{Quad}}$ and a set of messages $\Sigma \in \mathcal{P}_{\text{Quad}}$, $\mathsf{verify}(c, \Sigma) = \mathit{true}$ if and only if, for every process-proposal pair (P_j, v_j) which belongs to c, $\langle \text{proposal}, v_j \rangle_{\sigma_j} \in \Sigma$ (i.e., every process-proposal pair of c is accompanied by a properly signed proposal message).

Description of authenticated vector consensus (Algorithm 1). When a correct process P_i proposes a value $v \in \mathcal{V}_I$ to vector consensus (line 8), the process broadcasts a signed proposal message (line 9). Once P_i receives $n - t$ proposal messages (line 14), P_i constructs an input configuration $vector$ (line 15), and a proof Σ (line 16) from the received proposal messages. Moreover, P_i proposes $(vector, \Sigma)$ to Quad (line 17). Finally, when P_i decides a pair $(vector', \Sigma')$ from Quad (line 18), P_i decides $vector'$ from vector consensus (line 19).

The message complexity of Algorithm 1 is $O(n^2)$ as (1) processes only broadcast proposal messages, and (2) the message complexity of Quad is $O(n^2)$. We underline that the *communication complexity* of Algorithm 1, the number of bits sent by correct processes, is $O(n^3)$ as the communication complexity of Quad is $O(n^2 \cdot x) = O(n^3)$ (see [23]), where x is the size of a Quad proposal (in our case, $x \in \Theta(n)$). Due to space constraints, we delegate the full proof of the correctness and complexity of Algorithm 1 to Appendix C.1.

Algorithm 1 Authenticated Vector Consensus: Pseudocode (for process P_i)

```
1: Uses:
2:     Best-Effort Broadcast [19], instance beb          ▷ broadcast with no guarantees if the sender is faulty
3:     QUAD [23], instance quad
4: upon init:
5:     Integer $received\_proposals_i \leftarrow 0$          ▷ the number of received proposals
6:     Map(Process → $\mathcal{V}_I$) $proposals_i \leftarrow$ empty          ▷ received proposals
7:     Map(Process → Message) $messages_i \leftarrow$ empty          ▷ received PROPOSAL messages
8: upon propose($v \in \mathcal{V}_I$):
9:     invoke beb.broadcast($\langle$PROPOSAL, $v\rangle_{\sigma_i}$)          ▷ broadcast a signed proposal
10: upon reception of Message $m = \langle$PROPOSAL, $v_j \in \mathcal{V}_I\rangle_{\sigma_j}$ from process $P_j$ and $received\_proposals_i < n - t$:
11:     $received\_proposals_i \leftarrow received\_proposals_i + 1$
12:     $proposals_i[P_j] \leftarrow v_j$
13:     $messages_i[P_j] \leftarrow m$
14:     if $received\_proposals_i = n - t$:          ▷ received $n - t$ proposals; can propose to QUAD
15:         Input_Configuration vector ← input configuration constructed from $proposals_i$
16:         Proof $\Sigma \leftarrow$ set of messages containing all PROPOSAL messages from $messages_i$
17:         invoke quad.propose((vector, $\Sigma$))
18: upon quad.decide((Input_Configuration vector′, Proof $\Sigma'$)):
19:     trigger decide(vector′)
```

5.2.2 Universal. We construct Universal (Algorithm 2) directly from vector consensus (§5.2.1). When a correct process P_i proposes to Universal (line 3), the proposal is forwarded to vector consensus (line 4). Once P_i decides an input configuration c from vector consensus (line 5), P_i decides $\Lambda(c)$ (line 6).

Note that our implementation of Universal (Algorithm 2) is independent of the actual implementation of vector consensus. Thus, by employing our authenticated implementation of vector consensus (Algorithm 1), we obtain a general consensus algorithm with $O(n^2)$ message complexity. On the other hand, by employing a non-authenticated implementation of vector consensus (see Appendix C.2), we obtain a non-authenticated version of Universal, which implies that any validity property which satisfies C_S is solvable even in a non-authenticated setting (if $n > 3t$).

Algorithm 2 Universal: Pseudocode for process P_i

```
1: Uses:
2:     Vector Consensus, instance vec_cons
3: upon propose($v \in \mathcal{V}_I$):
4:     invoke vec_cons.propose($v$)
5: upon vec_cons.decide(Input_Configuration $c$):
6:     trigger decide($\Lambda(c)$)
```

Finally, we prove that Universal (Algorithm 2) is a general Byzantine consensus algorithm.

Theorem 5. *Let val be any validity property which satisfies C_S, and let $n > 3t$. Universal solves the Byzantine consensus problem with val. Moreover, if Universal employs Algorithm 1 as its vector consensus building block, the message complexity of Universal is $O(n^2)$.*

Proof. *Termination* and *Agreement* of Universal follow from *Termination* and *Agreement* of vector consensus, respectively. Moreover, the message complexity of Universal is identical to the message complexity of its vector consensus building block.

Finally, we prove that Universal satisfies *val*. Consider any execution $\mathcal{E}$ of Universal such that $\text{corresponding}(\mathcal{E}) = c^*$, for some input configuration $c^* \in \mathcal{I}$. Let $c \in \mathcal{I}_{n-t}$ be the input configuration correct processes decide from vector consensus in $\mathcal{E}$ (line 5). As vector consensus satisfies *Vector Validity*, we have that, for every process $P \in \pi(c^*) \cap \pi(c)$, P's proposals in c^* and c are identical. Hence, $c \sim c^*$. Therefore, $\Lambda(c) \in val(c^*)$ (by the definition of the Λ function). Thus, *val* is satisfied by Universal. □

As Universal (Algorithm 2) solves the Byzantine consensus problem with any validity property which satisfies C_S (Theorem 5) if $n > 3t$, C_S is sufficient for solvable validity properties when $n > 3t$. Lastly, as Universal relies on vector consensus, we conclude that *Vector Validity* is a *strongest* validity property. That is, a solution to any variant of the consensus problem can be obtained from vector consensus.

A note on the communication complexity of vector consensus. While the version of Universal which employs Algorithm 1 (as its vector consensus building block) has optimal message complexity, its communication complexity is $O(n^3)$. This presents a linear gap to the lower bound for communication complexity (also $\Omega(n^2)$, implied by Theorem 3), and to known optimal solutions for some validity properties (e.g., *Strong Validity*, proven to be $\Theta(n^2)$ [23]). At first glance, this seems like an issue inherent to vector consensus: the decided vectors are linear in size, suggesting that the linear gap could be inevitable. However, this is not the case. In Appendix C.3, we give a vector consensus algorithm with $O(n^2 \log n)$ communication complexity, albeit with exponential latency.[3] Is it possible to construct vector consensus with subcubic communication and polynomial latency? This is an important open question, as positive answers would lead to (practical) performance improvements of all consensus variants.

6 CONCLUDING REMARKS

This paper studies the validity property of partially synchronous Byzantine consensus. Namely, we mathematically formalize validity properties, and give necessary and sufficient conditions for a validity property to be solvable (i.e., for the existence of an algorithm which solves a consensus problem defined with that validity property, in addition to *Agreement* and *Termination*). Moreover, we prove a quadratic lower bound on message complexity for all non-trivial (and solvable) validity properties. Previously, this bound was mainly known for *Strong Validity*. Lastly, we introduce Universal, a general algorithm for consensus with any solvable (and non-trivial) validity property; Universal achieves $O(n^2)$ message complexity, showing that the aforementioned lower bound is tight (with $t \in \Omega(n)$).

We conjecture that our necessary and sufficient conditions for consensus solvability can easily be adapted to a synchronous environment. However, our proof technique for the lower bound on message complexity cannot be reused as such: this is because silent processes can be conclusively detected in synchrony. Thus, we believe that the lower bound on message complexity for synchronous consensus is one of the most important open questions. Another interesting question is whether our lower bound holds for randomized consensus. In [3], it is proven that randomized consensus with *Strong Validity* has $\Omega(t^2)$ expected message complexity. Can this bound be extended to all non-trivial validity properties?

Finally, we restate the question posed at the end of §5.2. Is it possible to solve vector consensus with $o(n^3)$ exchanged bits and polynomial latency? Recall that, due to the design of Universal (§5.2), any (non-trivial) consensus variant can be solved using vector consensus. Therefore, an

[3]Both our authenticated (Algorithm 1) and our non-authenticated (see Appendix C.2) variants of vector consensus have linear latency, which implies linear latency of Universal when employing any of these two algorithms.

upper bound on the complexity of vector consensus is an upper bound on the complexity of any consensus variant. Hence, lowering the $O(n^3)$ communication complexity of vector consensus (while preserving "reasonable" polynomial latency) constitutes an important future research direction.

