Crime and Punishment in Distributed Byzantine Decision Tasks

(Extended Version)

Pierre Civit, *Sorbonne University, CNRS, LIP6*
Seth Gilbert, *NUS Singapore*
Vincent Gramoli, *University of Sydney*
Rachid Guerraoui, *Ecole Polytechnique Fédérale de Lausanne (EPFL)*
Jovan Komatovic, *Ecole Polytechnique Fédérale de Lausanne (EPFL)*
Zarko Milosevic, *Informal Systems*
Adi Seredinschi, *Informal Systems*

***Abstract*—A *decision task* is a distributed input-output problem in which each process starts with its *input value* and eventually produces its *output value*. Examples of such decision tasks are broad and range from consensus to reliable broadcast to lattice agreement. A *distributed protocol* solves a decision task if it enables processes to produce admissible output values despite arbitrary (Byzantine) failures. Unfortunately, it has been known for decades that many decision tasks cannot be solved if the system is overly corrupted, i.e., *safety* of distributed protocols solving such tasks can be violated in unlucky scenarios.**

By contrast, only recently did the community discover that some of these distributed protocols can be made *accountable* by ensuring that correct processes irrevocably detect some faulty processes responsible for any safety violation. This realization is particularly surprising (and positive) given that accountability is a powerful tool to mitigate safety violations in distributed protocols. Indeed, exposing crimes and introducing punishments naturally incentivize exemplarity.

In this paper, we propose a generic transformation, called τ_{scr}, of any *non-synchronous* distributed protocol solving a decision task into its accountable version. Our τ_{scr} transformation is built upon the well-studied simulation of crash failures on top of Byzantine failures and increases the communication complexity by a quadratic multiplicative factor in the worst case.

I. Introduction

There are known limitations to the decision tasks distributed protocols can solve. For decades it has been known that, without additional assumptions (e.g., synchronous communication), no distributed protocol ensures the safety of the consensus decision task if more than $t_0 = \lceil n/3 \rceil - 1$ processes are Byzantine [29]. Similar results apply to set agreement [11] or lattice agreement [2]. These safety violations can be dramatic. Let us consider a blockchain application as an example where individuals store valuable assets. An agreement violation in the blockchain context could lead two correct processes to disagree about the current state of the blockchain. As a result of this disagreement, an attacker could convince some correct processes that they transferred assets while it is not the case: an undesirable situation leading to what is called a *double spending*.

Accountability is a potent property in mitigating safety violations. In the context of distributed protocols, accountability enables correct processes to conclusively detect culprits and obtain proof of their misbehavior after safety has been violated. Exposing culprits naturally incentivizes participants to behave correctly. In the synchronous setting, one can require processes to exchange authenticated messages and expose any non-responsive faulty process [23]. However, such an approach does not guarantee an attainment of irrefutable proof of misbehavior nor it works in the general setting. Only recently has the community devised accountable distributed protocols to solve decision tasks, like consensus [12], [34], for the general setting. As far as we know, each of these presents an accountable variant of a very specific distributed protocol, but no generic solution exists.

In this paper, we propose a generic transformation, called τ_{scr}, of any non-synchronous distributed protocol solving a decision task into an accountable version of the same protocol. First, we show that one must be able to detect *commission faults* – faults that occur once a faulty process invalidly sends a message – in order to achieve accountability in a non-synchronous setting. Indeed, we prove that (1) every irrevocable detection must be based on a detected commission fault (otherwise, a correct process can falsely be detected), and (2) (luckily for accountability!) whenever safety is violated, "enough" processes have committed commission faults. Furthermore, we separate all commission faults into (1) *equivocation faults*, faults associated with an act of claiming conflicting statements, and (2) *evasion faults*, faults that occur once a faulty process sends a message which cannot be sent given the previously received messages. Then, we illustrate that detecting equivocation faults is easier in non-synchronous settings than detecting evasion faults, concluding that equivocation faults are preferable means of violating safety in non-synchronous distributed protocols.

Finally, we observe that the approach exploited by the well-studied simulation [3], [7], [15], [17], [25], [26] of crash failures on top of Byzantine failures can be modified to ensure that evasion faults are *masked* (i.e., their effect is eliminated), thus allowing *only* equivocation faults to violate safety. Such a simulation is achieved using the secure broadcast [7] primitive: (1) each originally sent message is secure-broadcast, and (2) no secure-delivered message "affects" the receiver before

a correct causal past of the message has been established. Hence, no message that is a product of an evasion fault influences a correct process (even if the system is entirely corrupted), implying that all safety violations are necessarily consequences of equivocation faults. We base the τ_{scr} transformation on the aforementioned approach based on secure broadcast. Due to the complexity of the secure broadcast primitive, our transformation increases the communication and message complexities of the original distributed protocol by an $O(n^2)$ multiplicative factor.

Roadmap: We discuss the related work in §II. In §III, we introduce the computational model, distributed protocols, Byzantine decision tasks, and safety violations of these tasks. We define commission faults in §IV and show that accountability in a non-synchronous setting implies the ability to detect these faults. Our generic accountability transformation is introduced in §V. We conclude the paper in §VI. For space limitations, detailed definitions and proofs are delegated to the optional appendix.

II. Related Work

a) Byzantine failures: If a process deviates from a prescribed protocol, it commits a Byzantine failure [29]. The primary technique in tackling Byzantine failures in distributed computing is *masking*, i.e., hiding the effects of these failures [10], [18], [30], [36], [37]. An alternative approach is *detection* of Byzantine failures. Initially, detection of Byzantine failures was incorporated into the design of Byzantine *failure detectors* [19], [27], [31], which were used for solving the consensus [29] problem. Kihlstrom *et al.* [27] define the class of commission faults, which occur if (1) messages with the same header and different content are sent, or (2) an unjustified message is sent. Although quite similar to our definition of commission faults, there is a subtle difference between the definition given in [27] and ours: there exists a faulty behavior that we classify as a commission fault, which is not captured by the definition from [27]. Furthermore, Haeberlen *et al.* [24] studied the problem of generic fault detection in distributed systems. They, as the authors of [27], recognize commission faults as a separate class of Byzantine failures. The definition of commission faults given in [24] is based on the knowledge of correct processes, whereas ours relies on the knowledge of an "all-seeing" external observer. For instance, if a faulty process sends two conflicting messages m_1 and m_2, but only message m_1 is "observed" by a correct process, then the process does *not* commit a commission fault according to the definition given in [24]; our definition classifies such a behavior as a commission fault. The authors of [24] investigate the cost of detecting commission faults in terms of exchanged messages; in contrast, our work is concerned with the number of exchanged bits. Finally, the same authors presented PeerReview [23], a generic accountability add-on for distributed systems. The definition of "detectably faulty" processes given in [23] served as the main inspiration for our definition of commission faults.

b) Simulation of crash failures on top of Byzantine ones: Due to the nature of the crash and Byzantine failures, crash failures are easier to handle than Byzantine ones. Therefore, the community has explored ways of simulating crash failures on top of Byzantine failures [3], [7], [15], [17], [25], [26]. Such a simulation can be seen as a module θ which (1) connects the networking layer to a crash-resilient algorithm Π, and (2) allows only "benign" executions to reach Π by not forwarding any message from the networking layer to Π unless a valid behavior of the sender has previously been established. Thus, all Byzantine processes appear to Π as if they have crashed. We provide a more thorough intuition behind such simulations in §V. In this paper, we observe that the approach exploited by the aforementioned simulations can be reused towards obtaining accountability.

c) Accountability: Accountability, in general, requires correct processes to irrevocably detect faulty processes; such detection can be a part of the "normal flow" of the system [23] or can be demanded only upon some serious safety violations [13]. Observe that accountability does not allow "false detections", i.e., once a process is detected, the detection cannot be revoked (which is the crucial difference from the revocable detections usually performed by failure detectors). Specifically, the concept of accountability in the context of distributed computing is introduced in [23]. The authors describe a generic accountability layer for distributed protocols - PeerReview. The main weakness of PeerReview is that some types of malicious behaviors cannot be exposed in a non-synchronous setting; thus, malicious processes may only be permanently suspected (and never irrevocably detected) in some scenarios. Therefore, PeerReview does not provide "pure" accountability (at least not always). The specific sub-problem of accountable Byzantine consensus has only recently been defined [13] as the problem of solving consensus when possible, and detecting misbehaving participants when agreement is violated. The idea of the proposed solution is to ensure that disagreement always occurs as a result of equivocation, as is the case in τ_{scr}. This solution, called Polygraph, is specific to the DBFT consensus algorithm [18]. Casper [8] is an accountability overlay for blockchain systems. Ways to obtain accountability guarantees for specific "PBFT-like" consensus protocols are proposed in [34]. The authors of [34] aim to guarantee accountability only if the system is not entirely corrupted, i.e., only if the number of faulty processes does not exceed $2n/3$, where n is the total number of processes. Most recently, an efficient method for transforming a distributed protocol into an accountable protocol was proposed [14]; however, it only works for protocols where the decision of all processes is expected to be identical. The technique used in [14] relies on an additional "confirmation" communication round, ensuring that enough faulty processes must equivocate in this round to violate safety. It remains unclear whether (and how) this technique could be adapted to problems in which processes are not required to output identical values (e.g., k-set agreement [11], lattice agreement [2]). Hence, the transformation presented

in [14], although more efficient, is less general than τ_{scr}.

III. Preliminaries

A. Computational Model

We consider a set Ψ of $|\Psi| = n$ asynchronous processes that communicate by exchanging messages. Each process $p \in \Psi$ is assigned a *protocol* Π_p to follow. Formally, a protocol Π_p is a tuple $(\mathcal{S}_p, s_0^p, \mathcal{M}_p, \mathcal{I}_p, \mathcal{O}_p, \mathcal{T}_p)$, where $\mathcal{S}_p$ represents a set of states p can take,[1] $s_0^p \in \mathcal{S}_p$ is the initial state of p, $\mathcal{M}_p$ is a set of messages p can send or receive, $\mathcal{I}_p$ is a set of internal events p can observe, $\mathcal{O}_p$ is a set of internal events p can produce and $\mathcal{T}_p : \mathcal{S}_p \times P(\mathcal{M}_p \cup \mathcal{I}_p) \rightarrow \mathcal{S}_p \times P(\mathcal{M}_p \cup \mathcal{O}_p)$ maps a state and a set of received messages and observed internal events into a new state and a set of sent messages and produced internal events.[2]

A protocol Π_p does not send the same message more than once.[3] Moreover, each message sent by Π_p is properly authenticated, and any incoming duplicate messages or messages that cannot be authenticated are ignored. We assume that Π_p does not reveal the key material, i.e., if a message is signed by a process p and p follows its protocol, then p must have indeed sent the message. Processes can forward messages to other processes, they can include messages in other messages they send, and we assume that an included or forwarded message can still be authenticated. Each message m has a unique sender $\mathit{sender}(m) \in \Psi$ and a unique receiver $\mathit{receiver}(m) \in \Psi$. Finally, we assume a computationally bounded adversary, i.e., signatures of processes that follow their protocol *cannot* be forged.

a) Events, executions & behaviors: We define an *event* as a tuple (p, I, O), where $p \in \Psi$ is a process on which the event occurs, I represents a finite set of received messages and observed internal events and O represents a finite set of sent messages and produced internal events.[4] An *execution* is a well-formed sequence of events: (1) every received message was previously sent, and (2) if the execution is infinite, every sent message is received. Similarly, a *behavior* is a well-formed sequence of events: (1) all events occur on the same process $p \in \Psi$, (2) if a message m with $\mathit{sender}(m) = p$ is received in the behavior, then the message was previously sent in the behavior, and (3) if the behavior is infinite, every message m with $\mathit{receiver}(m) = p$ which is sent in the behavior is received in the behavior. Given an execution α, $\alpha|_p$ denotes the sequence of events in α associated with a process $p \in \Psi$ (i.e., the behavior of p given α).

A behavior $\beta_p = (p, I_1, O_1), (p, I_2, O_2), ...$ is *valid* according to Π_p if and only if it conforms to the assigned protocol Π_p, i.e., if and only if there exists a sequence of states $s_0, s_1, ...$ in $\mathcal{S}_p$ such that $s_0 = s_0^p$ and, for all $i \geq 1$, $\mathcal{T}_p(s_{i-1}, I_i) = (s_i, O_i)$. For every behavior β, we define $\mathit{sent}(\beta)$ (resp., $\mathit{received}(\beta)$) to be the set of sent (resp., received) messages in β. Finally, for every message m, we assume that there exists a valid behavior of $\mathit{sender}(m)$ in which m is sent.

b) Distributed protocols: A tuple $\Pi = (\Pi_p, \Pi_q, ..., \Pi_z)$, where $\Psi = \{p, q, ..., z\}$, is a *distributed protocol*. We assume that sets of messages each process can send or receive are identical (i.e., $\mathcal{M}_p = \mathcal{M}_q$, for all $p, q \in \Psi$); we denote this set of messages by $\mathcal{M}$.

A process p is *correct* in an execution α according to $\Pi = (\Pi_p, \Pi_q, ..., \Pi_z)$ if and only if $\alpha|_p$ is valid according to Π_p. Otherwise, p is *faulty* in α according to Π. If a process is correct in an infinite execution, then infinitely many events occur on the process (i.e., a process correct in an infinite execution is *live*). We denote by $\mathit{Corr}_\Pi(\alpha)$ the set of processes correct in execution α according to Π. Whenever we say that "α is an execution of a distributed protocol Π", we mean that each process is considered correct or faulty in α according to Π. The set of all possible executions of a distributed protocol Π is denoted by $\mathit{execs}(\Pi)$.

c) Communication network: We assume that the communication network is fully-connected and reliable, i.e., correct processes are able to communicate among themselves. Furthermore, we assume that the network is either asynchronous or partially synchronous [20].

If a network is asynchronous, there is no upper bound on message delays. A partially synchronous network behaves as an asynchronous network during some intervals of time, whereas during other intervals, messages are received in a timely fashion. Specifically, there exists an unknown *global stabilization time* (GST) such that there is no upper bound on message delays before GST, whereas there is an unknown upper bound on message delays after GST.

If the communication network of a distributed protocol is asynchronous (resp., partially synchronous), we say that the distributed protocol itself is *asynchronous* (resp., *partially synchronous*). A *non-synchronous* distributed protocol is an asynchronous or a partially synchronous distributed protocol.

B. Decision Tasks

Decision tasks represent an abstraction of distributed input-output problems. Each process has its *input value*. We assume that "$\perp$" denotes the special input value of a process that specifies that the input value is non-existent. A process may eventually produce its *output value*. The "$\perp$" output value of a process means that the process has not yet produced its output value. We denote by I_p (resp., O_p) the input (resp., output) value of process $p \in \Psi$. We note that some processes might never produce their output values if permitted by the definition of a decision task.

Any decision task could be defined as a relation between input and output values of processes. Since we assume that processes might fail (i.e., be Byzantine), we only care about input and output values of correct processes. Formally, at the beginning of each execution, each process is labelled as either *good* or *bad*. If a process is good, the process follows its protocol; otherwise, it may deviate from its protocol. For

[1] We refer to $\mathcal{S}_p$ as the *state set* of Π_p.

[2] We denote by $P(X)$ the power set of X.

[3] This constraint does not affect the generality of distributed protocols we consider since every message can include a nonce.

[4] Observe the difference between events and internal events.

the sake of simplicity, we slightly abuse the notation and use term "correct" (resp., "faulty" or "Byzantine") instead of "good" (resp., "bad"). Therefore, a decision task could be defined as a relation between input and output values of correct processes.

An *input configuration* of a decision task $\mathcal{D}$ is $\nu_I = \{(p, I_p) \text{ with } p \text{ is correct}\}$: an input configuration consists of input values of correct processes. Similarly, an *output configuration* of a decision task is denoted by $\nu_O = \{(p, O_p) \text{ with } p \text{ is correct}\}$: it contains output values of correct processes.

Formally, a decision task $\mathcal{D}$ is a tuple $(\mathcal{I}, \mathcal{O}, \Delta)$, where:

- $\mathcal{I}$ is the set of all possible input configurations of $\mathcal{D}$.
- $\mathcal{O}$ is the set of all possible output configurations of $\mathcal{D}$.
- $\Delta : \mathcal{I} \rightarrow 2^{\mathcal{O}}$, where $\nu_O \in \Delta(\nu_I)$ if and only if the output configuration $\nu_O \in \mathcal{O}$ is admissible given the input configuration $\nu_I \in \mathcal{I}$.

Without loss of generality, we assume that $\Delta(\nu_I) \neq \emptyset$, for every input configuration $\nu_I \in \mathcal{I}$. Moreover, for every $\nu_O \in \mathcal{O}$, there exists $\nu_I \in \mathcal{I}$ such that $\nu_O \in \Delta(\nu_I)$.

a) Solutions: A distributed protocol $\Pi_{\mathcal{D}}$ *solves* a decision task $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \Delta)$ with t_0*-resiliency* if and only if:

- For every $\nu \in \mathcal{I} \cup \mathcal{O}$, $|\nu| \geq n - t_0$, and
- In every execution with up to t_0 Byzantine processes, there exists (an unknown) time T_D such that $\nu_O \in \Delta(\nu_I)$, where $\nu_I \in \mathcal{I}$ denotes the input configuration that consists of input values of all correct processes, $\nu_O \in \mathcal{O}$ denotes the output configuration that consists of output values (potentially $\perp$) of all correct processes and no correct process p with $O_p = \perp$ updates its output value after T_D.

b) Accountable counterparts: We now formally define an accountable counterpart of a distributed protocol (Definition 3). Intuitively, an accountable counterpart of a distributed protocol $\Pi_{\mathcal{D}}$ is a distributed protocol that (1) behaves as $\Pi_{\mathcal{D}}$ in non-corrupted executions, and (2) provides accountability whenever safety is violated.