REFERENCES

[1] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. Fault-Scalable Byzantine Fault-Tolerant Services. *ACM SIGOPS Operating Systems Review 39*, 5 (2005), 59–74.

[2] Abraham, I., and Cachin, C. What about Validity? https://decentralizedthoughts.github.io/2022-12-12-what-about-validity/.

[3] Abraham, I., Chan, T. H., Dolev, D., Nayak, K., Pass, R., Ren, L., and Shi, E. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019* (2019), P. Robinson and F. Ellen, Eds., ACM, pp. 317–326.

[4] Abraham, I., Devadas, S., Nayak, K., and Ren, L. Brief Announcement: Practical Synchronous Byzantine Consensus. In *31st International Symposium on Distributed Computing (DISC 2017)* (2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[5] Abraham, I., Malkhi, D., Nayak, K., Ren, L., and Spiegelman, A. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. *arXiv preprint arXiv:1612.02916* (2016).

[6] Abraham, I., Malkhi, D., Nayak, K., Ren, L., and Spiegelman, A. Solidus: An Incentive-compatible Cryptocurrency Based on Permissionless Byzantine Consensus. *CoRR, abs/1612.02916* (2016).

[7] Abraham, I., Malkhi, D., and Spiegelman, A. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 337–346.

[8] Adya, A., Bolosky, W., Castro, M., Cermak, G., Chaiken, R., Douceur, J., Howell, J., Lorch, J., Theimer, M., and Wattenhofer, R. {FARSITE}: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)* (2002).

[9] Amir, Y., Danilov, C., Kirsch, J., Lane, J., Dolev, D., Nita-Rotaru, C., Olsen, J., and Zage, D. Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. In *International Conference on Dependable Systems and Networks (DSN'06)* (2006), IEEE, pp. 105–114.

[10] Aspnes, J. Randomized Protocols for Asynchronous Consensus. *Distributed Computing 16*, 2 (2003), 165–175.

[11] Ben-Or, M. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983* (1983), R. L. Probert, N. A. Lynch, and N. Santoro, Eds., ACM, pp. 27–30.

[12] Ben-Or, M., and El-Yaniv, R. Resilient-optimal interactive consistency in constant time. *Distributed Computing 16*, 4 (2003), 249–262.

[13] Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 351–371.

[14] Ben-Or, M., Kelmer, B., and Rabin, T. Asynchronous Secure Computations with Optimal Resilience (Extended Abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994* (1994), J. H. Anderson, D. Peleg, and E. Borowsky, Eds., ACM, pp. 183–192.

[15] Biran, O., Moran, S., and Zaks, S. A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *Journal of algorithms 11*, 3 (1990), 420–440.

[16] Bracha, G. Asynchronous Byzantine Agreement Protocols. *Inf. Comput. 75*, 2 (1987), 130–143.

[17] Buchman, E. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.

[18] Buchman, E., Kwon, J., and Milosevic, Z. The latest gossip on BFT consensus. Tech. Rep. 1807.04938, arXiv, 2019.

[19] Cachin, C., Guerraoui, R., and Rodrigues, L. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[20] Cachin, C., Kursawe, K., Petzold, F., and Shoup, V. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings* (2001), J. Kilian, Ed., vol. 2139 of *Lecture Notes in Computer Science*, Springer, pp. 524–541.

[21] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems 20*, 4 (2002).

[22] Chor, B., and Moscovici, L. Solvability in Asynchronous Environments (Extended Abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989* (1989), IEEE Computer Society, pp. 422–427.

[23] Civit, P., Dzulfikar, M. A., Gilbert, S., Gramoli, V., Guerraoui, R., Komatovic, J., and Vidigueira, M. Byzantine

Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony! In *36th International Symposium on Distributed Computing (DISC 2022)* (Dagstuhl, Germany, 2022), C. Scheideler, Ed., vol. 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 14:1–14:21.

[24] Civit, P., Gilbert, S., and Gramoli, V. Polygraph: Accountable Byzantine Agreement. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21)* (Jul 2021).

[25] Civit, P., Gilbert, S., Gramoli, V., Guerraoui, R., Komatovic, J., Milosevic, Z., and Seredinschi, A. Crime and Punishment in Distributed Byzantine Decision Tasks. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022* (2022), IEEE, pp. 34–44.

[26] Correia, M. From Byzantine Consensus to Blockchain Consensus. In *Essentials of Blockchain Technology*. Chapman and Hall/CRC, 2019, pp. 41–80.

[27] Correia, M., Neves, N. F., and Veríssimo, P. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *The Computer Journal 49*, 1 (2006), 82–96.

[28] Crain, T., Gramoli, V., Larrea, M., and Raynal, M. DBFT: Efficient Leaderless Byzantine Consensus and its Applications to Blockchains. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (NCA'18)* (2018), IEEE.

[29] Das, S., Xiang, Z., and Ren, L. Asynchronous Data Dissemination and its Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 2705–2721.

[30] Dolev, D., and Reischuk, R. Bounds on Information Exchange for Byzantine Agreement. *Journal of the ACM (JACM) 32*, 1 (1985), 191–204.

[31] Doudou, A., and Schiper, A. Muteness Detectors for Consensus with Byzantine Processes. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998* (1998), B. A. Coan and Y. Afek, Eds., ACM, p. 315.

[32] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the Presence of Partial Synchrony. *Journal of the Association for Computing Machinery, Vol. 35, No. 2, pp.288-323* (1988).

[33] Ezhilchelvan, P., Mostefaoui, A., and Raynal, M. Randomized Multivalued Consensus. In *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001* (2001), IEEE, pp. 195–200.

[34] Fischer, M. J., and Lynch, N. A. A Lower Bound for the Time to Assure Interactive Consistency. Tech. rep., GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1981.

[35] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[36] Fitzi, M., and Garay, J. A. Efficient Player-Optimal Protocols for Strong and Differential Consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (2003), pp. 211–220.

[37] Galil, Z., Haber, S., and Yung, M. Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model. In *Conference on the Theory and Application of Cryptographic Techniques* (1987), Springer, pp. 135–155.

[38] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 51–68.

[39] Gilbert, S., Lynch, N. A., and Shvartsman, A. A. Rambo: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks. *Distributed Computing 23*, 4 (2010), 225–272.

[40] Guerraoui, R., and Schiper, A. The Generic Consensus Service. *IEEE Trans. Software Eng. 27*, 1 (2001), 29–41.

[41] Hagit, A., and Zvi, A. Wait-Free n-Set Consensus When Inputs Are Restricted. In *International Symposium on Distributed Computing* (2002), Springer, pp. 326–338.

[42] Herlihy, M., Kozlov, D. N., and Rajsbaum, S. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[43] Herlihy, M., and Shavit, N. The Asynchronous Computability Theorem for t-Resilient Tasks. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA* (1993), S. R. Kosaraju, D. S. Johnson, and A. Aggarwal, Eds., ACM, pp. 111–120.

[44] Herlihy, M., and Shavit, N. The Topological Structure of Asynchronous Computability. *J. ACM 46*, 6 (1999), 858–923.

[45] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. Byzantine Fault Detectors for Solving Consensus. *British Computer Society* (2003).

[46] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (2007), pp. 45–58.

[47] Kotla, R., and Dahlin, M. High Throughput Byzantine Fault Tolerance. In *International Conference on Dependable Systems and Networks, 2004* (2004), IEEE, pp. 575–584.

[48] Kowalski, D. R., and Mostéfaoui, A. Synchronous Byzantine Agreement with Nearly a Cubic Number of Communication Bits. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing* (2013), pp. 84–91.

[49] Lamport, L. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121,*

December 2001) (2001), 51–58.

[50] Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (1982), 382–401.

[51] Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. In *Concurrency: the works of leslie lamport.* 2019, pp. 203–226.

[52] Lewis-Pye, A. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. *arXiv preprint arXiv:2201.01107* (2022).

[53] Libert, B., Joye, M., and Yung, M. Born and Raised Distributively: Fully Distributed Non-Interactive Adaptively-Secure Threshold Signatures with Short Shares. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing* (2014), pp. 303–312.

[54] Lu, Y., Lu, Z., Tang, Q., and Wang, G. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020* (2020), Y. Emek and C. Cachin, Eds., ACM, pp. 129–138.

[55] Luu, L., Narayanan, V., Baweja, K., Zheng, C., Gilbert, S., and Saxena, P. SCP: A Computationally-Scalable Byzantine Consensus Protocol For Blockchains. *Cryptology ePrint Archive* (2015).

[56] Lynch, N. A. *Distributed Algorithms.* Elsevier, 1996.

[57] Malkhi, D., Nayak, K., and Ren, L. Flexible Byzantine Fault Tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security* (2019), pp. 1041–1053.

[58] Maneas, S., Chondros, N., Diamantopoulos, P., Patsonakis, C., and Roussopoulos, M. On achieving interactive consistency in real-world distributed systems. *Journal of Parallel and Distributed Computing 147* (2021), 220–235.

[59] Melnyk, D., and Wattenhofer, R. Byzantine Agreement with Interval Validity. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)* (2018), IEEE, pp. 251–260.

[60] Mendes, H., Tasson, C., and Herlihy, M. Distributed Computability in Byzantine Asynchronous Systems. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014* (2014), D. B. Shmoys, Ed., ACM, pp. 704–713.

[61] Momose, A., and Ren, L. Multi-Threshold Byzantine Fault Tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 1686–1699.

[62] Momose, A., and Ren, L. Optimal Communication Complexity of Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)* (2021), S. Gilbert, Ed., vol. 209 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 32:1–32:16.

[63] Moran, S., and Wolfstahl, Y. Extended Impossibility Results for Asynchronous Complete Networks. *Information Processing Letters 26*, 3 (1987), 145–151.

[64] Mostefaoui, A., Rajsbaum, S., and Raynal, M. Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Journal of the ACM (JACM) 50*, 6 (2003), 922–954.

[65] Mostefaoui, A., Rajsbaum, S., and Raynal, M. Using Conditions to Expedite Consensus in Synchronous Distributed Systems. In *International Symposium on Distributed Computing* (2003), Springer, pp. 249–263.

[66] Mostefaoui, A., Rajsbaum, S., Raynal, M., and Roy, M. A Hierarchy of Conditions for Consensus Solvability. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing* (2001), pp. 151–160.

[67] Mostéfaoui, A., Rajsbaum, S., Raynal, M., and Roy, M. Condition-Based Protocols for Set Agreement Problems. In *International Symposium on Distributed Computing* (2002), Springer, pp. 48–62.

[68] Mostefaoui, A., Raynal, M., and Tronel, F. From Binary Consensus to Multivalued Consensus in asynchronous message-passing systems. *Information Processing Letters 73*, 5-6 (2000), 207–212.

[69] Neves, N. F., Correia, M., and Verissimo, P. Solving Vector Consensus with a Wormhole. *IEEE Transactions on Parallel and Distributed Systems 16*, 12 (2005), 1120–1131.

[70] Raynal, M. Consensus in Synchronous Systems: A Concise Guided Tour. In *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.* (2002), IEEE, pp. 221–228.

[71] Saraph, V., Herlihy, M., and Gafni, E. An Algorithmic Approach to the Asynchronous Computability Theorem. *J. Appl. Comput. Topol. 1*, 3-4 (2018), 451–474.

[72] Schmid, U., and Weiss, B. Synchronous Byzantine Agreement under Hybrid Process and Link Failures.

[73] Siu, H.-S., Chin, Y.-H., and Yang, W.-P. Reaching strong consensus in the presence of mixed failure types. *Information Sciences 108*, 1-4 (1998), 157–180.

[74] Stolz, D., and Wattenhofer, R. Byzantine Agreement with Median Validity. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)* (2016), vol. 46, Schloss Dagstuhl–Leibniz-Zentrum für Informatik GmbH, p. 22.

[75] Taubenfeld, G., Katz, S., and Moran, S. Initial Failures in Distributed Computations. *International Journal of Parallel Programming 18*, 4 (1989), 255–276.

[76] Vaidya, N. H., and Garg, V. K. Byzantine Vector Consensus in Complete Graphs. In *Proceedings of the 2013 ACM*

symposium on Principles of distributed computing (2013), pp. 65–73.

[77] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Verissimo, P. Efficient Byzantine Fault-Tolerance. *IEEE Transactions on Computers 62*, 1 (2011), 16–30.

[78] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 347–356.

[79] Zibin, Y. Condition-Based Consensus in Synchronous Systems. In *International Symposium on Distributed Computing* (2003), Springer, pp. 239–248.

A TRIVIALITY OF SOLVABLE VALIDITY PROPERTIES IF $n \leq 3t$: FORMAL PROOF

In this section, we give formal proofs of the intermediate results from §4.1. First, we formally prove that only v_{base} can be decided in any canonical execution which corresponds to any input configuration compatible with *base*.

Lemma 2 (restated)**.** Let $c \in \mathcal{I}$ be any input configuration such that $c \diamond base$. Let $\mathcal{E}_c \in execs(\mathcal{A})$ be any canonical execution such that $\mathsf{corresponding}(\mathcal{E}_c) = c$. If a value $v_c \in \mathcal{V}_O$ is decided by a correct process in $\mathcal{E}_c$, then $v_c = v_{base}$.

Proof. By contradiction, suppose that some value $v_c \neq v_{base}$ is decided by a correct process in $\mathcal{E}_c$. Since $c \diamond base$, there exists a correct process $Q \in \pi(c) \setminus \pi(base)$. Let $\mathcal{E}_c^Q$ be an infinite continuation of $\mathcal{E}_c$ in which Q decides; note that $\mathcal{E}_c^Q$ exists as $\mathcal{A}$ satisfies *Termination*. The following holds for process Q: (1) Q decides v_c in $\mathcal{E}_c^Q$ (as $\mathcal{A}$ satisfies *Agreement*), and (2) process Q is silent in $\mathcal{E}_{base}$. Let t_Q denote the time at which Q decides in $\mathcal{E}_c^Q$. Similarly, there exists a process $P \in \pi(base) \setminus \pi(c)$; observe that (1) process P decides v_{base} in $\mathcal{E}_{base}$, and (2) process P is silent in $\mathcal{E}_c^Q$. Let t_P denote the time at which P decides in $\mathcal{E}_{base}$.

We now construct an execution $\mathcal{E} \in execs(\mathcal{A})$ by "merging" $\mathcal{E}_{base}$ and $\mathcal{E}_c^Q$:

(1) Processes in $\pi(c) \cap \pi(base)$ behave towards processes in $\pi(base) \setminus \pi(c)$ as in $\mathcal{E}_{base}$, and towards processes in $\pi(c) \setminus \pi(base)$ as in $\mathcal{E}_c^Q$.
(2) Communication between (1) processes in $\pi(base) \setminus \pi(c)$ and (2) processes in $\pi(c) \setminus \pi(base)$ is delayed until after $\max(t_P, t_Q)$.
(3) We set GST to after $\max(t_P, t_Q)$.

The following holds for $\mathcal{E}$:

- Processes in $\pi(base) \ominus \pi(c)$ (symmetric difference) are correct in $\mathcal{E}$.
- Only processes in $\pi(base) \cap \pi(c)$ are faulty in $\mathcal{E}$. Recall that $|\pi(base) \cap \pi(c)| \leq t$ as $base \diamond c$.
- Process Q, which is correct in $\mathcal{E}$, cannot distinguish $\mathcal{E}$ from $\mathcal{E}_c^Q$ until time $\max(t_P, t_Q)$. Hence, process Q decides v_c in $\mathcal{E}$.
- Process P, which is correct in $\mathcal{E}$, cannot distinguish $\mathcal{E}$ from $\mathcal{E}_{base}$ until time $\max(t_P, t_Q)$. Hence, process P decides $v_{base} \neq v_c$ in $\mathcal{E}$.

Therefore, we reach a contradiction with the fact that $\mathcal{A}$ satisfies *Agreement*. Thus, $v_c = v_{base}$. □

Next, we formally prove that, for every input configuration $c_n \in \mathcal{I}_n$, there exists an execution $\mathcal{E}_n$ such that (1) $\mathcal{E}_n$ corresponds to c_n, and (2) v_{base} is decided in $\mathcal{E}_n$.

Lemma 3 (restated)**.** For every input configuration $c_n \in \mathcal{I}_n$, there exists an execution $\mathcal{E}_n \in execs(\mathcal{A})$ such that (1) $\mathsf{corresponding}(\mathcal{E}_n) = c_n$, and (2) v_{base} is decided in $\mathcal{E}_n$.