Let $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \Delta)$ be a decision task. Consider a set $C = \{(p, O_p \neq \perp), (q, O_q \neq \perp), ..., (z, O_z \neq \perp)\}$. We say that set C is *safe-extendable* according to $\mathcal{D}$ if and only if there exists an output configuration $\nu_O \in \mathcal{O}$ such that $(p, O_p) \in \nu_O$, for every $(p, O_p) \in C$. Intuitively, C is safe-extendable if and only if there exists an output configuration in which processes specified by C output specified values. For instance, if $\mathcal{D}$ is Byzantine consensus and $C = \{(p, v), (q, v' \neq v)\}$, then C is not safe-extendable (since correct processes never output different values in the Byzantine consensus task).

We are now ready to formally define when the safety of a decision task is violated by a distributed protocol.

Definition 1 (Safety Violation)**.** Let $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \Delta)$ be a decision task and let $\Pi_{\mathcal{D}}$ be a distributed protocol that solves $\mathcal{D}$. We say that $\Pi_{\mathcal{D}}$ *violates safety* of $\mathcal{D}$ in an execution α if and only if (1) a correct process p_1 outputs $O_{p_1} \neq \perp$ in α, a correct process p_2 outputs $O_{p_2} \neq \perp$ in α, ..., and a correct process p_x outputs $O_{p_x} \neq \perp$ in α, and (2) $C = \{(p_1, O_{p_1}), (p_2, O_{p_2}), ..., (p_x, O_{p_x})\}$ is not safe-extendable according to $\mathcal{D}$.

Note that Definition 1 does not cover a case where the outputs of correct processes are not valid according to their inputs. Such a scenario would arise only if the number of faulty processes is greater than t_0 (under the assumption that $\Pi_{\mathcal{D}}$ solves $\mathcal{D}$ with t_0-resiliency). However, the Δ function is *not* defined in this case (since $|\nu| \geq n - t_0$, for every $\nu \in \mathcal{I} \cup \mathcal{O}$).

Before defining an accountable counterpart of a distributed protocol, we define proof of culpability of a process. Proof of culpability is self-contained evidence that the corresponding process is faulty. In our work, as in many previous ones [13], [14], proof of culpability is a set of messages that a correct process would never send "together".

Definition 2 (Proof of Culpability)**.** Let Π be a distributed protocol. A set of messages M is *proof of culpability* of a process p according to Π if and only if:

- $\mathit{sender}(m) = p$, for every $m \in M$, and
- no execution α of Π exists such that (1) p sends every message $m \in M$ in α, and (2) p is correct in α.

At last, we are able to formally define an accountable counterpart of a distributed protocol.

Definition 3 (Accountable Counterpart)**.** Let $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \Delta)$ be a decision task. Let $\Pi_{\mathcal{D}}$ be an asynchronous (resp., a partially synchronous) distributed protocol that solves $\mathcal{D}$ with t_0-resiliency. An asynchronous (resp., a partially synchronous) distributed protocol $\bar{\Pi}_{\mathcal{D}}$ is an *accountable counterpart* of $\Pi_{\mathcal{D}}$ with factor $f \in [1, t_0]$ according to basis if there exists a homomorphic transformation $(\bar{\Pi}_{\mathcal{D}}, \Pi_{\mathcal{D}}, \mu_e)$ with $\mu_e : \mathit{execs}(\bar{\Pi}_{\mathcal{D}}) \rightarrow \mathit{execs}(\Pi_{\mathcal{D}})$ that satisfies the following:[5]

- *Solution Preservation:* $\bar{\Pi}_{\mathcal{D}}$ solves $\mathcal{D}$ with f-resiliency.
- *Accountability:* If safety of $\mathcal{D}$ is violated by $\bar{\Pi}_{\mathcal{D}}$, then every correct process detects at least $t_0 + 1$ processes faulty according to $\bar{\Pi}_{\mathcal{D}}$ and obtains proof of culpability of every detected process according to $\bar{\Pi}_{\mathcal{D}}$.
- *Syntactic Correspondence:* Let $\mathit{execs}(\Pi_{\mathcal{D}}, t_0)$ represent the set of all executions of $\Pi_{\mathcal{D}}$ with up to t_0 faulty processes. Then, the following holds:
  - Let $\bar{\alpha}$ be an execution of $\bar{\Pi}_{\mathcal{D}}$. If a process p is correct in $\bar{\alpha}$, then p is correct in $\mu_e(\bar{\alpha})$.
  - For every execution $\alpha \in \mathit{basis}$, where $\mathit{basis} \subseteq \mathit{execs}(\Pi_{\mathcal{D}}, t_0)$, there exists an execution $\bar{\alpha}$ of $\bar{\Pi}_{\mathcal{D}}$ such that $\alpha = \mu_e(\bar{\alpha})$.

Definition 3 is inspired by the definition of the homomorphic transformation μ_e presented in [24]. The difference is that our definition specifies which executions of the original distributed protocol are preserved (all executions that belong to basis), whereas the μ_e transformation does not. Other formal definitions of simulation mechanisms have been previously proposed [4], [5], [16], [32]. However, to the best of our knowledge, Definition 3 and the aforementioned

[5]Homomorphic transformations are formally defined in Definition 12.

μ_e formalism [24] are the only formulations that assume an asynchronous and Byzantine environment.

IV. Commission faults

This section is devoted to defining *commission faults*, a specific type of faults Byzantine processes could experience. We show that accountability in non-synchronous environments implies the ability to detect commission faults by proving that (1) irrevocable detections must be based on committed commission faults (otherwise, a correct process can wrongly be detected), and (2) whenever safety is violated, "enough" processes have committed commission faults (therefore, accountability is indeed possible).

A. Definition & Importance

Informally, a commission fault occurs once a faulty process sends a message a correct process would not send given the ongoing execution. We start by introducing an assumption that helps us define commission faults in a simple manner. The assumption plays a significant role in the formalism we present. It states that a message m sent by a process p is sent "at the end" of *exactly* one valid behavior of p.

Assumption 1 (Message-Behavior Mapping)**.** Consider a protocol Π_p assigned to process $p \in \Psi$ and a message $m \in \mathcal{M}$ with $\mathsf{sender}(m) = p$. There exists exactly one finite behavior $\beta_p = (p, I_1, O_1), ..., (p, I_h, O_h)$ such that (1) no duplicate or non-authenticated messages are received in β_p, (2) β_p is valid according to Π_p, and (3) $m \in O_h$. In this case, we write $m \mapsto \beta_p$.

Note that Assumption 1 is not a restrictive assumption. Namely, every protocol could be easily (although with a certain cost) transformed into a protocol that satisfies the assumption by encoding the entire ongoing execution in a sent message. Importantly, our τ_{scr} transformation (see §V) is *not* built upon Assumption 1, i.e., the assumption is important *solely* for defining commission faults.

Next, we define the message justification of a message. A set of messages $\mathcal{J}_m$ is the message justification of a message m if and only if $\mathcal{J}_m = \mathsf{received}(\beta_p)$, where $m \mapsto \beta_p$.

Definition 4 (Message Justification)**.** Consider a protocol Π_p assigned to process $p \in \Psi$ and a message $m \in \mathcal{M}$ with $\mathsf{sender}(m) = p$. Let β_p be a finite behavior such that $m \mapsto \beta_p$. The *message justification* of m is the $\mathsf{received}(\beta_p)$ set of messages; the message justification of m is denoted by $\mathcal{J}_m$.

Because of Assumption 1, each message has precisely one message justification. Next, we introduce equivocation. This term is well-known in the literature, and it is usually associated with an act of claiming multiple conflicting statements (e.g., "mutant" messages with the same header in [27]). We slightly expand the notion of equivocation to mean that a faulty process claims two statements that could not be stated jointly by a correct process (i.e., in a valid behavior).[6]

Definition 5 (Equivocation)**.** Let α be an execution of a distributed protocol Π. Consider a process $p \in \Psi$ and its behavior $\beta_p = \alpha|_p$. Process p commits an *equivocation* with respect to a message $m \in \mathsf{sent}(\beta_p)$ in α if and only if there exists a message $m' \in \mathsf{sent}(\beta_p)$ such that neither (β_p^m is a prefix of $\beta_p^{m'}$) nor ($\beta_p^{m'}$ is a prefix of β_p^m), where $m \mapsto \beta_p^m$ and $m' \mapsto \beta_p^{m'}$. In this case, m is *conflicting* with m'.

Note that conflicting messages m and m' do not need to be "produced" by valid finite behaviors, i.e., equivocation only requires conflicting messages to be sent. A correct process is certain that a process q is faulty once it observes conflicting messages sent by q. That is, deducing that a process is faulty follows directly from observing "products" of equivocation. Observe that proof of culpability (see Definition 2) proves that the detected process has committed an equivocation.

Evasion faults occur once a process sends a message without previously receiving all the messages necessary for the message to be sent.

Definition 6 (Evasion Fault)**.** Let α be an execution of a distributed protocol Π. Consider a process $p \in \Psi$ and its behavior $\beta_p = \alpha|_p$. Process p commits an *evasion fault* with respect to a message $m \in \mathsf{sent}(\beta_p)$ in α if and only if there exists a message $m' \in \mathcal{J}_m$ which is not received in β_p *before* (the first instance of) m is sent.[7]

Note that once correct processes observe a message m that is a "product" of an evasion fault, they are not aware that this indeed represents a manifestation of the fault. The reason is that evasion faults are concerned not only with sent, but also with *received* messages. In other words, it must be known not only which messages were sent, but also which messages were (not) received in order for a process that commits an evasion fault to be detected.

At last, we are ready to define commission faults. As mentioned in §II, our definition is inspired by the definition of "detectably faulty" processes from [23].

Definition 7 (Commission Fault)**.** Let α be an execution of a distributed protocol Π. Consider a process $p \in \Psi$ and its behavior $\beta_p = \alpha|_p$. Process p commits a *commission fault* with respect to a message $m \in \mathsf{sent}(\beta_p)$ in α if and only if p commits an equivocation or an evasion fault with respect to m in α.

Finally, we state the central results of the section: (1) every irrevocable detection must be based on a committed commission fault, and (2) whenever safety is violated, commission faults have been committed. Due to the lack of space, we provide complete formal proofs of the following theorems in Appendix A.

Theorem 1. *Let Π be a non-synchronous distributed protocol. Let α be an execution of Π such that a correct process p detects*

[6] Note that conflicting messages do not necessarily have the same header (as is the case in [27]). This represents the very subtle difference between our definition of equivocation faults and the definition presented in [27].

[7] If p sends m multiple times, then p commits an evasion fault if and only if there exists a message $m' \in \mathcal{J}_m$ which is not received before the first instance of m is sent.

a faulty process q without detecting any commission fault of q. Then, there exists an execution α' of Π such that (1) correct process p detects q, and (2) q is correct in α'.

Proof sketch. Since p does not detect any commission fault committed by q, there exists an execution α'' such that (1) p does not distinguish α and α'', and (2) q does not commit any commission fault in α''. Due to the fact that a correct process never detects a correct process, q is faulty in α'' (even though it does not commit commission faults). Finally, we create another execution α' in the following manner:

1) We start with $\alpha' \leftarrow \alpha''$.
2) We "repair" the behavior of q by selecting a valid behavior β_q of q such that all messages sent by q in α'' are sent in β_q.
3) For every message m, where m is sent in β_q and m is not sent in α'', the reception of m is delayed in α'.

The obtained α' execution satisfies the following properties: (1) α' cannot be distinguished from α'' by p, and (2) q is correct in α'. Therefore, p and q are correct in α' and p detects q in α', which concludes the theorem.

Theorem 2. *Let $\Pi_{\mathcal{D}}$ be a non-synchronous distributed protocol that solves a decision task $\mathcal{D}$ with t_0-resiliency. Let α be an execution of $\Pi_{\mathcal{D}}$ in which $\Pi_{\mathcal{D}}$ violates safety of $\mathcal{D}$. At least $t_0 + 1$ distinct processes commit commission faults in α.*

Proof sketch. By contradiction, let us assume that there exists an execution α of $\Pi_{\mathcal{D}}$ such that (1) $\Pi_{\mathcal{D}}$ violates safety of $\mathcal{D}$ in α, and (2) up to t_0 distinct processes commit commission faults in α. We construct an execution α' of $\Pi_{\mathcal{D}}$ in the following manner:

1) We start with $\alpha' \leftarrow \alpha$.
2) For every process f, where f is a faulty processes that does *not* commit any commission fault in α, we "repair" the behavior of f by selecting a valid behavior β_f of f such that all messages sent by f in α are sent in β_f.
3) For every process f, where f is a faulty processes that does not commit any commission fault in α, and every message m, where m is sent in β_f and m is not sent in α, the reception of m is delayed in α'.

By construction, $\Pi_{\mathcal{D}}$ violates safety of $\mathcal{D}$ in α' and there exist up to t_0 faulty processes in α'. We reach a contradiction with the fact that $\Pi_{\mathcal{D}}$ solves $\mathcal{D}$ with t_0-resiliency.

B. Detection

In this subsection, we discuss the detection mechanisms for equivocation and evasion faults. We provide an intuition of why we build our τ_{scr} transformation (see §V) around the idea of *masking* evasion faults, thus allowing *only* equivocation faults to cause safety violations.

a) Detecting equivocation: As mentioned in the previous subsection, once a correct process p observes conflicting messages sent by a process s, p immediately concludes that s is faulty. The reason is that no correct process ever sends conflicting messages. Thus, for an equivocation that impacts correct processes to be detected, it is sufficient to ensure that all correct processes eventually observe all messages received by correct processes. This protocol design can be achieved by having correct processes rebroadcasting every "learned" message. Such a solution introduces a quadratic communication complexity overhead.

b) Detecting evasion faults: In the case of evasion faults, messages sent by a faulty sender do not provide self-contained proof of its misbehavior. Specifically, a correct process that aims to detect an evasion fault needs to be aware of which messages are (not) received by the sender.

We provide a simple scenario that illustrates why detecting evasion faults might be more cumbersome than detecting equivocation; the summary of the scenario is presented in Figure 1. Consider processes r, p, q and s and a distributed protocol in which process r sends m_r to p, process p sends m_p to q upon receiving m_r and process q sends m_q to s upon receiving m_p. Suppose that process s needs to detect whether an evasion fault with respect to m_q has occurred upon reception of m_q.

We first investigate an execution α_2 in which processes q and s are correct and q sends m_q. Note that q receives m_p in α_2. In α_2, it is necessary for q to piggyback m_p in m_q. Let us explain why. Suppose that q does *not* piggyback m_p in m_q in α_2 (illustrated in Figure 1). Then, at the moment of reception of m_q, process s cannot distinguish α_2 from α_3, where α_3 is an execution in which q is faulty and commits an evasion fault with respect to m_q. Only processes that can distinguish α_3 from α_2 are (1) process p, since it does not send m_p in α_3 and it sends m_p in α_2, and (2) process q since it does not receive m_p in α_3 and it receives m_p in α_2. However, we are able to create continuations of α_2 and α_3 that are indistinguishable for "sufficiently long" to process s (and r, since r is correct in α_2 and α_3) in the following manner:

- In α_2, messages sent by processes p and q are delayed.
- In α_3, processes p and q are silent.

Since s must detect the evasion fault in the continuation of α_3 and it must not detect the evasion fault in the continuation of α_2, we conclude that the detection problem cannot be solved. That is why q needs to piggyback m_p in m_q, i.e., the piggybacking would create a difference in executions α_2 and α_3 and allow process s to detect the evasion fault in α_3.

Furthermore, what happens if s also aims to detect a potential evasion fault with respect to $m_p \in \mathcal{J}_{m_q}$ upon reception of m_q? In this case, process p must piggyback m_r in m_p. Importantly, m_r must be piggybacked in a way which does not allow process q to extract m_p *without* extracting m_r. In the idealized PKI, process p could achieve this by sending a message $[(m_r)_{\sigma_r}, m_p]_{\sigma_p}$, where σ_p (resp., σ_r) is the signature of p (resp., r): process q cannot extract properly signed m_p without extracting m_r as well. Why is this necessary? If process p does not do this, there exist the following two executions:

- execution α_4 in which p is faulty and commits an evasion fault with respect to m_p and q is correct and sends m_q;

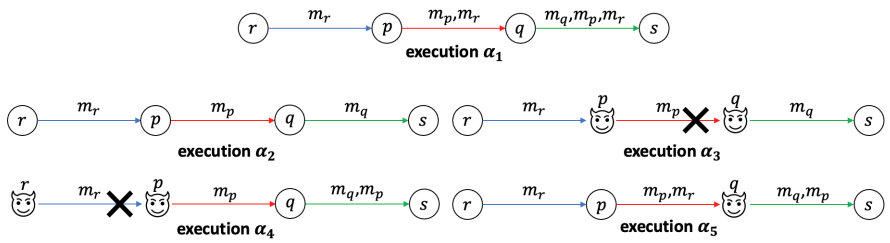

Fig. 1: In execution α_2, process q behaves correctly and sends m_q *without* sending m_p. In execution α_3, process q commits an evasion fault with respect to m_q (note that q cannot send m_p since it has not received m_p). However, these two executions are indistinguishable to process s and, hence, s cannot detect the evasion fault in α_3. In α_4, process p commits an evasion fault with respect to m_p and process q behaves correctly and sends m_q (along with m_p) upon receiving m_p. In α_5, process p is correct and sends m_p along with m_r in a way that *allows* process q to extract only m_p. Furthermore, process q is faulty and it sends m_q along with m_p (but without m_r) to s. Hence, executions α_4 and α_5 are indistinguishable to s and neither p nor q nor r can be detected. Finally, α_1 illustrates an execution in which (1) all processes are correct, (2) process p sends m_p along with m_r in a way that does *not* allow q to extract only m_p, and (3) process q sends m_q along with m_p, m_r to s.