Proof. Fix any input configuration $c_n \in \mathcal{I}_n$. We construct an input configuration $c_{n-t} \in \mathcal{I}_{n-t}$:

(1) For every process $P_i \notin \pi(base)$, we include a process-proposal pair (P_i, v) in c_{n-t} such that $v = \mathsf{proposal}(c_n[i])$. Note that there are t such processes as $|\pi(base)| = n - t$.
(2) We include $n - 2t$ process-proposal pairs (P_i, v) in c_{n-t} such that (1) $P_i \in \pi(base)$, and (2) $v = \mathsf{proposal}(c_n[i])$. That is, we "complete" c_{n-t} (constructed in the step 1) with $n - 2t$ process-proposal pairs such that the process is "borrowed" from *base*, and its proposal is "borrowed" from c_n.

Observe that $c_{n-t} \diamond base$ as (1) $|\pi(c_{n-t}) \cap base| \leq t$ (because $n - 2t \leq t$ when $n \leq 3t$), (2) there exists a process $P \in \pi(base) \setminus \pi(c_{n-t})$ (because, when constructing c_{n-t}, we excluded $t > 0$ processes from *base*; step 1), and (3) there exists a process $Q \in \pi(c_{n-t}) \setminus \pi(base)$ (because we included $t > 0$ processes in c_{n-t} which are not in *base*; step 1).

Let $\mathcal{E}_{n-t} \in execs(\mathcal{A})$ denote any infinite canonical execution such that $\mathsf{corresponding}(\mathcal{E}_{n-t}) = c_{n-t}$. As $\mathcal{A}$ satisfies *Termination*, some value is decided by correct processes in $\mathcal{E}_{n-t}$; due to Lemma 2, that value is v_{base}. Finally, we are able to construct an infinite execution $\mathcal{E}_n \in execs(\mathcal{A})$ such that (1) $\mathsf{corresponding}(\mathcal{E}_n) = c_n$, and (2) v_{base} is decided in $\mathcal{E}_n$:

(1) All processes are correct in $\mathcal{E}_n$.
(2) Until some correct process $P \in \pi(c_{n-t})$ decides v_{base}, $\mathcal{E}_n$ is identical to $\mathcal{E}_{n-t}$.
(3) Afterwards, every process $Q \notin \pi(c_{n-t})$ "wakes up" with the proposal specified in c_n.
(4) GST is set to after all processes have taken a computational step.

Therefore, v_{base} is indeed decided in $\mathcal{E}_n$ and $\mathsf{corresponding}(\mathcal{E}_n) = c_n$, which concludes the proof. □

B LOWER BOUND ON MESSAGE COMPLEXITY: FORMAL PROOF

In this section, we give a formal proof of Lemma 7. Recall that we have fixed an algorithm $\mathcal{A}$ which solves the Byzantine consensus problem with a non-trivial validity property *val*.

Lemma 7 (restated)**.** The message complexity of $\mathcal{E}_{base}$ is at least $(\lceil \frac{t}{2} \rceil)^2$.

Proof. By Lemma 5, there exists a behavior β_Q of process Q in which Q decides a value v_Q without having received any message (from any other process). Let t_Q denote the time at which Q decides v_Q in β_Q. Moreover, there exists an execution $\mathcal{E}_v$ in which (1) Q is faulty and silent, and (2) correct processes decide a value $v \neq v_Q$ (by Lemma 6). Let t_v denote the time at which a correct process decides $v \neq v_Q$ in $\mathcal{E}_v$.

We now construct an execution $\mathcal{E}$ in the following way:

(1) Processes in $Corr_{\mathcal{A}}(\mathcal{E}_v) \cup \{Q\}$ are correct in $\mathcal{E}$. All other processes are faulty.
(2) All messages from and to Q are delayed until after $\max(t_Q, t_v)$.
(3) Process Q exhibits the behavior β_Q.
(4) Until $\max(t_Q, t_v)$, no process in $Corr_{\mathcal{A}}(\mathcal{E}_v)$ can distinguish $\mathcal{E}$ from $\mathcal{E}_v$.
(5) GST is set to after $\max(t_Q, t_v)$.

As no process in $Corr_{\mathcal{A}}(\mathcal{E}_v)$ can distinguish $\mathcal{E}$ from $\mathcal{E}_v$ until $\max(t_Q, t_v)$, $v \neq v_Q$ is decided by a correct process in $\mathcal{E}$. Moreover, Q decides v_Q in $\mathcal{E}$ (step 3 of the construction). Thus, *Agreement* is violated in $\mathcal{E}$, which contradicts the fact that $\mathcal{A}$ satisfies *Agreement*. Hence, the starting assumption is not correct: in $\mathcal{E}_{base}$, correct processes send (at least) $(\lceil \frac{t}{2} \rceil)^2$ messages. □

C VECTOR CONSENUS: FORMAL PROOFS & OMITTED ALGORITHMS

In Appendix C.1, we prove the correctness and complexity of our authenticated implementation of vector consensus (Algorithm 1). We dedicate Appendix C.2 to a non-authenticated implementation of vector consensus. Finally, in Appendix C.3, we give an implementation of vector consensus with $O(n^2 \log n)$ communication complexity. Throughout the entire section, we assume that $n > 3t$.

C.1 Authenticated Implementation (Algorithm 1): Formal Proofs

In this subsection, we prove the correctness and complexity of our authenticated implementation of vector consensus. We start with the correctness.

Theorem 6. *Algorithm 1 is correct.*

Proof. *Agreement* follows directly from the fact that Quad satisfies *Agreement*. *Termination* follows from (1) *Termination* of Quad, and (2) the fact that, eventually, all correct processes receive $n - t$ proposal messages (as there are at least $n - t$ correct processes).

We now prove that Algorithm 1 satisfies *Vector Validity*. Let a correct process P decide $vector' \in \mathcal{I}_{n-t}$ from vector consensus (line 19). Hence, P has decided $(vector', \Sigma')$ from Quad, where (1) Σ' is some proof, and (2) $\mathsf{verify}(vector', \Sigma') = true$ (due to the specification of Quad). Furthermore, if there exists a process-proposal pair $(P, v \in \mathcal{V}_I)$ in $vector'$, where P is a correct process, a properly signed PROPOSAL message belongs to Σ'. As correct processes send PROPOSAL messages only for their proposals (line 9), v was indeed proposed by P. Thus, the theorem. □

Finally, we prove the complexity of Algorithm 1.

THEOREM 7. *The message complexity of Algorithm 1 is $O(n^2)$.*

PROOF. The message complexity of the specific instance of QUAD utilized in Algorithm 1 is $O(n^2)$. Additionally, correct processes exchange $O(n^2)$ PROPOSAL messages. Thus, the message complexity is $O(n^2) + O(n^2) = O(n^2)$. □

C.2 Non-Authenticated Implementation: Pseudocode & Formal Proofs

We now present a non-authenticated implementation (Algorithm 3) of vector consensus. The design of Algorithm 3 follows the reduction from the binary consensus to the multivalue consensus (e.g., [28]). Namely, we use the following two building blocks in Algorithm 3:

(1) Byzantine Reliable Broadcast [16, 19]: This primitive allows processes to disseminate information in a reliable manner. Formally, the Byzantine reliable broadcast exposes the following interface: (1) **request** broadcast(m), and (2) **indication** deliver(m'). The primitive satisfies the following properties:
- *Validity:* If a correct process P broadcasts a message m, P eventually delivers m.
- *Consistency:* No two correct processes deliver different messages.
- *Integrity:* Every correct process delivers at most one message. Moreover, if a correct process delivers a message m from a process P and P is correct, then P broadcast m.
- *Totality:* If a correct process delivers a message, every correct process delivers a message.

In Algorithm 3, we use a non-authenticated implementation [16] of the Byzantine Reliable Broadcast primitive.

(2) Binary DBFT [28], a non-authenticated algorithm which solves the Byzantine consensus problem with *Strong Validity*.

Let us briefly explain how Algorithm 3 works; we focus on a correct process P_i. First, P_i reliably broadcasts its proposal (line 11). Once P_i delivers a proposal of some process P_j (line 12), P_i proposes 1 to the corresponding DBFT instance (line 17). Eventually, $n - t$ DBFT instances decide 1 (line 18). Once that happens, P_i proposes 0 to all DBFT instance to which P_i has not proposed (line 22). When all DBFT instances have decided (line 23), P_i decides an input configuration associated with the first $n - t$ processes whose DBFT instances decided 1 (constructed at line 24).