- execution α_5 in which p is correct and sends m_p, q is faulty and sends m_q to s without including m_r.

Now, upon reception of m_q, executions α_4 and α_5 are indistinguishable to s. Moreover, we can create indistinguishable continuations of α_4 and α_5:

- In α_4, we delay messages sent by q and make processes r and p silent.
- In α_5, we delay messages sent by r and p and make q silent.

Hence, process s cannot "safely" detect the evasion fault with respect to m_p in α_4.

Finally, in an execution α_1 in which all processes (r, p, q and s) are correct, process p sends m_p and m_r, and process q sends m_q, m_p and m_r (see Figure 1). Observe that the considered piggybacking technique transforms evasion faults into equivocation faults since there exist messages that can never be sent by a correct process (e.g., a message m_q without message m_p being piggybacked).[8] The presented scenario shows that, in some cases, enabling detection of evasion faults (by transforming them into equivocation faults) could lead to a lengthy chain of piggybacked messages. Fortunately, there exists a simple way to make all evasion faults harmless, thus avoiding any need for their detection; we provide more details in §V.

V. Generic Accountability Transformation τ_{scr}

In this section, we present our generic accountability transformation τ_{scr} that maps any non-synchronous t_0-resilient distributed protocol into its accountable counterpart with factor $f = \mathit{min}(\lceil n/3 \rceil - 1, t_0)$, where n is the total number of processes. First, we provide an intuition behind τ_{scr} (§V-A). Next, we overview τ_{scr} (§V-B) and briefly discuss its implementation (§V-C). Then, we argue that τ_{scr} indeed produces an accountable counterpart of a non-synchronous distributed protocol with factor $f = \mathit{min}(\lceil n/3 \rceil - 1, t_0)$ (§V-D). Lastly, we show that τ_{scr} increases the communication and message complexities by an $O(n^2)$ multiplicative factor, and discuss other applications of τ_{scr} (§V-E).

[8]Formally, the definition of equivocation faults (Definition 5) would need to be expanded by stating that an equivocation also occurs once a process sends a single message that can never be sent if the process was correct.

A. Intuition

Consider a distributed system Ψ with $|\Psi| = n$ processes that execute a distributed protocol $\Pi_{\mathcal{D}}$. Imagine an (unrealistic) oracle θ that belongs to the system and obtains the following responsibilities:

1) *Message relaying:* All communication between processes goes through θ. Specifically, if a process $p \in \Psi$ wants to send a message m to $q \in \Psi$, p sends m to θ which forwards m to q. Moreover, θ is connected with all processes via FIFO communication links.
2) *Correctness verification:* Whenever a process sends a message to θ (to have the message relayed to its recipient), the process accompanies the message with its current behavior (i.e., with the behavior that instructed the sender to send the message). Such construction allows θ to verify the correctness of the sender prior to relaying its message.
   Specifically, θ associates the $\mathit{current}_p$ behavior with each process $p \in \Psi$; initially, $\mathit{current}_p$ is empty, for every process $p \in \Psi$. Once a process p wants to send a message m, it sends (m, β_p) to θ, where β_p is the current behavior of p. When θ receives (m, β_p), it performs the following steps:
   a) It verifies that β_p is valid.
   b) It verifies that $\mathit{current}_p$ is a prefix of β_p.
   c) It verifies that all messages received in β_p were previously relayed by θ in the order of reception specified by β_p.
   d) If all verifications successfully pass, then (1) $\mathit{current}_p \leftarrow \beta_p$, and (2) m is relayed to its recipient. Otherwise, p is ignored forever by θ (and no message sent by p, including m, is ever relayed by θ).

Due to the presented construction, θ "sees", at any point in time, an execution that is *benign*, i.e., an execution in which all processes are either correct or have crashed. Furthermore, every message m relayed by θ has a "fully-correct" causal

past. Lastly, no "product" of an evasion fault is ever relayed by θ, implying that effects of evasion faults are *eliminated*.

The main idea behind our τ_{scr} transformation is to simulate concepts performed by the θ oracle. We explain how that is achieved in the following subsection.

B. Overview

Each process is a hierarchical composition of its four layers (see Figure 2):

1) The state-machine layer: This layer dictates the behavior of the process, i.e., it instructs which messages are sent and which internal events are produced given the received messages and observed internal events.
2) The verification layer: The responsibility of this layer is creating a benign execution of the system (i.e., it simulates the correctness verification responsibility of θ). Specifically, the verification module builds a benign execution out of all secure-delivered messages (see the secure broadcast layer below). Observe that this layer is concerned with *all* processes of the system (whereas the state-machine layer is concerned only with the "host" process). Finally, the verification layer performs a *local* computation, i.e., it fulfills its duty irrespectively of the number of faulty processes.
3) The secure broadcast layer: Every message instructed to be sent by the state-machine layer is secure-broadcast (Definition 29). The secure broadcast primitive ensures that (1) all processes secure-deliver the same set of messages, and (2) secure-deliveries of messages from a single sender are performed in the order the messages were secure-broadcast by the sender. These two properties are guaranteed only if the number of faulty processes does not exceed $\lceil n/3 \rceil - 1$.
4) The network layer: The layer is concerned with network manipulation (i.e., the sending and receiving of messages).

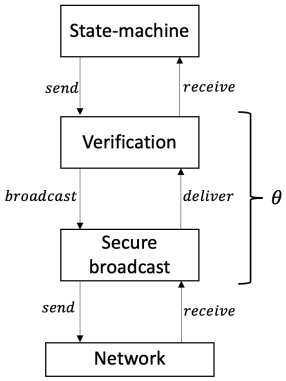

Fig. 2: Overview of the τ_{scr} transformation

We now explain how the presented layers work in harmony to implement our τ_{scr} transformation. Let us focus on a single correct process $p \in \Psi$. Every message m instructed to be sent by the state-machine of p is (1) accompanied by the entire ongoing behavior of p up to the point of sending m (i.e., accompanied by all messages received by p thus far),[9] and (2) secure-broadcast. In this way, p "announces" to all processes what its ongoing behavior is to allow all processes to safely verify the correctness of p. The correctness verification of p by a correct process q carries in the way imposed by θ (see §V-A):

1) It is checked whether the accompanied behavior is indeed correct.
2) It is checked whether previously verified behavior of p is a prefix of the accompanied behavior (this verification passes because of the order-preservation property of secure broadcast and the fact that p is correct).
3) If either of the previous two verifications does not pass, process p is declared as faulty and is ignored by q in the future. In our example, p is correct, implying that it will never be declared as faulty by q.
4) Process q verifies that all messages received by p in the accompanied behavior are "part" of the benign execution built by the verification module of process q.[10] Note that in executions with up to $\lceil n/3 \rceil - 1$ Byzantine processes, since p is correct and the properties of the secure broadcast primitive hold, this condition is eventually satisfied.
5) Once the last condition is fulfilled, the accompanied behavior of p is included in the benign execution built by the verification module of q. Moreover, if $\textit{receiver}(m) = q$, the message m is propagated to the state-machine layer of q to have q react upon the message m.

Observe that the presented verification strategy prevents any evasion fault from affecting a correct process. Indeed, if a faulty process has committed an evasion fault, the first verification step fails, and the process is ignored forever. Lastly, note that all correct processes "see" the *same* benign execution (created by their verification modules) in all non-corrupted executions (i.e., executions with up to $\lceil n/3 \rceil - 1$ Byzantine processes). More precisely, if there are less than $\lceil n/3 \rceil$ faulty processes, the verification modules of all correct processes build the same behavior of every process in the system; note that, formally speaking, observed benign executions (which are sequences of events) can *differ* in the order of events that are not causally related.

In a nutshell, the presented construction of τ_{scr} allows each correct process to act only upon observing a benign execution. Importantly, τ_{scr} *masks* all evasion faults, making them harmless, which was its main design goal. We further explain in §V-D how the presented design enables τ_{src} to produce an accountable counterpart of a non-synchronous distributed protocol.

[9] The transformation implementation (see §V-C) introduces an optimization by piggybacking just a segment of the behavior obtained after the last message was sent, i.e., every received message is piggybacked at most *once*.

[10] Since we assume that all messages are authenticated (see §III-A), a process cannot claim to have received a message if that is not the case.

C. Transformation Implementation

We briefly discuss the simplified implementation of τ_{scr}, given in Algorithm 1. The full implementation is presented in Algorithm 3 (Appendix B).

Algorithm 1 Pseudocode of τ_{scr} - process p

```
1:  upon INIT:
2:      seqNum ← 1
3:      delivered ← ∅                          ▷ secure-delivered messages
4:      validated ← ∅                          ▷ validated messages
5:      next ← [0]^n
6:      receivedMeantime ← []                  ▷ array of messages
7:      current ← [empty]^n                    ▷ behaviors of processes
8:  upon SEND(m):
9:      M ← (m, seqNum, receivedMeantime)_σp
10:     seqNum ← seqNum + 1
11:     receivedMeantime ← []
12:     SECUREBROADCAST(M)
13: upon SECUREDELIVER(M):
14:     delivered ← delivered ∪ {M}
15: ▷ The Validated function is defined in Algorithm 3
16: upon exists (m, sn, recMeantime)_σq ∈ delivered such
    that sn = next[q]+1 ∧ Validated(recMeantime) = true:
17:     if reception of recMeantime after current[q] results
    in a valid behavior β_q that sends m then
18:         validated ← validated ∪ {(m, sn)}
19:         current[q] ← β_q
20:     end if
21: upon exists (m, sn)_σq ∈ validated such that sn =
    next[q] + 1:
22:     next[q] ← sn
23:     if receiver(m) = p then
24:         RECEIVE(m)
25:         receivedMeantime.append(m)
26:     end if
27: upon exists (m, sn)_σq, (m', sn')_σq ∈ delivered such
    that sn = sn' ∧ m ≠ m':
28:     DETECT(q)                              ▷ equivocation
```

The pseudocode captures the implementation details of the verification module, as well as the secure broadcast module. Specifically, the main aim of the pseudocode is to define a sequence of actions taking place once a process is instructed (by its state-machine layer) to send a message. Moreover, we define when the state-machine receives a message from the verification module. Let us take a closer look at Algorithm 1.

Once the state-machine aims to send a message (line 8), the verification module appends to the message (line 9) the following: (1) the sequence number, and (2) all received messages (by the state-machine layer) since the last secure-broadcast message (the *receivedMeantime* variable). Then, the enriched message is disseminated using the secure broadcast primitive (line 12). On the other hand, once the process secure-delivers a message (line 13), it does not propagate the message to the state-machine layer right away (if the message is indeed intended for the process). At this moment, it only includes the message into the *delivered* set (line 14), the set of all secure-delivered messages.

The message is propagated to the state-machine layer only once it belongs to the built benign execution, i.e., only once it belongs to the *validated* set (if a message is included in the *validated* set, the message is *validated* or *valid-delivered*). A message m is validated (i.e., valid-delivered) once (1) all previously sent messages by $\mathit{sender}(m)$ are validated (line 16), (2) all received messages accompanying m are validated (line 16), and (3) it is verified that m is sent in a correct behavior (line 17).

Lastly, as soon as it is observed that a process sends two different messages associated with the same sequence number (line 27), the process is detected (line 28). Since no correct process ever sends two different messages associated with the same sequence number (ensured by line 10), the detected process is indeed faulty. We make a small remark regarding line 27. Namely, the secure broadcast primitive traditionally ensures the "no-duplication" property, i.e., no correct process ever secure-delivers two different messages with the same sequence number (which implies that the condition of line 27 could never be satisfied). However, we assume that the "no-duplication" property is not satisfied by the secure broadcast primitive we use. Note that it is sufficient for a correct process to observe (on any "level") two conflicting messages sent by the same sender. Hence, the condition of line 27 could be satisfied whenever any two conflicting messages are observed (irrespectively of the level to which they "belong").

D. Solution Preservation & Accountability & Syntactic Correspondence

In order to show that our transformation τ_{scr} indeed produces an accountable counterpart of a distributed protocol with factor $f = \mathit{min}(\lceil n/3 \rceil - 1, t_0)$, we need to show that τ_{scr} (1) preserves the solution of a decision task, (2) provides accountability whenever the safety of the decision task is violated, and (3) obtains homomorphism between executions of the transformed and original protocol.

a) Solution preservation: In Appendix E, we define a certain class of transformations producing *pseudo-extensions*. An important feature of the pseudo-extension formalism is that there exists a homomorphism $\mu_e : \mathit{execs}(\bar{\Pi}) \to \mathit{execs}(\Pi)$ between executions of a pseudo-extension $\bar{\Pi}$ and executions of the original protocol Π.

Moreover, we define two distributed properties: *integrity* and *obligation*. The integrity property is satisfied if and only if every received message m has indeed been sent by its appearing source (formally defined in Definition 22); note that the integrity property trivially follows from the non-forgeability property of digital signatures. The obligation property is satisfied if and only if correct processes are able to communicate between each other, i.e., if and only if every

message sent by a correct process to a correct process is eventually received (formally defined in Definition 23).

Let $\Pi_{\mathcal{D}}$ be a distributed protocol that solves a decision task $\mathcal{D}$ with t_0-resiliency and let $\bar{\Pi}_{\mathcal{D}}$ be a pseudo-extension of $\Pi_{\mathcal{D}}$. We prove in Lemma 18 that if $\bar{\Pi}_{\mathcal{D}}$ satisfies both integrity and obligation in an execution, then $\bar{\Pi}_{\mathcal{D}}$ "solves" $\mathcal{D}$ in that execution. Given the fact that τ_{scr} produces pseudo-extensions (Lemma 19), the fact that our transformation ensures the obligation property whenever the number of faulty processes is less than or equal to $\lceil n/3 \rceil - 1$ and that our transformation ensures the integrity property regardless of the number of faulty processes, it follows that $\bar{\Pi}_{\mathcal{D}}$ solves $\mathcal{D}$ with $\mathit{min}(\lceil n/3 \rceil - 1, t_0)$-resiliency.

b) Accountability: Recall that the verification module works correctly irrespectively of the number of faulty processes. Moreover, the integrity property is ensured even in an entirely corrupted system. Hence, if correct processes output values that cause a safety violation, then at least $t_0 + 1$ pairs of conflicting messages could be observed from as many Byzantine processes (Lemma 27), where t_0 is the resiliency of the original distributed protocol.

The previous statement comes as no surprise. Indeed, every correct process p that outputs a value leading to a safety violation has observed a benign execution α^p (via its verification module); note that the α^p execution "instructs" p to output its value. If safety is violated and no more than t_0 pairs of conflicting messages are sent, it would be possible to devise an execution where t_0 faulty processes violate safety by interacting with each correct process p, which outputs a value leading to the safety violation, *exactly* as they do in α^p. Hence, we reach a contradiction with the fact that the distributed protocol solves its decision task with t_0-resiliency.

c) Syntactic correspondence: Lastly, we show that our transformation τ_{scr} preserves the "way" the original protocol solves the problem. Specifically, a distributed protocol $\tau_{scr}(\Pi_{\mathcal{D}})$ solves a decision task $\mathcal{D}$ (ensured because of the solution preservation) in the same way as $\Pi_{\mathcal{D}}$.

Formally, $\tau_{scr}(\Pi_{\mathcal{D}})$ (which is a pseudo-extension of $\Pi_{\mathcal{D}}$), preserves all the fully-correct FIFO executions of $\Pi_{\mathcal{D}}$, i.e., for every FIFO execution α of $\Pi_{\mathcal{D}}$, where $\mathit{Corr}_{\Pi}(\alpha) = \Psi$, there exists an execution $\bar{\alpha}$ of $\tau_{scr}(\Pi_{\mathcal{D}})$ such that $\alpha = \mu_e(\bar{\alpha})$. Intuitively, an execution is a FIFO execution if all messages are received in the order in which they were sent (FIFO executions are formally defined in Definition 36).

Theorem 3. *Let $\Pi_{\mathcal{D}}$ be a non-synchronous distributed protocol that solves a decision task $\mathcal{D}$ with t_0-resiliency. Then, $\tau_{scr}(\Pi_{\mathcal{D}})$ is an accountable counterpart of $\Pi_{\mathcal{D}}$ with factor $f = \mathit{min}(\lceil n/3 \rceil - 1, t_0)$ according to a basis which consists of all fully-correct FIFO executions of $\Pi_{\mathcal{D}}$.*

Proof. Theorem 7 proves that τ_{scr} achieves accountability and solution preservation. The proof that τ_{scr} allows for syntactic correspondence requires additional formalism presented in appendices E and F. □

We conclude this subsection by stating that τ_{scr} could be generalized to allow accountability even in synchronous environments or partially synchronous environments in which message delays after GST are bounded by a known parameter. In such scenarios, the only modification to our τ_{scr} transformation is increasing every timeout duration at processes by 3 times in order to accommodate for the increase in message delays introduced by the secure broadcast primitive.

E. Complexity

Lastly, we present the complexity overhead of τ_{scr}.

Theorem 4. *Let $\Pi_{\mathcal{D}}$ be a non-synchronous distributed protocol that solves a decision task $\mathcal{D}$ with t_0-resiliency and let $\bar{\Pi}_{\mathcal{D}} = \tau_{scr}(\Pi_{\mathcal{D}})$. Let $\bar{\alpha}$ be an execution of $\bar{\Pi}_{\mathcal{D}}$ and let $\alpha = \mu_e(\bar{\alpha})$. The following holds:*

- *Communication complexity: Let cc' and cc be the communication complexities of $\bar{\alpha}$ and α, respectively. Then, $\mathsf{cc}' = \mathsf{cc} \cdot O(n^2) \cdot \kappa$, where κ is the security number.*
- *Message complexity: Let mc' and mc be the message complexities of $\bar{\alpha}$ and α, respectively. Then, $\mathsf{mc}' = \mathsf{mc} \cdot O(n^2) \cdot \kappa$, where κ is the security number.*
- *Memory complexity: Let memc' and memc be the memory complexities of $\bar{\alpha}$ and α, respectively. Then, $\mathsf{memc}' = \mathsf{memc} + n \cdot cc \cdot \kappa$, where κ is the security number.*
- *Delay complexity: Let dc' and dc be the delay complexities of $\bar{\alpha}$ and α, respectively. Then, $\mathsf{dc}' = 3 \cdot \mathsf{dc}$.*

Proof. We prove the theorem by using the well-known "double-echo" secure broadcast [7], which implies a quadratic overhead per message and tripling of message delays. □

Remark 1. The broader application of τ_{scr} includes:

- Distributed protocols in which violation of *any* safety property triggers accountability as long as lack of privacy is acceptable (formal treatment given in Appendix D).
- Randomized distributed protocols in which (1) safety is ensured deterministically, and (2) private channels are not required for liveness (formal treatment given in Appendix G).
- Sub-quadratic committee-based blockchains (formal treatment given in Appendix H).

VI. Conclusion

We presented a transformation of any non-synchronous distributed protocol into an accountable distributed protocol that remains practical. The main idea behind our transformation is to allow only benign executions to reach the state-machine layer of correct processes, following the ideas previously presented in [3], [7], [15], [17], [25], [26]. Future work includes designing accountable distributed protocols that lower the $O(n^2)$ multiplicative communication overhead of our generic τ_{scr} transformation by focusing on specific distributed protocols.

References

[1] Alpern, B., and Schneider, F. B. Defining Liveness. *Inf. Process. Lett. 21*, 4 (1985), 181–185.

[2] Attiya, H., Herlihy, M., and Rachman, O. Efficient Atomic Snapshots Using Lattice Agreement. In *Distributed Algorithms* (1992), A. Segall and S. Zaks, Eds., Springer Berlin Heidelberg, pp. 35–53.

[3] Attiya, H., and Welch, J. L. *Distributed computing - Fundamentals, Simulations, and Advanced Topics (2. ed.).* Wiley series on parallel and distributed computing. Wiley, 2004.

[4] Bazzi, R. A. Automatically increasing fault tolerance in distributed systems. G. I. of Technology School of Information and G. U. S. Computer Science Atlanta, Eds.

[5] Bazzi, R. A., and Neiger, G. Optimally Simulating Crash Failures in a Byzantine Environment. In *Distributed Algorithms, 5th International Workshop, WDAG '91, Delphi, Greece, October 7-9, 1991, Proceedings* (1991), S. Toueg, P. G. Spirakis, and L. M. Kirousis, Eds., vol. 579 of *Lecture Notes in Computer Science*, Springer, pp. 108–128.

[6] 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.

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

[8] Buterin, V., and Griffith, V. Casper the Friendly Finality Gadget. *arXiv preprint arXiv:1710.09437* (2017).

[9] Canetti, R., and Rabin, T. Fast asynchronous Byzantine agreement with optimal resilience. 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. 42–51.

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

[11] Chaudhuri, S. More Choices Allow More Faults: Set Consensus Problems In Totally Asynchronous Systems. *Information and Computation 105*, 1 (1993), 132–158.

[12] Civit, P., Gilbert, S., and Gramoli, V. Brief Announcement: Polygraph: Accountable Byzantine Agreement. In *Proceedings of the 34th International Symposium on Distributed Computing (DISC'20)* (Oct 2020), Schloss Dagstuhl, pp. 45:1–45:3.

[13] 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).

[14] Civit, P., Gilbert, S., Gramoli, V., Guerraoui, R., and Komatovic, J. As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy! In *36th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30-June 3, 2022 (to appear)* (2022), IEEE.

[15] Clement, A., Junqueira, F., Kate, A., and Rodrigues, R. On the (Limited) Power of Non-Equivocation. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012* (2012), pp. 301–308.

[16] Clement, A., Junqueira, F., Kate, A., and Rodrigues, R. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2012), PODC '12, Association for Computing Machinery, p. 301–308.

[17] Coan, B. A. A Compiler that Increases the Fault Tolerance of Asynchronous Protocols. *IEEE Trans. Computers 37*, 12 (1988), 1541–1553.

[18] 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.

[19] 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* (1998), p. 315.

[20] 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).

[21] Guerraoui, R., Komatovic, J., Kuznetsov, P., Pignolet, Y., Seredinschi, D., and Tonkikh, A. Dynamic Byzantine Reliable Broadcast. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)* (2020), Q. Bramas, R. Oshman, and P. Romano, Eds., vol. 184 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 23:1–23:18.

[22] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., and Seredinschi, D. Scalable Byzantine Reliable Broadcast. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary* (2019), J. Suomela, Ed., vol. 146 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 22:1–22:16.

[23] Haeberlen, A., Kouznetsov, P., and Druschel, P. PeerReview: Practical Accountability for Distributed Systems. *SOSP'07* (2007).

[24] Haeberlen, A., and Kuznetsov, P. The Fault Detection Problem. In *Principles of Distributed Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings* (2009), T. F. Abdelzaher, M. Raynal, and N. Santoro, Eds., vol. 5923 of *Lecture Notes in Computer Science*, Springer, pp. 99–114.

[25] Ho, C., Dolev, D., and Van Renesse, R. Making Distributed Applications Robust. In *International Conference On Principles Of Distributed Systems* (2007), Springer, pp. 232–246.

[26] Ho, C., Van Renesse, R., Bickford, M., and Dolev, D. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *NSDI* (2008), vol. 8, pp. 175–188.

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

[28] King, V., Saia, J., Sanwalani, V., and Vee, E. Scalable Leader Election. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006* (2006), ACM Press, pp. 990–999.

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

[30] 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.

[31] Malkhi, D., and Reiter, M. Unreliable Intrusion Detection In Distributed Computations. In *Proceedings 10th Computer Security Foundations Workshop* (1997), IEEE, pp. 116–124.

[32] Neiger, G., and Toueg, S. Automatically Increasing the Fault-Tolerance of Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988* (1988), D. Dolev, Ed., ACM, pp. 248–262.

[33] Pass, R., and Shi, E. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria* (2017), A. W. Richa, Ed., vol. 91 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 39:1–39:16.

[34] Sheng, P., Wang, G., Nayak, K., Kannan, S., and Viswanath, P. BFT Protocol Forensics. In *Computer and Communication Security (CCS)* (Nov 2021).

[35] Szpilrajn, E. Sur l'extension de l'ordre partiel. *Fundamenta mathematicae 1*, 16 (1930), 386–389.

[36] Yin, M., Malkhi, D., Reiter, M. K., Golan-Gueta, 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.

[37] Zheng, X., and Garg, V. K. Byzantine Lattice Agreement in Asynchronous Systems. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)* (2020), Q. Bramas, R. Oshman, and P. Romano, Eds., vol. 184 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 4:1–4:16.

Appendix

We separate the appendix into two parts. The first part formally proves the results presented in the paper, whereas the second part further generalizes our τ_{scr} transformation.

a) Part One: In Appendix A, we formally prove that accountability in non-synchronous environments implies detection of commission faults (formal proofs of theorems 1 and 2 are included in Appendix A). Secondly, Appendix B contains proof of correctness of τ_{scr}: it proves the solution preservation and the accountability properties introduced in Definition 3 (the proof of syntactic correspondence is delegated to Appendix F).

b) Part Two: We start by giving the formal definition of a homomorphic transformation in Appendix C. Next, we generalize our τ_{scr} transformation to allow for accountability whenever any safety distributed property is violated (Appendix D). Appendix E introduces transformations mapping distributed protocols into pseudo-extensions of those protocols. We formally present a slightly modified version of our transformation τ_{scr} in Appendix F: the modified version of τ_{scr} allows for some optimizations that the original transformation does not consider. Furthermore, we prove the syntactic correspondence property in Appendix F. Next, we present how τ_{scr} can be applied to randomized distributed protocols in Appendix G. Additionally, Appendix H shows how τ_{scr} can be used in permissionless distributed protocols. Finally, Appendix I illustrates how (the modified version of) our transformation is applied to PBFT [10].

Appendix A
Accountability Implies Detection of Commission Faults

We devote this section to proving that accountability in a non-synchronous setting demands the ability to detect commission faults. Specifically, we prove that (1) irrevocable detections must be based on detected commission faults (otherwise, a correct process could be irrevocably detected; see Theorem 1), and (2) whenever safety is violated, "enough" processes have committed commission faults (see Theorem 2). Throughout the entire section, we fix a decision task $\mathcal{D}$ and a non-synchronous distributed protocol $\Pi_{\mathcal{D}} = (\Pi_p, \Pi_q, ..., \Pi_z)$, where $\Psi = \{p, q, ..., z\}$, that solves $\mathcal{D}$ with t_0-resiliency.

A. Assumptions

We start by restating all the assumptions we pose about $\Pi_{\mathcal{D}} = (\Pi_p, \Pi_q, ..., \Pi_z)$. A protocol Π_p, assigned to a process $p \in \Psi$, does not send the same message more than once. Moreover, each message sent by Π_p is properly authenticated and any incoming duplicate messages or messages that cannot be authenticated are ignored. We assume that Π_p does not reveal the key material, i.e., if a message is signed by a process p and p is correct, then p must have indeed sent the message. Processes can forward messages to other processes and they can include messages in other messages they send, and we assume that an included or forwarded message can still be authenticated. Moreover, Assumption 1, which states that each message maps into exactly one finite and valid behavior that ends with that message being sent, stands. Only finitely many messages are exchanged in every finite execution of $\Pi_{\mathcal{D}}$. Lastly, each message m has a unique sender $\mathit{sender}(m) \in \Psi$ and a unique receiver $\mathit{receiver}(m) \in \Psi$.

B. If a Process Does Not Commit Commission Faults, It Can Be Correct

This subsection proves the main intermediate result of this section - a faulty process which does not commit any commission fault could be "replaced" by a correct process such that no originally correct process could observe any difference. First, we restate the definition of an execution (see §III-A).

Definition 8 (Execution)**.** Execution α is a sequence of events such that:
1) for every message m received in α, message m is sent in α prior to the reception, and
2) if α is infinite, then every sent message is received.

Observe that a finite execution can contain messages that are sent and not yet received. Moreover, recall that a message can be sent multiple times by faulty processes. Hence, we demand Definition 8 to account for this. Specifically:
- for every instance of a message m received in an execution α, there exists a corresponding instance sent in α prior to the reception, and
- if α is infinite, then every sent instance of a message is received.

Next, we define the *causality* set of messages given a specific execution α of $\Pi_{\mathcal{D}}$. Formally, let α be any execution of $\Pi_{\mathcal{D}}$ in which no duplicate or non-authenticated messages are sent (thus, no duplicate or non-authenticated messages are received). Moreover, let $C = \mathit{Corr}_{\Pi_{\mathcal{D}}}(\alpha)$, $F = \{f \mid f \text{ is faulty in } \alpha, \text{ but it does not commit any commission fault in } \alpha\}$ and $B = \Psi \setminus (C \cup F)$.[11] For every message m sent in α, we define the $\mathit{causality}(m, \alpha)$ set in the following manner:
1) Let $\mathit{causality}(m, \alpha) \leftarrow \emptyset$ and $\mathit{newlyAdded} \leftarrow \{m\}$.
2) Repeat until $\mathit{newlyAdded} = \emptyset$.

[11]Each process that belongs to B is faulty and commits commission faults in α.

a) Let $m' \in \mathit{newlyAdded}$.
b) If $\mathit{sender}(m') \in C \cup F$, then $\mathit{newCausality}(m, \alpha) \leftarrow \mathit{causality}(m, \alpha) \cup \mathcal{J}_{m'}$.
Otherwise, $\mathit{newCausality}(m, \alpha) \leftarrow \mathit{causality}(m, \alpha) \cup \mathit{received}(\beta_{m'})$, where $\beta_{m'}$ is the behavior of $\mathit{sender}(m')$ in α that ends with message m' being sent (i.e., $\beta_{m'} = ...(\mathit{sender}(m'), I, O)$, where $m' \in O$).
c) $\mathit{newlyAdded} \leftarrow \mathit{newlyAdded} \setminus \{m'\}$.
d) $\mathit{newlyAdded} \leftarrow \mathit{newlyAdded} \cup (\mathit{newCausality}(m, \alpha) \setminus \mathit{causality}(m, \alpha))$.
e) $\mathit{causality}(m, \alpha) \leftarrow \mathit{newCausality}(m, \alpha)$.

Note that every message $m' \in \mathit{causality}(m, \alpha)$, where m is a message sent in α, is sent in α.

Now, we prove that the $\mathit{causality}(m, \alpha)$ set defines a strict partial order relation. Again, let α be any execution of $\Pi_{\mathcal{D}}$ in which no duplicate or non-authenticated messages are sent (thus, no duplicate or non-authenticated messages are received). Moreover, let $C = \mathit{Corr}_{\Pi_{\mathcal{D}}}(\alpha)$, $F = \{f \mid f \text{ is faulty in } \alpha, \text{ but it does not commit any commission fault in } \alpha\}$ and $B = \Psi \setminus (C \cup F)$. Finally, let sent_α denote the set of messages sent in α. For any two messages $m_1, m_2 \in \mathit{sent}_\alpha$, $m_1 \overset{\alpha}{\prec} m_2$ if and only if $m_1 \in \mathit{causality}(m_2, \alpha)$. Next, we prove that the "$\overset{\alpha}{\prec}$" relation is a strict partial order relation. To this end, we prove that the relation is irreflexive and transitive:

- The relation is irreflexive: By contradiction, suppose that there exists a message $m \in \mathit{sent}_\alpha$ such that $m \overset{\alpha}{\prec} m$. Hence, there exists a chain of messages $m, m_1, m_2, ..., m_x, m$ such that m is sent before m_1 in α, m_1 is sent before m_2 in α, ..., m_x is sent before m in α. Therefore, m is sent before m in α. Given that duplicate messages are not sent in α, this is impossible and the relation is irreflexive.
- The relation is transitive: Follows from the definition of the $\mathit{causality}(m, \alpha)$ set.

Next, we show that faulty processes which have not committed any commission fault can be "replaced" by correct processes without any originally correct process observing the difference. This result allows us to obtain proofs of theorems 1 and 2.

Lemma 1. Let α be a finite execution of $\Pi_{\mathcal{D}}$ in which no duplicate or non-authenticated messages are sent (thus, no duplicate or non-authenticated messages are received). Moreover, let (C, F, B) be a partition of Ψ such that $C = \mathit{Corr}_{\Pi_{\mathcal{D}}}(\alpha)$, $F = \{f \mid f \text{ is faulty in } \alpha, \text{ but it does not commit any commission fault in } \alpha\}$ and $B = \Psi \setminus (C \cup F)$.

There exists a finite execution α' of $\Pi_{\mathcal{D}}$ such that:

- $\alpha'|_c = \alpha|_c$, for every process $c \in C$, and
- $\alpha'|_f$ is a valid behavior of f according to Π_f, for every process $f \in F$.

Proof. First, we associate every process $c \in C$ with $\beta_c = \alpha|_c$. Similarly, every process $b \in B$ is associated with $\beta_b = \alpha|_b$. Lastly, every process $f \in F$ is associated with β_f, where β_f is the shortest valid behavior of f such that all messages sent by f in α are sent in β_f (observe that such behavior of f exists due to the fact that f does not commit any commission fault in α); recall that f sends only finitely many messages in α.