THEOREM 8. *Algorithm 3 is correct.*

PROOF. We start by proving *Termination* of Algorithm 3. Eventually, at least $n - t$ DBFT instances decide 1 due to the fact that (1) no correct process proposes 0 to any DBFT instance unless $n - t$ DBFT instances have decided 1 (line 18), and (2) all correct processes eventually propose 1 to the DBFT instances which correspond to the correct processes (unless $n - t$ DBFT instances have already decided 1). When $n - t$ DBFT instances decide 1 (line 18), each correct process proposes to all instances to which it has not yet proposed (line 22). Hence, eventually all DBFT instances decide, and (at least) $n - t$ DBFT instances decide 1. Therefore, the rule at line 23 eventually activates at every correct process, which implies that every correct process eventually decides (line 25).

Next, we prove *Vector Validity*. If a correct process P decides an input configuration with a process-proposal pair (Q, v), P has delivered a PROPOSAL message from Q. If Q is correct, due to integrity of the reliable broadcast primitive, Q's proposal was indeed v.

Finally, *Agreement* follows from (1) *Agreement* of DBFT, and (2) consistency of the reliable broadcast primitive. Therefore, Algorithm 3 is correct. □

Algorithm 3 Non-Authenticated Vector Consensus: Pseudocode (for process P_i)

```
1: Uses:
2:     Non-Authenticated Byzantine Reliable Broadcast [16], instance $brb$
3:     Binary DBFT [28], instances $dbft[1], ..., dbft[n]$    ▷ one instance of the binary DBFT protocol per process
4: upon init:
5:     Map(Process → $\mathcal{V}_I$) $proposals_i$ ← empty    ▷ received proposals
6:     Map(Process → Message) $messages_i$ ← empty    ▷ received PROPOSAL messages
7:     Boolean $dbft\_proposing_i = true$    ▷ is $P_i$ still proposing to the DBFT instances
8:     Map(Process → Boolean) $dbft\_proposed_i$ ← {$false$, for every Process}
9:     Integer $dbft\_decisions_i \leftarrow 0$    ▷ the number of the DBFT instances which have decided
10: upon propose($v \in \mathcal{V}_I$):
11:     invoke $brb$.broadcast(⟨PROPOSAL, $v$⟩)    ▷ broadcast a proposal
12: upon reception of Message $m$ = ⟨PROPOSAL, $v_j \in \mathcal{V}_I$⟩ from process $P_j$:
13:     $proposals_i[P_j] \leftarrow v_j$
14:     $messages_i[P_j] \leftarrow m$
15:     if $dbft\_proposing_i = true$:
16:         $dbft\_proposed_i[P_j] \leftarrow true$
17:         invoke $dbft[j]$.propose(1)
18: upon $n - t$ DBFT instances have decided 1 (for the first time):
19:     $dbft\_proposing_i \leftarrow false$
20:     for every Process $P_j$ such that $dbft\_proposed_i[P_j] = false$:
21:         $dbft\_proposed_i[P_j] \leftarrow true$
22:         invoke $dbft[j]$.propose(0)
23: upon all DBFT instances decided, and, for the first $n - t$ processes $P_j$ such that $dbft[j]$ decided 1, $proposals_i[P_j] \neq \bot$:
24:     Input_Configuration $vector$ ← input configuration with $n - t$ process-proposal pairs corresponding to the first
        $n - t$ DBFT instances which decided 1
25:     trigger decide($vector$)
```

The main downside of Algorithm 3 is that its message complexity is $O(n^4)$. Therefore, non-authenticated version of UNIVERSAL has $O(n^4)$ message complexity, which is not optimal according to our lower bound (§5.1).

C.3 Implementation with $O(n^2 \log n)$ Communication: Pseudocode & Formal Proofs

In this subsection, we give an implementation of vector consensus with $O(n^2 \log n)$ communication complexity, which comes within a logarithmic factor of the lower bound (§5.1). This implementation represents a near-linear communication improvement over Algorithm 1 (§5.2), which achieves $O(n^3)$ communication complexity. We note that the following solution is highly impractical due to its exponential latency (worst-case $O(n^t)$, requiring idealized cryptographic primitives). However, our solution does represent a step towards closing the existing gap in the communication complexity of non-trivial and solvable validity properties.

C.3.1 Vector Dissemination. First, we formally define the *vector dissemination* problem, which plays the crucial role in our vector consensus algorithm with improved communication complexity. In this problem, each correct process *disseminates* a vector of $n - t$ values, and all correct processes eventually obtain (1) a hash of some disseminated vector, and (2) a storage proof. For every hash value H and every storage proof sp, we define valid_SP$(H, sp) \in \{true, false\}$. Formally, the vector dissemination problem exposes the following interface:

- **request** disseminate(Vector vec): a process disseminates a vector vec.
- **indication** obtain(Hash_Value H', Storage_Proof sp'): a process obtains a hash value H' and a storage proof sp'.

The following properties are required:

- *Termination:* Every correct process eventually obtains a hash value and a storage proof.
- *δ-Closeness:* Let t_{first} denote the first time a correct process obtains a hash value and a storage proof. Then, every correct process obtains a hash value and a storage proof by time $\max(\text{GST}, t_{first}) + \delta$.
- *Redundancy:* Let a (faulty or correct) process obtain a storage proof sp' such that, for some hash value H', $\text{valid\_SP}(H', sp') = true$. Then, (at least) $t + 1$ correct processes have cached a vector vec' such that $\text{hash}(vec') = H'$.
- *Integrity:* If a correct process obtains a hash value H' and a storage proof sp', the following holds: $\text{valid\_SP}(H', sp') = true$.

Slow broadcast. In order to solve the vector dissemination problem, we present a simple algorithm (Algorithm 4) which implements *slow broadcast.* In slow broadcast, each process disseminates its vector in "one-by-one" fashion, with a "waiting step" between any two sending events. Specifically, process P_1 broadcasts its vector by (1) sending the vector to P_1 (line 3), and then waiting δ time (line 4), (2) sending the vector to P_2 (line 3), and then waiting δ time (line 4), etc. Process P_2 broadcasts its vector in the same manner, but it waits $\delta \cdot n$ time (line 4). Crucially, if the system is synchronous, the waiting time of P_2 is (roughly) sufficient for P_1 to *completely* disseminate its vector. This holds for any two processes P_i and P_j such that $i < j$.

Algorithm 4 Slow Broadcast: Pseudocode (for process P_i)

```
1: upon broadcast(Vector $vec$):
2:     for each Process $P_j$:
3:         send ⟨SLOW_BROADCAST, $vec$⟩ to $P_j$
4:         wait for $\delta \cdot n^{(i-1)}$ time
5: upon reception of ⟨SLOW_BROADCAST, Vector $vec'$⟩ from process $P_j$:
6:     trigger deliver($vec'$, $P_j$)
```

Vector dissemination algorithm. Our solution is given in Algorithm 5. First, we give a concrete implementation of the $\text{valid\_SP}(\cdot, \cdot)$ function. Given a hash value H and a storage proof sp, $\text{valid\_SP}(H, sp) = true$ if and only if sp is a valid $(n-t)$-threshold signature of $\langle \text{STORED}, H \rangle$.

Let us explain Algorithm 5 from the perspective of a correct process P_i. When P_i starts disseminating its vector vec (line 8), P_i stores its hash (line 9) and slow-broadcasts the vector (line 10). Once P_i receives STORED messages from $n - t$ distinct processes (line 17), P_i combines received partial signatures into a storage proof (line 18). Then, P_i broadcasts (using the best-effort broadcast primitive) the constructed storage proof (line 19).

Whenever P_i receives a storage proof (line 21), P_i checks whether the storage proof is valid (line 22). If it is, P_i rebroadcasts the storage proof (line 23), obtains a hash value and the storage proof (line 24), and stops participating (i.e., sending and processing messages) in vector dissemination (line 25). Observe that, once P_i stops participating in vector dissemination (line 25), it stops participating in slow broadcast, as well.

Proof of correctness and complexity. We start by proving redundancy of Algorithm 5.

Lemma 8. Algorithm 5 satisfies redundancy.