Now, let $M_{sent,\alpha} = \{m \mid m \text{ is sent in } \alpha\}$. Let $\mathit{received} = \bigcup_{p \in \Psi} \mathit{received}(\beta_p)$ and let $\mathit{sent} = \bigcup_{p \in \Psi} \mathit{sent}(\beta_p)$. First, $\mathit{received} \subseteq M_{sent,\alpha}$. Furthermore, $M_{sent,\alpha} \subseteq \mathit{sent}$. Therefore, $\mathit{received} \subseteq \mathit{sent}$.

Let $\mathcal{E} = \bigcup_{p \in \Psi} \beta_p$ denote the union of all events in the aforementioned behaviors.[12] Note that for an event $e \in \mathcal{E}$ that receives a message m there exists an event $e' \in \mathcal{E}$ that sends m (since $\mathit{received} \subseteq \mathit{sent}$). For any two events $e_1 = (p_1, I_1, O_1), e_2 = (p_2, I_2, O_2)$ that belong to $\mathcal{E}$, $e_1 \mathbin{@} e_2$ if and only if:

1) $p_1 = p_2$ and e_1 precedes e_2 in β_{p_1}, or
2) there exists a message $m \in O_1 \cap I_2$, or
3) there exists an event $e_3 \in \mathcal{E}$ such that $e_1 \mathbin{@} e_3$ and $e_3 \mathbin{@} e_2$.

Note that if $m \in O_1 \cap I_2$ and $p_1 = p_2$, then e_1 precedes e_2 in β_{p_1} (otherwise, β_{p_1} is not a behavior since it receives a message which is not previously sent).[13] In other words, if $e_1 \mathbin{@} e_2$ by the second criterion and $p_1 = p_2$, then e_1 precedes e_2 in β_{p_1} (i.e., $e_1 \mathbin{@} e_2$ by the first criterion).

We now prove that the "@" relation is a strict partial order relation. Again, we prove that the relation is irreflexive and transitive:

- The relation is irreflexive: By contradiction, suppose that $e \mathbin{@} e$, for some event $e \in \mathcal{E}$. Let $e = (p, I, O)$. Since $e \mathbin{@} e$, there exists a (potentially empty) chain of events $e_1, e_2, ..., e_x$ such that $e \mathbin{@} e_1 \mathbin{@} e_2 \mathbin{@} ... \mathbin{@} e_x \mathbin{@} e$, where every two adjacent events are related due to the first or the second criterion. We distinguish two possibilities:
  - All events occur on the same process: This is trivially impossible.
  - All events do not occur on the same process: We separate continuous fragments of the $e \mathbin{@} e_1 \mathbin{@} e_2 \mathbin{@} e_3 \mathbin{@} ... \mathbin{@} e_x \mathbin{@} e$ chain that occur on the same process; let there be $r \leq x + 2$ fragments frg_1, frg_2, ..., frg_r. Observe that frg_1 and frg_r occur on process p. Moreover, each fragment (except frg_r) ends with a message being sent which is

[12] We slightly abuse the notation by treating sequences of events as sets of events.
[13] Recall that no duplicate messages are sent or received in β_p, for every process $p \in \Psi$.

received at the start of the next fragment; let a message m_i be sent at the end of frg_i and received at the start of frg_{i+1}, for every $i \in [1, r-1]$.

First, note that $m_i \in M_{sent,\alpha}$, for every $i \in [1, r-1]$. Let frg_{i+1} occur on a process q, for $i \in [1, r-1]$. We consider three possibilities:

* Let $q \in B$: In this case, m_i is received in α, which means that $m_i \in M_{sent,\alpha}$.
* Let $q \in C$: Again, m_i is received in α, which implies that $m_i \in M_{sent,\alpha}$.
* Let $q \in F$: Due to the construction of β_q, $m_i \in \mathcal{J}_{m'}$, where m' is a message sent by q in α. Therefore, m_i is received by q in α, which means that $m_i \in M_{sent,\alpha}$.

We prove that $m_i \overset{\alpha}{\prec} m_{i+1}$, for any $i \in [1, r-2]$. Let $\mathit{sender}(m_{i+1}) = q$. We separate two possibilities:

* Let $q \in B$: In this case, m_i is received before m_{i+1} is sent in α. Hence, $m_i \overset{\alpha}{\prec} m_{i+1}$.
* Let $q \in C \cup F$: In this case, $m_i \in \mathcal{J}_{m_{i+1}}$. Hence, $m_i \overset{\alpha}{\prec} m_{i+1}$.

Finally, we prove that $m_{r-1} \overset{\alpha}{\prec} m_1$. Recall that $\mathit{sender}(m_1) = p$. Again, we consider two possibilities:

* Let $p \in B$: In this case, m_{r-1} is received before m_1 is sent in α. Hence, $m_{r-1} \overset{\alpha}{\prec} m_1$.
* Let $p \in C \cup F$: In this case, $m_{r-1} \in \mathcal{J}_{m_1}$. Hence, $m_{r-1} \overset{\alpha}{\prec} m_1$.

Due to the transitivity property of the "$\overset{\alpha}{\prec}$" relation, we have that $m_1 \overset{\alpha}{\prec} m_1$, which violates the irreflexivity property of the "$\overset{\alpha}{\prec}$" relation. Therefore, the "@" relation is irreflexive.

- The relation is transitive: Follows from the third criterion.

Indeed, the "@" is a strict partial order relation, which implies the existence of a linear extension of the relation [35]. Hence, α' is any linear extension of the "@" relation: due to the "@" relation and the fact that $|\mathcal{E}| < \infty$, α' is a finite execution which, due to its construction, satisfies the claims of the lemma. □

C. Detections Not Based on Commission Faults Can Be Wrong

This subsection proves that an irrevocable detection made by a correct process must be based on a detected commission fault. We start by defining what it means for a correct process to detect a commission fault committed by another process.

Definition 9 (Detection of Commission Faults)**.** We say that a process p *detects a commission fault* of a process q in a behavior β_p which is valid according to Π_p if and only if there does not exist an execution α of $\Pi_{\mathcal{D}}$ such that (1) $\beta_p = \alpha|_p$, and (2) q does not commit any commission fault in α.

A process detects a commission fault according to Definition 9 if it is aware that, given its current behavior, the detected process must have committed a commission fault. Note that Definition 9 does not require for a proof of a committed commission fault to be obtained. In other words, a correct process p might just "locally" detect a commission fault committed by a process q without being able to prove to any other process that q has committed commission faults; for example, if a process p receives a message m', where $\mathit{sender}(m') = q$, without having previously sent a message m with $m \in \mathcal{J}_{m'}$, process p detects the evasion fault with respect to m' committed by q *without* being able to prove this evasion fault to any other process.

Next, we say that a process p *conclusively detects* (i.e., irrevocably detects) a process q if and only if p produces the $\mathit{detect}(q)$ internal event. We require that correct processes *never* conclusively detect correct processes. More formally, we require *detection accuracy* to hold.

Definition 10 (Detection Accuracy)**.** Let α be an execution of $\Pi_{\mathcal{D}}$ and let a correct process p conclusively detect a process q in α. Then, q is faulty in α.

Finally, we prove that a correct process which conclusively detects a faulty process without detecting commission faults can make a "mistake", i.e., it can violate detection accuracy.

Theorem 5. *Let $\Pi_{\mathcal{D}}$ be a non-synchronous distributed protocol.*[14] *Let α be an execution of $\Pi_{\mathcal{D}}$ such that (1) a correct process p conclusively detects a faulty process q in α, and (2) p does not detect a commission fault of q in $\alpha|_p$. Then, there exists an execution α' of $\Pi_{\mathcal{D}}$ such that (1) p conclusively detects q, and (2) both p and q are correct in α', i.e., detection accuracy is violated in α'.*

Proof. Let β_p be the prefix of $\alpha|_p$ that ends with p conclusively detecting q. Since p does not detect a commission fault of q in $\alpha|_p$, there exists (by Definition 9) a finite execution α'' of $\Pi_{\mathcal{D}}$ such that (1) $\beta_p = \alpha''|_p$, and (2) q does not commit any commission fault in α''. Since p conclusively detects q in α'', q is faulty in α'' (by detection accuracy).

First, we remove (1) all duplicate sending and receiving events from α'', and (2) all sent non-authenticated messages from α'' (i.e., if a message m is sent or received multiple times in α'', we "keep" just the first such action and remove all other;

[14]Note that $\Pi_{\mathcal{D}}$ does not need to solve a decision task, i.e., it can be *any* non-synchronous distributed protocol.

if a non-authenticated message m is sent or received, the action is removed).[15] Note that the newly obtained α'' is still well-formed, p conclusively detects q in α'' (since correct processes ignore duplicate and non-authenticated messages; see §III-A) and q does not commit any commission fault in α''.

According to Lemma 1, there exists an execution α' such that, for every process $c \in \mathit{Corr}_{\Pi_\mathcal{D}}(\alpha'')$, $\alpha'|_c = \alpha''|_c$, and q is correct in α' (since q does not commit any commission fault in α''). Therefore, p conclusively detects correct process q in α', which means that the detection accuracy property is violated in α'. The theorem holds. □

D. Commission Faults Are Necessary to Violate Safety

This subsection proves that commission faults are necessary to violate safety in non-synchronous distributed protocols. Specifically, whenever a safety of $\mathcal{D}$ is violated by $\Pi_\mathcal{D}$, at least $t_0 + 1$ processes have committed commission faults; recall that $\Pi_\mathcal{D}$ solves $\mathcal{D}$ with t_0-resiliency.

Theorem 6. *Let $\Pi_\mathcal{D}$ be a non-synchronous distributed protocol that solves a decision task $\mathcal{D}$ with t_0-resiliency. Let α be an execution of $\Pi_\mathcal{D}$ in which $\Pi_\mathcal{D}$ violates safety of $\mathcal{D}$. At least $t_0 + 1$ distinct processes commit commission faults in α.*

Proof. We prove the theorem by contradiction. Specifically, we assume that there exists an execution α of $\Pi_\mathcal{D}$ such that (1) $\Pi_\mathcal{D}$ violates safety of $\mathcal{D}$ in α, and (2) the number of distinct processes that commit commission faults in α is less than or equal to t_0.

Since $\Pi_\mathcal{D}$ violates safety of $\mathcal{D}$ in α, there exists a finite prefix of α denoted by α_i such that (1) C is not safe-extendable according to $\mathcal{D}$, where $C = \{(p, O_p \neq \bot), (q, O_q \neq \bot), ..., (z, O_z \neq \bot)\}$, and (2) a correct process p outputs $O_p \neq \bot$ in α_i, a correct process q outputs $O_q \neq \bot$ in α_i, ..., a correct process z outputs $O_z \neq \bot$ in α_i.

We aim to devise an execution α' of $\Pi_\mathcal{D}$ such that (1) no process $p \in \mathit{Corr}_{\Pi_\mathcal{D}}(\alpha_i)$ distinguishes α_i from α' (hence, safety of $\mathcal{D}$ is violated in α'), and (2) the number of faulty processes in α' is less than or equal to t_0. If we can devise such an execution, we will reach a contradiction with the fact that $\Pi_\mathcal{D}$ solves $\mathcal{D}$ with t_0-resiliency.

First, we remove (1) all duplicate sending and receiving events from α_i, and (2) all sent non-authenticated messages from α_i (i.e., if a message m is sent or received multiple times in α_i, we "keep" just the first such action and remove all other; if a non-authenticated message m is sent or received, the action is removed). Note that the newly obtained α_i is still well-formed and it violates safety (since correct processes ignore duplicate and non-authenticated messages; see §III-A). Moreover, the number of processes committing commission faults stays the same.

We denote by B_1 the set of faulty processes that have committed commission faults in α_i, where $|B_1| \leq t_0$ (by the assumption). Moreover, we denote by B_2 the set of faulty processes that have not committed any commission fault in α_i, where $|B_2| \geq 1$ (since $\Pi_\mathcal{D}$ solves $\mathcal{D}$ with t_0-resiliency). According to Lemma 1, there exists an execution α' such that, for every process $c \in \mathit{Corr}_{\Pi_\mathcal{D}}(\alpha_i)$, $\alpha'|_c = \alpha_i|_c$, and all processes from the B_2 set are correct in α'. Hence, α' violates safety of $\mathcal{D}$ with less than or equal to t_0 faulty processes (up to $|B_1| \leq t_0$ processes are faulty in α'). This is a contradiction with the fact that $\Pi_\mathcal{D}$ solves $\mathcal{D}$ with t_0-resiliency. Hence, the theorem holds. □

In a nutshell, Theorem 5 shows that all conclusive (i.e., irrevocable) detections made by correct processes must be based on committed commission faults. Fortunately, Theorem 6 shows that $t_0 + 1$ processes commit commission faults whenever safety of a decision task is violated by a distributed protocol that solves the task with t_0-resiliency. Therefore, accountability is *possible:* every distributed protocol solving a decision task can be made accountable.

Appendix B
Proof of correctness of τ_{scr}

In this section we prove the correctness of the transformation τ_{scr} briefly described in algorithm 1 in section V. A more detailed version can be found in algorithms 3 and 2. The transformation τ_{scr} implicitly define a causal relationship between validated messages. We start by making it explicit in definition 11.

Definition 11 (referencing secure-delivered messages)**.** Let Π be a distributed protocol with $\mathcal{M}$ denoting the attached set of messages and $\bar{\Pi} = \tau_{scr}(\Pi)$. Let $\bar{\alpha} \in execs(\bar{\Pi})$. We note $\tilde{M}^{sd}(\bar{\alpha})$ (resp. $M^{sd}(\bar{\alpha})$) the set of messages $\tilde{M}$ (resp. the set of pairs $(m, s) \in \mathcal{M} \times \mathsf{N}$) s. t. $\exists p \in Corr_{\bar{\Pi}}(\bar{\alpha})$ that secure-delivered $\tilde{M} = ((M', s), \underline{M})_{\sigma_q}$ (resp. that secure-delivered $\tilde{M} = ((M', s), \underline{M})_{\sigma_q}$ with $m \in M'$ and $q = sender(m)$).

Let $\tilde{M}^1 = ((M'_1, s_1), \underline{M}_1)_{\sigma_q}$, $\tilde{M}^2 = ((M'_2, s_2), \underline{M}_2)_{\sigma_p} \in \tilde{M}^{sc}(\bar{\alpha})$.

We say that $\tilde{M}_2$ *directly refers to* $\tilde{M}_1$ if

- $q = p$ and $s_1 = s_2 + 1$
- $\exists m_1 \in M'_1$ with $(m_1, s_1) \in \underline{M}^2$ ($\underline{M}^2$ is seen as a set of pairs (m, s) of $\mathcal{M} \times \mathsf{N}$ with a slight abuse of notation.)

[15] We assume that duplicate or non-authenticated messages are not necessary for a faulty process to be able to send some message. In other words, if a faulty process can send a message m in a behavior in which it receives duplicate or non-authenticated messages, then m can be sent in a behavior without duplicate (i.e., with just a single instance of each received message) and non-authenticated messages.

Algorithm 2 Helper algorithm for verification layer of τ_{scr} - process p

1: **function** ValidBehaviour($\zeta_q^0, [M^1, ..., M^\ell], M^*$)
2: $q \leftarrow sender(M^*)$
3: β_q is initialized to the empty sequence
4: **for** $i \in [1, \ell - 1]$ **do**
5: $(\zeta_q^i, M^\phi) = \mathcal{T}_q(\zeta_p^{i-1}, M^i)$
6: $\beta_q \leftarrow \beta_q || (q, M^i, \emptyset)$ ▷ $||$ is the concatenation operator
7: **if** $M^\phi \neq \emptyset$ **then**
8: **return** $(\perp, \perp)$ ▷ Not Valid
9: **end if**
10: **end for**
11: $(\zeta_q^\ell, M^{tosend}) = \mathcal{T}_q(\zeta_q^{\ell-1}, M^\ell)$ $\beta_q \leftarrow \beta_q || (q, M^i, M^{tosend})$
12: **if** $M^{tosend} \neq M^*$ **then**
13: **return** $(\perp, \perp)$ ▷ Not Valid
14: **else**
15: **return** (ζ_q^ℓ, β_q) ▷ Valid
16: **end if**
17:
18: **function** ValidBehaviour'($\zeta_q, [\{(m_i^1, sn_i^1)\}_{i \in I^1}, ..., \{(m_i^\ell, sn_i^\ell)\}_{i \in I^\ell}], M^*$)
19: **return** *ValidBehaviour* $(\zeta_q^0, [\{m_i^1\}_{i \in I^1}, ..., \{m_i^\ell\}_{i \in I^\ell}], M^*)$
20:
21: **function** IsValid($\underline{M}, validated$) ▷ $\underline{M}$ is an array of sets of pair of $\mathcal{M} \times \mathsf{N}$
22: $b \leftarrow \forall i \in [0, |\underline{M}| - 1], \forall (m', \mathsf{sn}') \in \underline{M}[i], (m', \mathsf{sn}') \in \mathit{validated}$
23: $b' \leftarrow \forall i, j \in [0, |\underline{M}| - 1], \forall ((m_1, \mathsf{sn}_1), (m_2, \mathsf{sn}_2)) \in \underline{M}[i] \times \underline{M}[j]$, $sender(m_1) = sender(m_2)$: $\mathsf{sn}_1 < \mathsf{sn}_2 \implies i \leq j$
24: **return** $b \wedge b'$

We note *refers to* the transitive closure of the relation *directly refers to*. We use the notation $\tilde{M}^1 \prec_{\tilde{M}^{sc}(\bar{\alpha})} \tilde{M}^2$ to say that $\tilde{M}^2$ refers to $\tilde{M}^1$.