Proof. Let a (correct or faulty) process obtain a storage proof sp' such that $\text{valid\_SP}(H', sp') = true$, for some hash value H'. Hence, $n - t$ processes have signed a STORED message for H' (as $\text{valid\_SP}(H', sp') = true$). Among these $n - t$ processes, at least $t + 1$ are correct (as $n > 3t$). Before

Algorithm 5 Vector Dissemination: Pseudocode (for process P_i)

```
1:  Uses:
2:      Best-Effort Broadcast [19], instance beb      ▷ broadcast with no guarantees if the sender is faulty
3:      Slow Broadcast, instance slow                  ▷ see Algorithm 4
4:  upon init:
5:      Hash_Value $H_i \leftarrow \bot$               ▷ hash of the message $P_i$ slow-broadcasts
6:      Map(Hash_Value $\rightarrow$ Vector) $vectors_i \leftarrow$ empty     ▷ received vectors
7:      Set(Process) $disseminated_i \leftarrow$ empty     ▷ processes who have disseminated a vector
8:  upon disseminate(Vector $vec$):
9:      $H_i \leftarrow$ hash($vec$)
10:     invoke slow.broadcast($vec$)
11: upon slow.deliver(Vector $vec'$, Process $P_j$):
12:     if $P_j \notin disseminated_i$:
13:         $disseminated_i \leftarrow disseminated_i \cup \{P_j\}$
14:         $vectors_i$[hash($vec'$)] $\leftarrow vec'$     ▷ cache $vec'$
15:         send $\langle$STORED, hash($vec'$)$\rangle_{\sigma_i}$ to $P_j$     ▷ acknowledge the reception by sending a signature to $P_j$
16: ▷ acknowledgements are received
17: upon reception of Message $m_j = \langle$STORED, Hash_Value $H'\rangle_{\sigma_j}$ such that $H' = H_i$ from $n - t$ distinct processes:
18:     Storage_Proof $sp \leftarrow Combine(\{\sigma \mid \sigma$ is a signature of a received STORED message$\})$
19:     invoke beb.broadcast($\langle$STORAGE_PROOF, $H_i, sp\rangle$)     ▷ disseminate the storage proof
20: ▷ a storage proof is received
21: upon reception of Message $m = \langle$STORAGE_PROOF, Hash_Value $H'$, Storage_Proof $sp'\rangle$:
22:     if $sp'$ is a valid $(n - t)$-threshold signature of $\langle$STORED, $H'\rangle$:     ▷ check that the storage proof is valid
23:         invoke beb.broadcast($\langle$STORAGE_PROOF, $H', sp'\rangle$)     ▷ rebroadcast the storage proof
24:         trigger obtain($H', sp'$)
25:         stop participating in vector dissemination and slow broadcast
```

sending (and signing) a STORED message for H', all these correct processes have cached a vector vec' (line 14), where hash$(vec') = H'$. Thus, the lemma. □

Next, we prove δ-closeness.

Lemma 9. Algorithm 5 satisfies δ-closeness.

Proof. Let P_{first} be a correct process which obtains a hash value and a storage proof at time t_{first}. Before the aforementioned attainment, P_{first} rebroadcasts the hash value and the storage proof (line 23). Hence, every correct process receives a hash value and storage proof by time $\max(\text{GST}, t_{first}) + \delta$. □

The following lemma proves that, if a correct process P_i starts the dissemination of its vector at time t_i, then every correct process obtains a hash value and a storage proof by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 3\delta$. We emphasize that the $\max(\text{GST}, t_i) + \delta \cdot n^i + 3\delta$ time is not tight; we choose it due to the simplicity of the presentation.

Lemma 10. If a correct process P_i starts the dissemination of its vector at time t_i, every correct process obtains a hash value and a storage proof by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 3\delta$.

Proof. We separate the proof into two cases:

- There exists a correct process which obtains a hash value and a storage proof by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 2\delta$. In this case, the statement of the lemma holds as every correct process obtains a hash value and a storage proof by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 3\delta$ due to the "rebroadcasting step" (line 23).

- There does not exist a correct process which obtains a hash value and a storage proof by time $T = \max(\text{GST}, t_i) + \delta \cdot n^i + 2\delta$. Hence, no process stops participating in vector dissemination by time T, i.e., no process executes line 25 by time T. Every correct process receives a SLOW_BROADCAST message from process P_i by time $\max(\text{GST}, t_i) + \delta \cdot n^i + \delta$.
  Thus, by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 2\delta$, P_i receives $n - t$ partial signatures (line 17). Finally, by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 3\delta$, every correct process receives a STORAGE_PROOF message from P_i (line 21), and obtains a hash value and a storage proof (line 24). In this case, the statement of the lemma holds.

As the statement of the lemma holds in both cases, the proof is concluded. □

The next lemma proves that Algorithm 5 satisfies termination.

Lemma 11. Algorithm 5 satisfies termination.

PROOF. Follows directly from Lemma 10. □

Next, we prove integrity.

Lemma 12. Algorithm 5 satisfies integrity.

PROOF. Follows from the check at line 22. □

Therefore, Algorithm 5 solves the vector dissemination problem.

THEOREM 9. *Algorithm 5 is correct.*

Lastly, we prove that the communication complexity of Algorithm 5 is $O(n^2)$. Recall that the communication complexity denotes the number of bits sent by correct processes at and after GST.

THEOREM 10. *Let no correct process start the dissemination of its vector after time GST + δ. Then, the communication complexity of Algorithm 5 is $O(n^2)$.*

?

PROOF. Let i be the minimum index such that (1) process P_i is correct, and (2) P_i sends a SLOW_BROADCAST message at some time $\geq$ GST. If i does not exist, the lemma trivially holds.

Let t_i denote the time at which P_i starts the dissemination of its vector (line 8). By assumption, $t_i \leq \text{GST} + \delta$. Every correct process obtains a hash value and a storage proof by time $\max(\text{GST}, t_i) + \delta \cdot n^i + 3\delta$ (by Lemma 10). Thus, as $t_i \leq \text{GST} + \delta$, every correct process obtains a hash value and a storage proof by time $\text{GST} + \delta \cdot n^i + 4\delta$. Moreover, by time $\text{GST} + \delta \cdot n^i + 4\delta$, all correct processes stop sending SLOW_BROADCAST messages (due to line 25).

Let P_j be a correct process such that $j > i$. Due to the slow broadcast primitive (Algorithm 4), P_j has a "waiting step" of (at least) $\delta \cdot n^i$ time. Therefore, during the $[\text{GST}, \text{GST} + \delta \cdot n^i + 4\delta]$ period, P_j can send only $O(1)$ SLOW_BROADCAST messages. Thus, at most one correct process (i.e., P_i) sends more than $O(1)$ SLOW_BROADCAST messages during the $[\text{GST}, \text{GST} + \delta \cdot n^i + 4\delta]$ period; that process sends at most n SLOW_BROADCAST messages. As each message is of size $O(n)$ (since it carries a vector of $n - t$ values), the communication complexity of Algorithm 5 is $O(n) \cdot O(1) \cdot O(n) + 1 \cdot O(n) \cdot O(n) = O(n^2)$. □

C.3.2 Vector Consensus with $O(n^2 \log n)$ Communication Complexity. Finally, we are ready to present our vector consensus algorithm (Algorithm 6) with subcubic communication complexity. Our algorithm consists of three building blocks: (1) vector dissemination (Appendix C.3.1), (2) QUAD (§5.2.1), and (3) ADD [29], an algorithm for asynchronous data dissemination. In Algorithm 6, we rely on a specific instance of QUAD in which (1) each proposal is a hash value, and (2) given a hash value H and a (QUAD) proof Σ,[4] verify(H, Σ) = *true* if and only if valid_SP(H, Σ) = *true* (recall the vector dissemination problem; Appendix C.3.1). Below, we briefly explain ADD.

[4]Do not confuse QUAD proofs with storage proofs of the vector dissemination problem (Appendix C.3.1).

ADD. This algorithm solves the *data dissemination* [29] problem defined in the following way. Let M be a data blob which is an input of (at least) $t+1$ correct processes; other correct processes have $\bot$ as their input. The data dissemination problem requires every correct process to eventually output M, and no other message. The key feature of ADD is that it solves the problem with $O(n^2 \log n)$ communication complexity. (For full details on ADD, refer to [29].)