For every message $m_1 \in \mathcal{M}$, if $m_1 \in M_1'$, we say that (m_1, s_1) *belong to* $\tilde{M}_1$. If $(m_1, s_1) \in \underline{M}^2$ we say that $\tilde{M}_2$ *directly refers to* (m_1, s_1) ($\underline{M}^2$ is seen as a set of pairs (m, s) of $\mathcal{M} \times \mathsf{N}$ with a slight abuse of notation). If (m_1, s_1) belongs to $\tilde{M}^1$ and (m_2, s_2) belongs to $\tilde{M}^2$ with $\tilde{M}^2$ that directly refers to (resp. refers to) $\tilde{M}^1$, we say that (m_2, s_2) directly refers to (resp. refers to) (m_1, s_1) and we note $(m_1, s_1) \prec_{M^{sd}(\bar{\alpha})} (m_2, s_2)$ for (m_2, s_2) refers to (m_1, s_1).

A *circle* of $\tilde{M}^{sd}(\bar{\alpha})$ is a subset $\{\tilde{M}_1, ..., \tilde{M}_k\}$ of $\tilde{M}^{sd}(\bar{\alpha})$ s. t. $\tilde{M}_1$ directly refers to $\tilde{M}_k$ and $\forall i \in [2, k]$, $\tilde{M}_i$ directly refers to $\tilde{M}_{i-1}$. We note $\mathit{circles}(\tilde{M}^{sd}(\bar{\alpha}))$ the union set of circles of $\tilde{M}^{sd}(\bar{\alpha})$. Finally we note $\tilde{M}^{wf}(\bar{\alpha}) = \tilde{M}^{sd}(\bar{\alpha}) \setminus circles(\tilde{M}^{sd}(\bar{\alpha}))$ and $M^{wf}(\bar{\alpha})$ the set of pairs $(m, s) \in \mathcal{M} \times \mathsf{N}$ that belong to a member of $\tilde{M}^{wf}(\bar{\alpha})$($wf$ stands for well-formed).

Lemma 2 ($\prec_{M^{wf}(\bar{\alpha})}$ is a strict partial order)**.** Let Π be a distributed protocol with $\mathcal{M}$ denoting the attached set of messages and $\bar{\Pi} = \tau_{scr}(\Pi)$. Let $\bar{\alpha} \in execs(\bar{\Pi})$. Then the relation $\prec_{\tilde{M}^{sd}(\bar{\alpha})}$ (resp. $\prec_{M^{sd}(\bar{\alpha})}$) is a strict partial order on $\tilde{M}^{wf}(\bar{\alpha})$ (resp. on $M^{wf}(\bar{\alpha})$)

Proof. The transitivity comes from the definition built via transitive closure, while irreflexivity comes also by construction from the fact $\tilde{M}^{wf}(\bar{\alpha}) = \tilde{M}^{sd}(\bar{\alpha}) \setminus circles(\tilde{M}^{sd}(\bar{\alpha}))$. □

Safety preservation: We prove a sequence of lemma to show that the transformation preserves safety.

First, we show that for every behaviour β_p executing by the state-machine layer of a correct process p, process p stores a set of valid behaviors $VB^p = behaviours_p[:][1] = \{\beta_q^p | q \in \Psi\}$ explainable by a unique fully-benign execution α with $\forall q \in \Psi$, $\alpha|_q = \beta_q^p$.

Lemma 3 (Only full-benign execution reach state-machine (even for $t = n - 1$))**.** Let Π be a non-synchronous distributed protocol. Let $\bar{\alpha} \in execs(\bar{\Pi})$ with $\bar{\Pi} = \tau_{scr}(\Pi)$. Let $p \in Corr_{\bar{\Pi}}(\bar{\alpha})$. p stores a set VB^p , $behaviours_p[:][1] = \{\beta_q = behaviours_p[q][1] | q \in \Psi\}$ of valid-behaviours, s. t. it exists a fully-benign execution $\alpha \in execs(\Pi)$ with $\forall q \in \Psi, \alpha|_q = \beta_q$.

Proof. The stored behaviour $\beta_q = behaviours[q][1]$ is uniquely defined by messages secure-delivered from q. The validity-check of line 17 applied to process q using the sub-algorithm $ValidBehaviour$ (algorithm 2), ensures the validity of the stored behaviour, i. e. the potential non-valid events of q do not affect the state machine Π_p and are not stored in behaviour[q][1].

Algorithm 3 Pseudocode of τ_{scr} - process p

1: **upon** INIT
2: $\mathit{seqNum} \leftarrow 1$
3: $\mathit{delivered} \leftarrow \emptyset$ ▷ secure-delivered messages
4: $\mathit{validated} \leftarrow \emptyset$ ▷ validated messages
5: $\mathit{next} \leftarrow [0]^n$
6: $\mathit{behaviours} \leftarrow [(s_p^0, []), (s_q^0, []), ..., (s_z^0, [])]$ for $p, q, ..., z \in \Psi$
7: $\mathit{lastReceived} \leftarrow []$
8: **upon** SEND(M)
9: $\tilde{M} \leftarrow ((M, \mathit{seqNum}), \mathit{lastReceived})_{\sigma_p}$
10: $\mathit{seqNum} \leftarrow \mathit{seqNum} + 1$
11: $\mathit{lastReceived} \leftarrow []$ ▷ A correct process p scr-bcast a message $m \in \mathcal{M}$ with $receiver(m) = p$ only once
12: SECUREBROADCAST($\tilde{M}$)
13: **upon** SECUREDELIVER($\tilde{M}$)
14: $\mathit{delivered} \leftarrow \mathit{delivered} \cup \{\tilde{M}\}$
15: **upon exists** $((M^*, \mathit{sn}), M^{\ell r})_{\sigma_q} \in \mathit{delivered}$ such that $sn = next[q] + 1 \wedge IsValid(M^{\ell r}, validated)$
16: $\zeta_q \leftarrow behaviours[q][0]$
17: $(\zeta_q', \beta_q') \leftarrow \mathit{ValidBehaviour}'(\zeta_q, M^{\ell r}, M^*)$
18: **if** $\zeta_q' \neq \perp$ **then** ▷ In practice, if $\zeta_q' \neq \perp$, then p will ignore q forever
19: $\mathit{validated} \leftarrow \mathit{validated} \cup \bigcup_{m \in M^*} \{(m, \mathit{sn})\}$
20: $behaviours[q][0] \leftarrow \zeta_q'$
21: $behaviours[q][1] \leftarrow behaviours[q][1] || \beta_q'$
22: **end if**
23: **upon exists** $(m, \mathit{sn}) \in \mathit{validated}$ such that $sn = next[sender(m)] + 1$
24: $\mathit{next}[sender(m)] \leftarrow \mathit{sn}$
25: **if** $\mathit{receiver}(m) = p$ **then**
26: $\mathit{lastReceived}.append(\{m, \mathit{sn}\})$
27: RECEIVE(m)
28: **end if**
29: **upon exists** $(m, \mathit{sn})_{\sigma_q}, (m', \mathit{sn}')_{\sigma_q} \in \mathit{delivered}$ such that $\mathit{sn} = \mathit{sn}'$ **and** $m \neq m'$
30: DETECT(q) ▷ equivocation

It is impossible for p to validate non well-formed messages that refers to each other since they have to be validated one by one and so the last predicate of line 15 will never be verified. Thus the messages validated by p are in a subset of $M^{wf}(\bar{\alpha})$ where $\prec_{M^{sd}(\bar{\alpha})}$ is a strict partial order by lemma 2. Hence it is possible to extend this partial order with a total order to construct an execution $\alpha \in execs(\Pi)$ with $\forall q \in \Psi, \alpha|_q = \beta_q$. □

Then, we show that correct processes agree on a common set of valid behavior explainable by a fully-correct execution of the original protocol.

Lemma 4 (Agreement on correct execution for $t < n/3$)**.** Let Π be a non-synchronous distributed protocol. Let $\bar{\alpha} \in execs(\bar{\Pi})$ with $\bar{\Pi} = \tau_{scr}(\Pi)$. As long as $t < n/3$, all the correct processes in$Corr_{\bar{\Pi}}(\bar{\alpha})$, store a set $VB(\bar{\alpha}) = \{\beta_q | q \in \Psi\}$ of valid-behaviours, s. t. it exists a fully-benign execution $\alpha \in execs(\Pi)$ with $\forall q \in \Psi, \alpha|_q = \beta_q$.

Proof. The sequences of messages secure-delivered by correct processes from r verify a prefix relationship by Source-Order property of secure-bcast. Since the behaviour of r is uniquely defined by messages secure-delivered from r, we preserve the prefix relationship.

Hence, we can take $VB(\bar{\alpha}) = \{\beta_q^{p(q)} | q \in \Psi\}$ where $\beta_q^{p(q)}$ is the minimal common prefix of $\{\beta_q^r \in \bigcup_{r \in \Psi} VB^r | q \in \Psi\}$. The validity of each behaviour comes from previous lemma 3. □

Last lemma 4 will be enough for safety preservation and a key argument for accountability.

Liveness preservation: We prove a sequence of lemma to show that the transformation preserves safety.

Here we show that a message validated by a correct process is eventually validated by all the correct processes.

Lemma 5 (uniformity of validation for $t < n/3$)**.** Let Π be a distributed protocol with $\mathcal{M}$ denoting the attached set of messages and $\bar{\Pi} = \tau_{scr}(\Pi)$. Let $(m, s) \in \mathcal{M} \times \mathsf{N}$ validated by a correct process p in an execution $\bar{\alpha} \in execs(\bar{\Pi})$. If $t < n/3$, then every correct process eventually validates (m, s).

Proof. To be validated, necessarily (m, s) belongs to $\tilde{M} \in \tilde{M}^{sc}(\bar{\alpha})$. We note $causal\text{-}past((m, s), p, \bar{\alpha})$ the maximum set of messages $\tilde{M'}$ secure-delivered by p in $\bar{\alpha}$ s. t. $\tilde{M}$ refers to $\tilde{M'}$. Clearly $causal\text{-}past((m, s), p, \bar{\alpha})$ is finite since $\tilde{M}^{sd}(\bar{\alpha})$ is finite because $\bar{\alpha}$ is finite.

By properties of secure-broadcast, as long as $t < n/3$, all the correct processes eventually secure-deliver the sames messages, and so $causal\text{-}past((m, s), p, \bar{\alpha})$.

Now by induction, we show that all messages in $causal\text{-}past((m, s), p, \bar{\alpha})$ are validated by p. Basis: by definition (m, s) is validated. Induction: Let $(m_{k+1}, s_{k+1}) \prec_{M^{sd}(\bar{\alpha})} (m_k, s_k)$ s. t. (m_k, s_k) directly refers to (m_{k+1}, s_{k+1}) either i) $sender(m_k) = sender(m_{k+1})$ and $s_k = s_{k+1} + 1$ or ii) not. If i), by predicate $sn = next[sender(m_k)] + 1$ of line 15, the line 24 has been activated after the line 23 and by predicate $(m_{k+1}, s_{k+1}) \in validated$ of line 23, (m_{k+1}, s_{k+1}) has been accepted. If ii) by second predicate of line 15, we have $IsValid(\underline{M}^{\ell r}_{k+1}, validated)$ executed at line 21 of algorithm 2 and so by line 22 of algorithm 2, (m_{k+1}, s_{k+1}) has been validated.

The induction terminates since $causal\text{-}past((m, s), p, \bar{\alpha})$ is finite and hence all the messages in $causal\text{-}past((m, s), p, \bar{\alpha})$ have been validated by p.

Let q be another correct process that eventually stores $causal\text{-}past((m, s), p, \bar{\alpha})$.

By definition of $causal\text{-}past((m, s), p, \bar{\alpha})$ and by construction of the algorithm, the validation of a message in $causal\text{-}past((m, s), p, \bar{\alpha})$ do not depend on messages not in $causal\text{-}past((m, s), p, \bar{\alpha})$. By symmetry of the algorithm, if all the messages in $causal\text{-}past((m, s), p, \bar{\alpha})$ have been validated by p, q will do the same.

□

Now, we show that a message m sent from the state-machine layer of a correct process is eventually validated by all the correct processes and hence will reach the state-machine layer of $receiver(m)$ if it is correct.

Lemma 6 (obligation of validation)**.** Let Π be a distributed protocol with $\mathcal{M}$ denoting the attached set of messages and $\bar{\Pi} = \tau_{scr}(\Pi)$. If a correct process p send a message m, then it is eventually validated by every correct process.

Proof. We first show it is eventually validated by process p. The process p $secure\text{-}bcast(\tilde{M})$ with $\tilde{M} = ((M', s), \underline{M})_{\sigma_p}$ and $m \in M'$.

The messages in $\underline{M}$ have been added via line 26 of algorithm 3 and so have been validated by predicate of line 23 of algorithm 3.

Let assume A_1: (m, s) is the first message non-validated by p. So either it exists $(m'', s - 1)$ sent by p that has been validated or $s = 1$. In both case, $s = next[p] + 1$ which means predicate of line 15 of algorithm 3 is eventually met.

We note e^{ℓ}_p the event containing the sent of last validated message if it exists and ζ_p the state equal to the one reached immediately after e^{ℓ}_p if it exists and equal to s^0_p otherwise.

Since p is correct, message M' has been built accordingly to Π_p, the state ζ_p, and the last received messages updating according to line 26 and line 11 of algorithm 3. Thus, the state $\zeta'_p = \mathsf{ValidBehaviour}(\zeta_p, \underline{M}, M')$ computed at line 17 will return $\zeta'_p \neq \perp$ and line 19 will be visited s. t. (m, s) will be validated.

Since the first non-validated yet message sent by p is eventually validated, by induction, all the messages sent by p are eventually validated by p.

Finally, by uniformity property proved at previous lemma 5, all the messages sent by a correct process p are eventually validated by every correct process q.

□

It is time to show that both liveness and safety are preserved by the solution.

Lemma 7 (Safety and Liveness preservation for $t \leq t_0$ and $t < n/3$)**.** Let Π be a non-synchronous distributed protocol solving a decision task $\mathcal{D}$ as long as $t \leq t_0$. Then $\bar{\Pi} = \tau_{scr}(\Pi)$ is a a non-synchronous distributed protocol solving decision task $\mathcal{D}$ as long as $t \leq t_0$ and $t < n/3$.

Proof. Safety: By previous lemma 4, $\exists \alpha \in execs(\Pi)$ s. t. $\forall q \in \Psi, \alpha|_q$ is valid and only $\alpha|_q$ is reaching the state machine of a correct process q. Since Π is at least 0-resilient, safety is preserved.

Liveness: By lemma 6, every message m sent from state machine Π_p of a correct process p is eventually received by state machine Π_q of a correct process q as long as $t < n/3$. Thus an infinite execution $\bar{\alpha} \in execs(\bar{\Pi})$ corresponds to an infinite fully-benign execution $\alpha \in execs(\Pi)$ with at most t_0 crashed processes. If the original algorithm ensure liveness as long as $t \leq t_0$ without synchrony assumption, then $\bar{\Pi}$ does the same as long as $t \leq t_0$ and $t < n/3$. If the original algorithm ensures liveness under partial synchrony as long as $t \leq t_0$, then $\bar{\Pi}$ does also the same as long as $t \leq t_0$ and $t < n/3$ since

the time of message-delivery will stay bounded after transformation because secure-bcast has a bounded round complexity and partial synchrony eventually concerns all channels connecting pairs of correct processes. □

Accountability: We use the store of common valid behaviours to provide accountability.

Lemma 8 (Disseminated Proof)**.** Let Π be a non-synchronous protocol solving a distributed task $\mathcal{D}$ with t_0-resiliency. Let $\bar{\Pi} = \tau_{scr}(\Pi)$, $\bar{\alpha} \in execs(\bar{\Pi})$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq 2$ such that $\bar{\Pi}$ violates safety of $\mathcal{D}$ in $\bar{\alpha}$. Then there exists $i, j \in C \subseteq Corr_{\bar{\Pi}}(\bar{\alpha})$, s. t. the union of their respective stored set of valid behaviours VB^i and VB^j contains at least $t_0 + 1$ pairs of mutant messages from as many different Byzantine processes.

Proof. Because of lemma 3, every correct process p stores a set VB^p of valid behaviours that can be explained by a fully-benign execution $\alpha_p \in execs(\Pi)$ justifying the decision of p in $\bar{\alpha}_p$.

By contradiction, we assume the lemma no to be true. We will build an execution $\alpha' \in execs(\Pi)$ that would violate the safety distributed-property with less than t_0' Byzantine faults, which will be in contradiction with t_0'-resiliency of Π. For every correct $i \in C \subset Corr_{\bar{\Pi}}(\bar{\alpha})$, α_i is fully-benign because of valid enabling. We note $\beta_i = \alpha_i|_i$, the behavior that i should have in the execution α_i of the original algorithm. We fix B, $|B| \leq t_0'$ that will play the role of Byzantine processes. We note $K = \Psi \setminus B$ (Korrect). For every (really) correct process k in K, for every fully-correct behavior stored , by lemma 4 the sent messages are not conflicting and respect a common-prefix, that is $\forall i, j, k \in K, \beta_{k,i} = \alpha_i|_k$, $\beta_{k,j} = \alpha_j|_k$, either $\beta_{k,i} \leq \beta_{k,j}$ or $\beta_{k,j} \leq \beta_{k,i}$.