Description of vector consensus. We give the description of Algorithm 6 from the perspective of a correct process P_i. When P_i proposes its value (line 10), it disseminates the value (using the best-effort broadcast primitive) to all processes (line 11). Once P_i receives proposals of $n-t$ distinct processes (line 16), it constructs an input configuration (line 17), and starts disseminating it (line 18).[5]

When P_i obtains a hash value H and a storage proof sp (line 19), P_i proposes (H, sp) to QUAD (line 21). Observe that verify$(H, sp) = true$ (due to the integrity property of vector dissemination). Once P_i decides from QUAD (line 22), it starts ADD (line 24). Specifically, P_i checks whether it has cached an input configuration whose hash value is H' (line 23). If so, P_i inputs the input configuration to ADD; otherwise, P_i inputs $\bot$. Once P_i outputs an input configuration from ADD (line 25), it decides it (line 26).

Algorithm 6 $O(n^2 \log n)$ Vector Consensus: Pseudocode (for process P_i)

```
1:  Uses:
2:      Best-Effort Broadcast [19], instance beb        ▹ broadcast with no guarantees if the sender is faulty
3:      Vector Dissemination, instance disseminator                                        ▹ see Algorithm 5
4:      QUAD [23], instance quad
5:      ADD [29], instance add
6:  upon init:
7:      Integer $received\_proposals_i \leftarrow 0$                          ▹ the number of received proposals
8:      Map(Process $\rightarrow \mathcal{V}_I$) $proposals_i \leftarrow$ empty                      ▹ received proposals
9:      Map(Process $\rightarrow$ Message) $messages_i \leftarrow$ empty               ▹ received PROPOSAL messages
10: upon propose($v \in \mathcal{V}_I$):
11:     invoke beb.broadcast($\langle$PROPOSAL, $v\rangle_{\sigma_i}$)                        ▹ broadcast a signed proposal
12: upon reception of Message $m = \langle$PROPOSAL, $v_j \in \mathcal{V}_I\rangle_{\sigma_j}$ from process $P_j$ and $received\_proposals_i < n - t$:
13:     $received\_proposals_i \leftarrow received\_proposals_i + 1$
14:     $proposals_i[P_j] \leftarrow v_j$
15:     $messages_i[P_j] \leftarrow m$
16:     if $received\_proposals_i = n - t$:          ▹ received $n - t$ proposals; can start disseminating
17:         Input_Configuration $vector \leftarrow$ input configuration constructed from $proposals_i$
18:         invoke disseminator.disseminate($vector$)
19: upon disseminator.obtain((Hash_Value $H$, Storage_Proof $sp$)):
20:     if have not yet proposed to QUAD:
21:         invoke quad.propose(($H, sp$))
22: upon quad.decide((Hash_Value $H'$, Storage_Proof $sp'$)):
23:     Input_Configuration $vector' \leftarrow$ a cached vector whose hash value is $H'$                 ▹ can be $\bot$
24:     invoke add.input($vector'$)
25: upon add.output(Input_Configuration $vector''$):
26:     trigger decide($vector''$)
```

[5]Recall that this input configuration is actually a vector of $n - t$ values.

Proof of correctness and complexity. We start by proving that Algorithm 6 is correct.

THEOREM 11. *Algorithm 6 is correct.*

PROOF. Let us prove that Algorithm 6 satisfies all properties of vector consensus.

- *Agreement:* Due to *Agreement* of QUAD, no two correct processes decide different pairs from QUAD (line 22). Hence, for all correct processes which input a non-$\bot$ input configuration to ADD (line 24), they input the same input configuration. Thus, due to the specification of ADD, all correct processes output the same input configuration from ADD (line 25). *Agreement* is satisfied.
- *Termination:* Every correct process broadcasts its proposal (line 11). Thus, every correct eventually receives $n-t$ proposals (line 16), and starts the dissemination of an input configuration (line 18). Due to the termination property of vector dissemination (Lemma 11), every correct process eventually obtains a hash value and a storage proof (line 19). Thus, every correct process eventually proposes to QUAD (line 21). Due to *Termination* of QUAD, every correct process eventually decides from QUAD (line 22). As the pair decided from QUAD includes a storage proof (due to the specification of QUAD), at least $t+1$ correct processes have cached an input configuration whose hash value is decided from QUAD (by the redundancy property of vector dissemination). Thus, due to the specification of ADD, every correct process eventually outputs an input configuration from ADD (line 25). *Termination* is satisfied.
- *Vector Validity:* Let vec' be an input configuration of $n-t$ proposals decided by a correct process (line 26). Hence, a storage proof sp' such that $\mathsf{valid\_SP}(\mathsf{hash}(vec'), sp') = \mathit{true}$ is obtained (due to the specification of ADD). Therefore, vec' is cached by (at least) $t+1$ correct processes (due to the redundancy property of vector dissemination). Before a correct process caches a vector (Algorithm 5), it verifies that it is associated with corresponding PROPOSAL messages; we omit this check for brevity. As correct processes only send PROPOSAL messages for their proposals (line 11), *Vector Validity* is satisfied.

The theorem holds. □

Lastly, we show the communication complexity of Algorithm 6.

THEOREM 12. *The communication complexity of Algorithm 6 is $O(n^2 \log n)$.*

PROOF. The communication complexity of a single best-effort broadcast instance is $O(n)$. Every correct process starts the dissemination of its vector by time GST + δ (as every correct process receives $n-t$ proposals by this time). Thus, the communication complexity of vector dissemination is $O(n^2)$ (by Theorem 10). The communication complexity of QUAD is $O(n^2)$ (see [23]). Moreover, the communication complexity of ADD is $O(n^2 \log n)$ (see [29]). As Algorithm 6 is a composition of the aforementioned building blocks, its communication complexity is $n \cdot O(n) + O(n^2) + O(n^2) + O(n^2 \log n) = O(n^2 \log n)$. □

D EXTENDED FORMALISM

In this section, we give intuition behind an extension of our formalism which is suitable for the analysis of blockchain-specific validity properties, such as *External Validity* [18, 20, 78]. *External Validity* stipulates that any decided value must satisfy a predetermined logical predicate. However, the "difficulty" of this property is that the logical predicate (usually) verifies a cryptographic proof, which processes might not know "in advance" (see Appendix D.1).

In a nutshell, we make our original formalism more expressive by (1) making the input ($\mathcal{V}_I$) and output ($\mathcal{V}_O$) spaces "unknown" to the processes, and (2) taking into account "proposals" of faulty processes. In the rest of the paper:

- We refer to the formalism introduced in the main body of the paper as the "original formalism".
- We refer to the formalism we introduce below as the "extended formalism".

We start by giving an intuition behind our extended formalism (Appendix D.1). Then, we introduce some preliminaries (Appendix D.2). Finally, we define our extended formalism (Appendix D.3).

D.1 Intuition

In the original formalism, processes "know" the entire input space $\mathcal{V}_I$ and the entire output space $\mathcal{V}_O$. That is, processes are able to "produce" any value which belongs to $\mathcal{V}_I$ or $\mathcal{V}_O$. However, this assumption limits the expressiveness of our formalism as it is impossible to describe a Byzantine consensus problem in which input or output spaces are not "known". Let us give an example.

Imagine a committee-based blockchain which establishes two roles:

- *Clients* are the users of the blockchain. They issue *signed* transactions to the blockchain.
- *Servers* are the operating nodes of the blockchain. Servers receive signed transactions issued by the clients, and solve the Byzantine consensus problem to agree on the exact order transactions are processed.

As servers propose transactions *signed by the clients* and they do not have access to the private keys of the clients, servers do not "know" the input space $\mathcal{V}_I$ nor the output space $\mathcal{V}_O$ of the Byzantine consensus problem. Hence, our original formalism cannot describe the Byzantine consensus problem in the core of the aforementioned blockchain.

Extended vs. original formalism. As highlighted above, the main difference between the two formalisms is that the extended one allows us to specify the "knowledge level" of the input and output spaces. In the extended formalism, a process is able to "learn" output values by observing input values. That is, we define a *discovery function* that defines which output values are learned given observed input values. In the committee-based blockchain example, once a server observes signed (by the issuing clients) transactions tx_1 and tx_2, it learns the following output values: (1) tx_1, (2) tx_2, (3) $tx_1||tx_2$, and (4) $tx_2||tx_1$.[6]

The second difference between the original and the extended formalism is that the extended formalism takes into account "proposals" of faulty processes. Indeed, the original formalism does not enable us to define which values are admissible given the adversary's knowledge of the input space. Think of the aforementioned example with a blockchain system. If no process (correct or faulty) obtains a transaction tx, tx cannot be decided. However, if *only* a faulty process obtains a transaction tx, tx could be an admissible decision. This scenario *can* be described by the extended formalism, while it *cannot* by the original one.

D.2 Preliminaries

We denote by $\mathcal{V}_I$ the input space of Byzantine consensus, i.e., processes propose values contained in $\mathcal{V}_I$. Similarly, $\mathcal{V}_O$ denotes the output space, i.e., processes decide values which belong to $\mathcal{V}_O$.

Membership functions. We define two *membership functions*:

- $\mathsf{valid\_input} : \{0,1\}^* \to \{\mathit{true}, \mathit{false}\}$: Intuitively, the $\mathsf{valid\_input}(\cdot)$ function specifies whether a bit-sequence belongs to the input space $\mathcal{V}_I$.
- $\mathsf{valid\_output} : \{0,1\}^* \to \{\mathit{true}, \mathit{false}\}$: Intuitively, the $\mathsf{valid\_output}(\cdot)$ function specifies whether a bit-sequence belongs to the output space $\mathcal{V}_O$.

We assume that each process has access to these two functions. That is, each process can verify whether an arbitrary sequence of bits belongs to the input ($\mathcal{V}_I$) or output ($\mathcal{V}_O$) space. In the case of

[6]We denote by "$||$" the concatenation operation.

a committee-based blockchain (Appendix D.1), the membership functions are simply signature-verification functions.

Discovery function. We define a function discover: $2^{\mathcal{V}_I} \longrightarrow 2^{\mathcal{V}_O}$. Given a set of proposals $V_I \subseteq \mathcal{V}_I$, $\text{discover}(V_I) \subseteq \mathcal{V}_O$ specifies the set of decisions which are "discoverable" by V_I. We assume that each process has access to the $\text{discover}(\cdot)$ function. Moreover, for any two sets V_I^1, V_I^2 with $V_I^1 \subseteq V_I^2$, $\text{discover}(V_I^1) \subseteq \text{discover}(V_I^2)$; in other words, "knowledge" of the output space can only be improved upon learning more input values.

Let us take a look at the committee-based blockchain example again (Appendix D.1). If a server obtains a proposal tx, it learns tx as a potential decision. We model this "deduction" concept using the $\text{discover}(\cdot)$ function: $\text{discover}\big(\{tx\}\big) = \{tx\}$.

Adversary pool. Given an execution $\mathcal{E}$, $\mathcal{P}(\mathcal{E}) \subseteq \mathcal{V}_I$ defines the *adversary pool* in $\mathcal{E}$. Informally, the adversary pool represents the input values the adversary "knows". In the example of a committee-based blockchain (Appendix D.1), the adversary pool is a set of signed transactions which the adversary "learns" from the clients.

We underline that the adversary pool is an abstract concept. Specifically, the adversary pool represents the "starting knowledge" the adversary has. However, the notion of the "starting knowledge" must be precisely defined once all particularities of the exact considered system are taken into account. Due to the sophisticated details as the aforementioned one, we believe that the formalism suitable for blockchain-specific validity properties deserves a standalone paper.

D.3 Validity

We start by restating the definition of process-proposal pairs. A *process-proposal* pair is a pair (P, v), where (1) $P \in \Pi$ is a process, and (2) $v \in \mathcal{V}_I$ is a proposal. Given a process-proposal pair $pp = (P, v)$, $\text{proposal}(pp) = v$ denotes the proposal associated with pp.

An *input configuration* is a tuple $\big[pp_1, pp_2, ..., pp_x, \rho\big]$ of x process-proposal pairs and a set $\rho \subseteq \mathcal{V}_I$, where (1) $n - t \leq x \leq n$, (2) every process-proposal pair is associated with a distinct process, and (3) if $x = n$, $\rho = \emptyset$. Intuitively, an input configuration represents an assignment of proposals to correct processes, as well as a "part" of the input space known to the adversary. For example, an input configuration $\big[(P_1, v), (P_2, v), (P_3, v), \{v, v', v''\}\big]$ describes an execution in which (1) only processes P_1, P_2, and P_3 are correct, (2) processes P_1, P_2, and P_3 propose the same value v, and (3) faulty processes know only v, v', and v''.

We denote by $\mathcal{I}$ the set of all input configurations. For every input configuration $c \in \mathcal{I}$, we denote by $c[i]$ the process-proposal pair associated with process P_i; if such a process-proposal pair does not exist, $c[i] = \bot$. Moreover, we define by $\text{pool}(c)$ the set of input values associated with c (the "ρ" field of c). Next, $\pi(c) = \{P_i \in \Pi \mid c[i] \neq \bot\}$ denotes the set of all processes included in c. Finally, $\text{correct\_proposals}(c) = \{v \in \mathcal{V}_I \mid \exists i \in [1, n] : c[i] \neq \bot \land \text{proposal}(c[i]) = v\}$ denotes the set of all proposals of correct processes (as specified by c).

Given (1) an execution $\mathcal{E} \in \mathit{execs}(\mathcal{A})$, where $\mathcal{A}$ is an algorithm which exposes the $\text{propose}(\cdot)/\text{decide}(\cdot)$ interface, and (2) an input configuration $c \in \mathcal{I}$, we say that $\mathcal{E}$ *corresponds* to c if and only if (1) $\pi(c) = \mathit{Corr}_{\mathcal{A}}(\mathcal{E})$, (2) for every process $P_i \in \mathit{Corr}_{\mathcal{A}}(\mathcal{E})$, P_i's proposal in $\mathcal{E}$ is $\text{proposal}(c[i])$, and (3) $\mathcal{P}(\mathcal{E}) = \text{pool}(c)$. We denote by $\text{corresponding}(\mathcal{E}) = c$ the input configuration to which $\mathcal{E}$ corresponds.

A validity property val is a function $\mathit{val} : \mathcal{I} \rightarrow 2^{\mathcal{V}_O}$ such that, for every input configuration $c \in \mathcal{I}$, $\mathit{val}(c) \neq \emptyset$. Algorithm $\mathcal{A}$, where $\mathcal{A}$ exposes the $\text{propose}(\cdot)/\text{decide}(\cdot)$ interface, *satisfies* a validity property val if and only if, in every execution $\mathcal{E} \in \mathit{execs}(\mathcal{A})$, no correct process decides a value $v' \notin \mathit{val}\big(\text{corresponding}(\mathcal{E})\big)$. That is, an algorithm satisfies a validity property if and only if correct processes decide only admissible values.

Assumptions on executions. Lastly, we introduce two assumptions that conclude our proposal for the extended formalism.

Assumption 1. For every execution $\mathcal{E}$ of any algorithm $\mathcal{A}$ which solves the Byzantine consensus problem with some validity property, if a correct process P decides a value $v' \in \mathcal{V}_O$ in $\mathcal{E}$, then $v' \in \mathsf{discover}\big(\mathsf{correct\_proposals}(c) \cup \mathsf{pool}(c)\big)$, where $\mathsf{corresponding}(\mathcal{E}) = c$.

Assumption 1 states that correct processes can only decide values which are "discoverable" using all the proposals of correct processes and the knowledge of the adversary. For example, if every correct process proposes the same value $v \in \mathcal{V}_I$ and the adversary pool contains only $v' \in \mathcal{V}_I$, then a correct process can only decide a value from $\mathsf{discover}(\{v, v'\})$.

Next, we introduce an assumption concerned only with the canonical executions (executions in which faulty processes do not take any computational step).

Assumption 2. For every canonical execution $\mathcal{E}$ of any algorithm $\mathcal{A}$ which solves the Byzantine consensus problem with some validity property, if a correct process P decides a value $v' \in \mathcal{V}_O$ in $\mathcal{E}$, then $v' \in \mathsf{discover}\big(\mathsf{correct\_proposals}(c)\big)$, where $\mathsf{corresponding}(\mathcal{E}) = c$.

Intuitively, Assumption 2 states that, if faulty processes are silent, correct processes can only decide values which can be discovered using their own proposals. In other words, correct processes cannot use "hidden" proposals (possessed by the "silent" adversary) to discover a decision.

Finally, we underline that these two assumptions do not completely prevent "unreasonable" executions. For example, given these two assumptions, a (correct or faulty) process is still able to send a message with a value which cannot be discovered using the proposals of correct processes and the adversary pool. Hence, an assumption that prevents such an execution should be introduced. Thus, due to the complexity we envision for the extended formalism, we leave it out of this paper. In the future, we will focus on this interesting and important problem.