We construct α^* as follows: The Byzantine processes in B behave with each correct i as they did in execution α_i, using the behavior $\beta_{b,i} = \alpha_i|_b$ for each $b \in B$. In α^*, each correct process $i \in K$ behaves as they did in α_i, using $\beta_i = \alpha_i|_i$. By lemma 4 and the fact that K did not commit any commission fault, every member of K has the same behavior in each α_i, so nothing change in the communication between members of K. Then each correct process i visit the sequence of states $\vec{s}_i = (s_i^0, ..., s_i^n)$ corresponding to the behavior β_i so that α^* violates the safety property. But this is no possible since we assumed the original algorithm ensures safety of $\mathcal{D}$ with t_0'-resiliency and $|corr_{\Pi}(\alpha^*)| = |K| \geq |\Psi| - t$. Thus there is a contradiction which prove that the assumption was not true. Finally, we conclude that at least $t_0' + 1$ mutant messages from as many Byzantine processes are stored in the union of respective sets of valid behaviours VB^i and VB^j stored by two correct processes $i, j \in C$. □

When two correct nodes i and j reach respective states s_i and s_j, they respectively store sets of valid behaviours VB^i and VB^j that respectively justify s_i and s_j. A way to centralise the potential proofs of culpability is to broadcast VB^i after the decision. In fact, each message $\tilde{M}_1 = \langle (M_1, sn_q), \underline{M}_1, \rangle_q$ secure-delivered by process i, will be echoed to j (and vice-versa). If process j stores the message (instead of ignoring it because it already secure-delivered $\tilde{M}_2 = \langle (M_2, sn_q), \underline{M}_2, \rangle_q$ with $\tilde{M}_2 \neq \tilde{M}_1$), the correct processes will be able to centralise the proof. We slightly abuse the notation τ_{scr} to refer to the extension of τ_{scr} where mutant messages are stored.

Lemma 9 (Accountability)**.** Let Π be a non-synchronous protocol solving a distributed task $\mathcal{D}$ with t_0-resiliency. Let $\bar{\Pi} = \tau_{scr}(\Pi)$, $\bar{\alpha} \in execs(\bar{\Pi})$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq 2$ that leads to such that $\bar{\Pi}$ violates safety of $\mathcal{D}$ in $\bar{\alpha}$, then then every correct process eventually stores $t_0 + 1$ pair of mutant messages from as many Byzantine processes.

Proof. The pairs of mutant messages are stored in the union of valid behaviours set according to lemma 8. Thus these two correct processes just have to broadcast their stored valid behaviours to everybody. □

We can conclude and states that τ_{scr} both provides accountability and preserves safety and liveness of the original protocol.

Theorem 7 (τ_{scr} provides accountability and preserves solution)**.** *Let Π be a non-synchronous protocol solving a distributed task $\mathcal{D}$ with t_0-resiliency. Then $\bar{\Pi} = \tau_{scr}(\Pi)$ solves distributed task $\mathcal{D}$ with t_0-resiliency and provides accountability.*

Proof. By lemma 9 and lemma 7. □

Appendix C
Homomorphic Transformations

This section is devoted to formally defining a homomorphic transformation used in the definition of an accountable counterpart (Definition 3).

Definition 12 (Homomorphic Transformation)**.** Let Π be a distributed protocol. We say that $(\bar{\Pi}, \Pi, \mu_e)$ is a *homomorphic transformation* if μ_e is a total map $\mathsf{execs}(\bar{\Pi}) \rightarrow \mathsf{execs}(\Pi)$, verifying:

- *Prefix ordering preservation:* For every execution $\bar{\alpha}_2$ being a prefix of an execution $\bar{\alpha}_1$, $\mu_e(\bar{\alpha}_2)$ is a prefix of $\mu_e(\bar{\alpha}_1)$.
- *Correctness preservation:* For every execution $\bar{\alpha}$, for every valid behavior $\bar{\beta}_i = \bar{\alpha}|_i$ according to $\bar{\Pi}$, $\mu_e(\bar{\alpha})|_i$ is a valid behavior according to Π.

- *Homomorphic transition:* For every process p, there exist two computable total map functions $\mu^i_{sm}, \mu^o_{sm} : \bar{\mathcal{S}}_p \times \mathcal{P}(\bar{\mathcal{M}}) \rightarrow \mathcal{S}_p \times \mathcal{P}(\mathcal{M})$, s. t. $\forall \bar{s}_1, \bar{s}_2 \in \bar{\mathcal{S}}_p, \bar{m}_{in}, \bar{m}_{out} \subseteq \bar{\mathcal{M}}, i_{in} \subseteq \mathcal{I}_p, \bar{i}_{out} \subseteq \bar{\mathcal{O}}_p : [\bar{\mathcal{T}}_p(\bar{s}_1, \bar{m}_{in} \cup i_{in}) = (\bar{s}_2, \bar{m}_{out} \cup \bar{i}_{out})] \implies [\mathcal{T}_p(\mu^i_{sm}((\bar{s}_1, \bar{m}_{in} \cup i_{in})) = (\mu^o_{sm}((\bar{s}_2, \bar{m}_{out} \cup (i_{out} \setminus XO))))]$ where μ^i_{sm}, μ^o_{sm} preserve the internal events, that is $\mu^i_{sm}((\bar{s}_1, \bar{m}_{in} \cup i_{in})) = (s_1, m_{in} \cup i_{in})$ with $m_{in} \cap \mathcal{I} = \emptyset$ and $\mu^o_{sm}((\bar{s}_2, \bar{m}_{out} \cup \bar{i}_{out})) = (s_2, m_{out} \cup i_{out})$ with $m_{out} \cap \mathcal{O} = \emptyset$ and $i_{out} = \bar{i}_{out} \setminus XO$.

The prefix preservation stipulates that the homomorphism cannot change the past of what happened. The correctness preservation specifies that a correct process cannot be considered faulty in an execution of the original protocol if it is correct in an execution of the transformed protocol. The homomorphic transition condition restricts the power of the homomorphism. We stress that μ^i_{sm}, μ^o_{sm} have to be computable, thus a process cannot use some inaccessible information like the internal states of all other processes to perform a transition. Hence, homomorphic transformation cannot be used as a non-computable way to reduce communication complexity or to circumvent impossibility results.

This transformation is slightly more general than the one proposed in [24]. For example, in [24] (item X4, page 4), each message $m \in \mathcal{M}$ has at least one counterpart $\bar{m} \in \bar{\mathcal{M}}$ s. t. $\mu_m(\bar{m}) = m$ where μ_m is message mapping appearing in the transition homomorphism of [24] (item X6 page 4). In our definition, there are two mappings μ^i_{sm} and μ^o_{sm} where the operands are pairs of the form (state, message). This allows us to deliver a message of the original algorithm upon the reception of a certain number of messages from witnesses, typically via a double-echo reliable-broadcast algorithm. This kind of transformation is not covered by [24] since it would mask some faults (an execution α with a certain fault instance would not have pre-image in the function μ_e) instead of preserving them and then detecting them, which was the original aim of [24].

Appendix D
τ_{scr} Generalization

In the main body of the paper, we proposed a transformation that ensures accountability whenever a distributed protocol violates safety of its decision task. In this section, we lay ground to showing that the application of our transformation is broader. Specifically, we define a *distributed property* of distributed protocol and categorize them into *safety* and *liveness*. Moreover, we define what it means to violate a safety distributed property[16]. This section is crucial to proving that our transformation ensures accountability whenever *any* safety distributed property is violated.

A. Distributed Properties

Our definition of distributed properties is based on [1], where properties are sets of sequences of states. The difference is that we replace states by vectors of internal states of correct processes. By construction, this choice excludes the notion of privacy. For example, we cannot express the fact that no (Byzantine) process learns a fact that was supposed to be a secret.

Definition 13 (Distributed Property)**.** Let $\Pi = (\Pi_p, \Pi_q, ..., \Pi_z)$ be a distributed protocol. Let $\mathcal{S}_i$ denote the state set of a protocol Π_i, for every $i \in \Psi$.

Consider a set $\Phi \subseteq \Psi$. We say that $\Phi = (\phi_1, \phi_2, ..., \phi_k)$ *generates* $\underline{\mathcal{S}}(\Phi)$ if and only if $\underline{\mathcal{S}}(\Phi) = \mathcal{S}_{\phi_1} \times \mathcal{S}_{\phi_2} \times ... \times \mathcal{S}_{\phi_k}$.

A *consistent pair-state* of Π is a pair $(\Phi, \underline{s}) \in P(\Psi) \times \underline{\mathcal{S}}(\Phi)$, that is, a pair where the first element is a set of processes, while the second element is a vector of internal states of the processes in the first element.

We denote by $\Gamma(\Pi)$ the set of all consistent pair-states of Π, i.e., $\Gamma(\Pi) = \{(\Phi, \underline{s}) \,|\, \Phi \subseteq \Psi, \underline{s} \in \underline{\mathcal{S}}(\Phi)\}$. We denote by $\Gamma(\Pi)^*$ (resp., $\Gamma(\Pi)^\omega$) the set of finite (resp., infinite) sequence of $\Gamma(\Pi)$. We denote by $\Gamma(\Pi)^{\omega *}$ the set $\Gamma(\Pi)^\omega \cup \Gamma(\Pi)^*$.

Let $\gamma \in \Gamma(\Pi)^{\omega *}$. We say that γ respects *monotonic inclusion* if $\Phi_i \subseteq \Phi_j$, for every element $(\Phi_i, \underline{s}_i)$ that follows an element $(\Phi_j, \underline{s}_j)$ in γ.

Let $\gamma \in \Gamma(\Pi)^{\omega *}$ that respects the monotonic inclusion. A *restriction* γ' of γ is an element of $\Gamma(\Pi)^{\omega *}$ such that (1) $|\gamma'| = |\gamma|$, (2) γ' respects the monotonic inclusion, and (3) for every $i \in |\gamma|$, let $\gamma'[i] = (\Phi'_i, \underline{s}'_i)$ and $\gamma[i] = (\Phi_i, \underline{s}_i)$ the respective i-th element of γ and γ', then $\Phi'_i \subseteq \Phi_i$ and $\underline{s}'_i = \underline{s}_i|_{\Phi'_i}$, i.e., the sequence of states $\underline{s}'_i$ is the restriction of $\underline{s}_i$ on Φ'_i.

A *distributed property* P of the distributed protocol Π is an element of $\Gamma(\Pi)^\omega$ such that every element of P respects (1) the monotonic inclusion, and (2) P is closed under restriction, i.e., if $\gamma \in P$, then $\gamma' \in P$, for every restriction γ' of γ.

A consistent pair-state represents a joint state of some set of processes. Since we operate in Byzantine environment, we are interested in ensuring some guarantees at correct processes only. Hence, a consistent pair-state is used to abstract a joint state of all correct processes. Note that the monotonic inclusion concept comes from the fact that number of correct processes could only decrease.

The last point of the definition that requires a special attention is the notion of a restriction. We precise that if a property is verified by a set of correct processes, then it is also verified by each subset of correct processes considering that the

[16]Do not confuse violation of safety of a decision task with violation of a safety distributed property.

complementary set is, in fact, Byzantine. This constraint is necessary, since we cannot prevent Byzantine processes from behaving correctly and eventually becoming Byzantine.

Our formalism allows properties that are not preserved under finite repetition of individual state. However, we need to avoid the possibility for Byzantine processes to artificially impose this repetition. Thus, the Byzantine processes will be ignored in our definition of generation of sequence of states (Definition 16).

Definition 14 (Safety Distributed Property)**.** Let Π be a distributed protocol. A distributed property P of Π is a *safety distributed property* of Π if and only if the following holds:

$$\forall \gamma \in \Gamma(\Pi)^{\omega} : \gamma \notin P \implies [\exists i \geq 0 : \forall \gamma' \in \Gamma(\Pi)^{\omega} : \gamma_i \frown \gamma' \notin P],$$
where $\gamma_i \in \Gamma(\Pi)^{*}$ represents a prefix of size i of γ.

Intuitively, a safety property is a property that cannot be repaired. Specifically, if an execution violates a safety property, then there exists a prefix of the execution such that the property is violated in all its continuations. For example, the agreement property of the consensus problem [29] is a safety property. Indeed, once correct processes disagree, the agreement is violated and will never be satisfied again.

Definition 15 (Liveness Distributed Property)**.** Let Π be a distributed protocol. A distributed property P of Π is a *liveness distributed property* of Π if and only if the following holds:

$$\forall \gamma \in \Gamma(\Pi)^{*}, \exists \gamma \in \Gamma(\Pi)^{\omega} : \gamma \frown \gamma' \in P.$$

A liveness property ensures that "something good eventually happens". More formally, a distributed property is a liveness distributed property if, for every finite execution, there exists an infinite continuation where the property is satisfied. For instance, the termination property of the consensus problem is a liveness property. If the termination is not satisfied in an execution α, there exists a continuation α' of α that satisfies the termination (a correct process *eventually* decides).

Let Π be a distributed protocol. For each execution α of Π, we use the following notation:

- α_i represents the prefix of α of size i;
- $\mathit{state}^{corr}(\Pi, \alpha)$ denotes a vector composed of states of each process in $\mathit{Corr}_{\Pi}(\alpha)$ after α;
- $ps^{corr}(\Pi, \alpha) = (\mathit{Corr}_{\Pi}(\alpha), \mathit{state}^{corr}(\Pi, \alpha))$ represents a consistent pair-state of $\Gamma(\Pi)$.

Definition 16 (Generation of a Sequence of States)**.** Let Π be a distributed protocol and let P be a distributed property of Π. Let α be a finite or infinite execution of Π. We denote by $\mathit{gen}(\alpha, \Pi)$ the sequence $ps^{corr}(\Pi, \alpha_1), ..., ps^{corr}(\Pi, \alpha_n), ...$ where $\alpha|_k$ denotes the prefix of size k of α and all consistent pair-states $ps^{corr}(\Pi, \alpha_k)$ with α_k ending on event (p_k, I_k, O_k) with $p_k \notin \mathit{Corr}_{\Pi}(\alpha_k)$ are removed.

Definition 17 (Ensuring a Distributed Property)**.** Let Π be a distributed protocol and let P be a distributed property of Π. We say that Π *ensures* P *with* t_0*-resiliency* if and only if for every infinite execution α of Π where $|\mathit{Corr}_{\Pi}(\alpha)| \geq |\Psi| - t_0$, $\mathit{gen}(\alpha, \Pi)$ is in P.

Definition 17 states that a distributed protocol Π ensures a distributed property P with t_0-resiliency if all executions where number of faulty processes does not exceed t_0 satisfy P. Let a distributed protocol Π satisfy distributed properties $P_1, P_2, ..., P_k$ with $t_1, t_2, ..., t_k$-resiliency, respectively. Then, we say that Π is a t-resilient protocol, where $t = \mathit{min}(t_1, t_2, ..., t_k)$.

Definition 18 (Violating a Safety Distributed Property)**.** Let Π be a distributed protocol and let P be a safety distributed property of Π. We say that an execution α of Π *violates* P if and only if (1) α is finite and for every $\gamma' \in \Gamma(\Pi)^{\omega}$, $\mathit{gen}(\alpha, \Pi) \frown \gamma' \notin P$, or (2) α is infinite and $\mathit{gen}(\alpha, \Pi) \notin P$.

A safety distributed property P is violated in a finite execution α of Π if there does not exist an infinite continuation of α where P is satisfied. Moreover, a safety property is violated in an infinite execution if P is not satisfied in the execution. For example, suppose that α represents an execution of a consensus protocol such that two correct processes disagree. Clearly, the agreement property is violated in α since there does not exist a continuation of α where the agreement property is satisfied.

B. General Accountable Counterpart

In Definition 3, we gave a definition of an accountable counterpart of distributed protocols that solve decision tasks. We now slightly expand this definition to imply accountability whenever *any* safety distributed property (see Definition 14) is violated.

Definition 19 (General Accountable Counterpart)**.** Let Π be an asynchronous (resp., a partially synchronous) t_0-resilient distributed protocol. We say that an asynchronous (resp., a partially synchronous) distributed protocol $\bar{\Pi}$ is a *general*

accountable counterpart of Π with factor $f \in [1, t_0]$ according to a basis if there exists a homomorphic transformation $(\bar{\Pi}, \Pi, \mu_e)$ [17] with $\mu_e : execs(\bar{\Pi}) \rightarrow execs(\Pi)$ that verifies the following conditions:

- *Property Preservation:* Let P be a distributed property ensured by Π with t_0'-resiliency. Then, the following hold:
  - If P is a safety distributed property of Π, then for every execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t_0'$, $\mu_e(\bar{\alpha})$ does not violate P.
  - If P is a liveness distributed property of Π and $t_0' \leq f$, then for every infinite execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t_0'$, $\mathit{gen}(\mu_e(\bar{\alpha}), \Pi) \in P$.
- *Accountability:* Let $\bar{\alpha}$ be an execution of $\bar{\Pi}$ such that $\alpha = \mu_e(\bar{\alpha})$ violates a safety distributed property P of Π and let Π ensure P with t_0'-resiliency. Then, in every infinite continuation of $\bar{\alpha}$, every correct process detects at least $t_0' + 1$ faulty processes according to Π.
- *Syntactic Correspondence:* Let $\mathit{execs}(\Pi, t_0)$ represent a set of all executions of Π with at most t_0 faulty processes. Then, the following hold:
  - Let $\bar{\alpha}$ be an execution of $\bar{\Pi}$. If a process $p \in \Psi$ is correct in $\bar{\alpha}$, then p is correct in $\mu_e(\bar{\alpha})$.
  - For every execution $\alpha \in \mathit{basis}$, where $\mathit{basis} \subseteq \mathit{execs}(\Pi, t_0)$, there exists an execution $\bar{\alpha}$ of $\bar{\Pi}$ such that $\alpha = \mu_e(\bar{\alpha})$.

A liveness property P, which is satisfied by the original distributed protocol Π with t_0'-resiliency, is ensured only if $t_0' \leq f$, because the transformation itself might be prevented from making progress with f_0 Byzantine processes, where $f < f_0 \leq t_0'$. As shown in §V, our transformation uses a factor of $f = \mathit{min}(\lceil n/3 \rceil - 1, t_0)$ and is based on the secure-broadcast primitive. Therefore, a liveness property that is ensured with $t_0' > \lceil n/3 \rceil - 1$ might not be preserved in a transformed protocol since liveness properties of the secure-broadcast could be violated when number of faulty processes exceeds $\lceil n/3 \rceil - 1$. A *basis* represents a set of preserved executions.

We stress the difference between Definition 3 and Definition 19. Indeed, according to Definition 19, for every $\bar{\alpha} \in execs(\bar{\Pi})$, accountability has to be ensured if $\alpha = \mu_e(\bar{\alpha})$ (and not $\bar{\alpha}$, as in Definition 3) violates a safety distributed property. Definition 19 is more general, but is less intuitive.

C. Solving a Decision Task = Ensuring Distributed Properties

This subsection proves that a distributed protocol $\Pi_{\mathcal{D}}$ that solves a decision task $\mathcal{D}$ ensures a distributed property which is the conjunction of a safety and liveness distributed properties. In other words, we show that solving of a decision task (as defined in §III-B) implies ensuring safety and liveness distributed properties.

Let $\Pi_{\mathcal{D}}$ be a distributed protocol that solves a decision task $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \Delta)$ with t_0-resiliency. We assume that input and output values are elements of internal states of processes; we denote the input (resp., output) value of an internal state s by $s.\mathit{input}$ (resp., $s.\mathit{output}$).

Let $\alpha \in \mathit{execs}(\Pi_{\mathcal{D}})$. For $\gamma = \mathit{gen}(\alpha, \Pi_{\mathcal{D}})$, for $i \in |\gamma|$, for $\gamma[i] = (\Phi, \underline{s})$, we note $\mathit{input}((\Phi, \underline{s})) = \bigcup_{p \in \Phi}(p, \underline{s}[p].\mathit{input})$ and $\mathit{output}((\Phi, \underline{s})) = \bigcup_{p \in \Phi}(p, \underline{s}[p].\mathit{output})$, where $\underline{s}[p]$ denotes the element of $\mathcal{S}_p$ [18] in $\underline{s}$.

Finally, for every $\Phi \subseteq \Psi$, for every output configuration $C = (\{(p, O_p) \,|\, p \in \Phi\})$, we have $\mathit{deciders}(C) = \{p \,|\, (p, O_p) \in C \bigwedge O_p \neq \bot\}$, and we define an *extension* of C as an output configuration $C_e \supseteq C$. An extension C_e of C is an output configuration in which all processes that have outputted their values in C still output the same value, while potentially including output values of other processes, as well.

A protocol $\Pi_{\mathcal{D}}$ solves $\mathcal{D}$ if and only if $\Pi_{\mathcal{D}}$ ensures the following distributed properties[19]:

- *Eventual Stability* $\tilde{P}_{\mathcal{D}}^{Stability}$*:* the set of restrictions of $P_{\mathcal{D}}^{Stability} = \{\gamma \in \Gamma(\Pi)^{\omega} \,|\, \exists i \in \mathsf{N}, \forall j \geq i, \mathit{output}(\gamma_i) = \mathit{output}(\gamma_j)\}$. (Liveness)
- *Eventual Output* $\tilde{P}_{\mathcal{D}}^{Output}$*:* the set of restrictions of $P_{\mathcal{D}}^{Output} = \{\gamma \in \Gamma(\Pi)^{\omega} \,|\, \exists i \in \mathsf{N}, \forall j \geq i, \mathit{output}(\gamma_j) \in \Delta(\mathit{input}(\gamma_j)\}$. (Liveness)
- *Non-revocation* $\tilde{P}_{\mathcal{D}}^{NonRevocation}$*:* the set of restrictions of $P_{\mathcal{D}}^{NonRevocation} = \{\gamma \in \Gamma(\Pi)^{\omega} \,|\, \forall i, j \in \mathsf{N}, i < j, (\Phi_i, \underline{s}_i) = \gamma[i], (\Phi_j, \underline{s}_j) = \gamma[j], \forall p \in \Phi_j, ((\underline{s}_i[p].\mathit{output} = O_p \neq \bot \implies \underline{s}_j[p].\mathit{output} = O_p) \bigwedge (\underline{s}_j[p].\mathit{input} = \underline{s}_i[p].\mathit{input}))\}$. (Safety)
- *Rigorous Consistency* $\tilde{P}_{\mathcal{D}}^{RConsistency}$*:* the set of restrictions of $P_{\mathcal{D}}^{RConsistency}$ $= \{\gamma \in \Gamma(\Pi)^{\omega} \,|\, \forall i \in \mathsf{N}, (\Phi_i, \underline{s}_i) = \gamma[i],$ there exists an extension ν_O of $\mathit{output}(\gamma[i]), \nu_O \in \Delta(\mathit{input}(\gamma[i]))\}$. (Safety)

The liveness properties require a stable admissible output. The admissibility depends on the inputs of correct processes. The safety properties require that the "current" output configuration could always be extended to an admissible output configuration without any revocations.

[17] Homomorphic transformation is formally defined in Definition 12.

[18] Recall that $\mathcal{S}_p$ represents a state set of process p.

[19] We implicitly assume that monotonic inclusion holds for these properties.

Difference from the simpler definition given in §III-B: At this point, a reader might notice that Definition 18 does not match Definition 1. Indeed, if safety of a decision task is violated (according to Definition 1), it means that the rigorous consistency property is violated.

However, one must notice that we also adapted Definition 3 to reach Definition 19. For every $\bar{\alpha} \in execs(\bar{\Pi})$, accountability has to be ensured only if $\alpha = \mu_e(\bar{\alpha})$ (and not $\bar{\alpha}$, as in Definition 3) violates a safety distributed property.

For example, let Π be a distributed protocol that solves the binary consensus problem that assumes strong validity. Let Π solves the binary consensus problem with t_0-resiliency and let $\bar{\Pi}$ be an accountable counterpart of Π (according to Definition 3). Consider an execution $\bar{\alpha} \in \mathsf{execs}(\bar{\Pi})$ with a set B of $|B| > n/3$ of processes that behave correctly, modulo some non-observable faults that do not have any impact on the correct processes. Let us assume that the set $\mathsf{Corr}_{\bar{\Pi}}(\bar{\alpha})$ of processes decides a value that has not been proposed by any process in $\mathsf{Corr}_{\bar{\Pi}}(\bar{\alpha})$, but by a process in B. Then:

- $P^{RConsistency}$ is violated by $\bar{\alpha}$ according to definition 18. However, this violation is *irrelevant* according to Definition 19.
- $P^{RConsistency}$ is not violated by $\alpha = \mu_e(\bar{\alpha})$ according to Definition 18 (since $B \subset \mathsf{Corr}_{\Pi}(\alpha)$) and accountability is not required by Definition 19.
- Safety of the binary consensus task is not violated (according to Definition 1); thus, accountability is not required (according to Definition 3).

Lemma 10 (Safety of a Decision Task Violated = Rigorous Consistency Violated)**.** Let $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \Delta)$ be a decision task, where $\mathcal{I}$ contains input configurations of any cardinality, i.e., $|\nu_I| \geq 1$, for every $\nu_I \in \mathcal{I}$. Moreover, let $\Pi_{\mathcal{D}}$ be a distributed protocol that solves $\mathcal{D}$ with t_0-resiliency.

Let there exist a homomorphic transformation $(\Pi_{\mathcal{D}}, \bar{\Pi}_{\mathcal{D}}, \mu_e)$. If $\bar{\Pi}_{\mathcal{D}}$ violates safety of $\mathcal{D}$ in an execution $\bar{\alpha}$, then $\alpha = \mu(\bar{\alpha})$ violates the rigorous consistency safety distributed property.

Proof. Let $\bar{\nu}_O$ be the output configuration of processes correct in $\bar{\alpha}$. According to Definition 1, there does not an input configuration $\bar{\nu}_I$ such that $\bar{\nu}'_O \in \Delta(\bar{\nu}_I)$, where $\bar{\nu}'_O$ is an extension of $\bar{\nu}_O$.

Recall that, according to the definition of a homomorphic transformation, $\mathsf{Corr}_{\bar{\Pi}_{\mathcal{D}}} \subseteq \mathsf{Corr}_{\Pi_{\mathcal{D}}}$. Let ν_O denote the output configuration of processes correct in α. For every $(p, \bar{O}_p) \in \bar{\nu}_O$, $(p, O_p) \in \nu_O$ and $\bar{O}_p = O_p$.

Suppose that α does not violate the rigorous consistency property. Therefore, any restriction of $\mathsf{gen}(\alpha, \Pi_{\mathcal{D}})$ does not violate the property (because of the restriction constraint). However, we know that the restriction over processes correct in $\bar{\alpha}$ conflicts this statement. Hence, the lemma. □

Before stating the generalization theorem of this subsection, we introduce the last assumption that has to be verified by the transformation.

Definition 20 (I/O-preserving transformation)**.** Let $(\bar{\Pi}, \Pi, \mu_e)$ be a homomorphic transformation. We say μ_e is I/O-preserving if $\forall(\bar{\alpha}, \alpha) \in execs(\bar{\Pi}) \times execcs(\Pi)$ s. t. $\mu_e(\bar{\alpha}) = \alpha$, $\forall p \in Corr(\bar{\Pi}, \bar{\alpha})$, p has the same input and output at the last state of $\bar{\alpha}|_p$ and $\alpha|_p$

We slightly abuse the notation, by only considering homomorphic transformation that are I/O-preserving ,

Theorem 8 (A general accountable counterpart is an accountable counterpart)**.** *Let $\mathcal{D}$ be a decision task, $\Pi_{\mathcal{D}}$ be a distributed protocol that solves $\mathcal{D}$ with t_0-resiliency. If $\bar{\Pi}_{\mathcal{D}}$ is a general accountable counterpart (according to definition 19) of $\Pi_{\mathcal{D}}$ then this is an accountable counterpart of $\Pi_{\mathcal{D}}$ (according to definition 3).*

Proof.
- Solution preservation: Let $(\bar{\alpha}, \alpha) \in execs(\bar{\Pi}) \times execs(\Pi)$ s. t. $\mu_e(\bar{\alpha}) = \alpha$. The I/O-preserving property ensures the input and the output configuration of $corr(\bar{\Pi}, \bar{\alpha})$ is preserved. Hence, the safety properties non-revocation and rigorous consistency are ensured. Moreover, if $\bar{\alpha}$ is infinite, it means α is infinite too. Hence the liveness property is also ensured.
- Accountability: it comes from previous lemma 10
- Syntactic correspondence: immediate since the definition is the same for accountable counterpart and general accountable counterpart.

□

In the remaining of the paper, our proofs are written in the general manner (i.e., with Appendix D in our mind).

D. Topology of Distributed Properties

Here we give the topological arguments from [1] to establish that any distributed property is the intersection of a safety distributed property and a liveness distributed property.

Let $DSeq = \{\gamma \in \Gamma(\Pi)^\omega) | \forall \gamma \in P, \gamma$ verifies 1) monotonic inclusion and 2) restriction closeness$\}$

Let $DProp = \mathcal{P}(DSeq)$. We can note that for every $\{P_i\}_{i \in I} \in DProp^I$, then $\bigcup_{i \in I} P_i \in DProp$ and $\bigcap_{i \in I} P_i \in DProp$.

Let $\mathcal{C} = \{P_S \in DProp | P_S$ is a safety property$\}$.

Lemma 11. $\mathcal{C} = \{P_S \in DProp | P_S \text{ is a safety property}\}$ is a set of closed sets of $DSeq$, sometimes called a topology of closed sets.

Proof. We need to show that $\mathcal{C}$ verifies :

1) $\emptyset, DSeq \in \mathcal{C}$
2) For every $\{P_i\}_{i\in I} \in \mathcal{C}^I$, with I countable, then $\bigcap_{i\in I} P_i \in \mathcal{C}$.
3) For every $\{P_i\}_{i\in I} \in \mathcal{C}^I$, with I finite, then $\bigcup_{i\in I} P_i \in \mathcal{C}$.

1) The first point is immediate since we cannot have both $\gamma \in DSeq$ and $\gamma \notin DSeq$, so $DSeq$ and $\emptyset$ are the trivial safety properties.
2) Let $\{P_i\}_{i\in I} \in \mathcal{C}^I$, with I countable. Let $P = \bigcap_{i\in I} P_i \in \mathcal{C}$ Let $\gamma \notin \mathcal{C}$. Then it exists $j \in I$, s. t. $\gamma \notin P_j$ which means it exists i s. t. for every $\gamma' \in DSeq$, $\gamma|_i\gamma' \notin P_j$ which means it exists i s. t. for every $\gamma' \in DSeq$, $\gamma|_i\gamma' \notin P$. Hence P is a safety property.
3) Let $\{P_i\}_{i\in I} \in \mathcal{C}^I$, with I finite. Let $P = \bigcup_{i\in I} P_i \in \mathcal{C}$. Let $\gamma \notin \mathcal{C}$. Then for every $j \in I$, $\gamma \notin P_j$ which means it exists i_j s. t. for every $\gamma' \in DSeq$, $\gamma|_{i_j}\gamma' \notin P_j$ which means it exists $i = max_{j\in I} i_j$ s. t. for every $\gamma' \in DSeq$, $\gamma|_i\gamma' \notin P$. Hence P is a safety property.

□

Let $\mathcal{O} = \{P_O \in DProp | DSeq \setminus P_O \in \mathcal{C}\}$, which is the set of open sets.
Hence $(DSeq, \mathcal{O})$ is a topological space.
Let $\mathcal{D} = \{P_S \in DProp | P_S \text{ is a liveness property}\}$.
We will show that $\mathcal{D}$ is the set of dense sets in $(DSeq, \mathcal{O})$.

Lemma 12. $\mathcal{D} = \{P_L \in DProp | P_L \text{ is a liveness property}\}$ is a set of dense sets of $(DSeq, \mathcal{O})$, i. e. for every $P_L \in \mathcal{D}$, for every $P_O \in \mathcal{O}$, $P_L \cap P_O \neq \emptyset$.

Proof. We need to show that for every $P_L \in \mathcal{D}$, for every $P_O \in \mathcal{O}$, $P_L \cap P_O \neq \emptyset$.

Let $P_L \in \mathcal{D}$ and $P_O \in \mathcal{O}$. Let $\gamma \in P_O := DSeq \setminus P_S$ with $P_S \in \mathcal{C}$. Hence $\gamma \notin P_S$, which means $\exists \gamma|_i$ s. t. $\forall \gamma' \in DSeq, \gamma|_i\gamma' \notin P_S$ that is, $\exists \gamma|_i$ s. t. $\forall \gamma' \in DSeq, \gamma|_i\gamma' \in P_O$. Furthermore, for every γ_j, it exists $\gamma'' \in DSeq$ s. t. $\gamma_j\gamma'' \in P_L$ by definition of liveness property. Hence it can be applied to $\gamma_j = \gamma|_i$. Thus we constructed $\gamma^* = \gamma|_i\gamma'' \in P_O \cap P_L$. Thus $\mathcal{D}$ is dense in $(DSeq, \mathcal{O})$

□

Theorem 9. *Every distributed property P can be written as the intersection $P_L \cap P_S$ of a liveness distributed property P_L and a safety distributed property P_S. Formally, $\forall P \in DProp$, $\exists (P_L, P_S) \in \mathcal{D} \times \mathcal{C}$ s. t. $P = P_L \cap P_S$.*

Proof. Let $\bar{P}$ the smallest safety property containing P. Let $P_L = \neg(\bar{P} \setminus P)$. Then,
$P_L \cap \bar{P} = \neg(\bar{P} \setminus P) \cap P =$
$(\neg\bar{P} \cup P) \cap P =$
$(\neg\bar{P} \cap \bar{P}) \cup (P \cap \bar{P}) =$
$P \cap \bar{P} = P.$

Now we will show that P_L is a liveness property. To do so we show that P_L is dense. By contradiction, let assume it exists $P_0 \in \mathcal{O}$ so that $P_0 \subset \neg P_L$. Hence $P_0 \subset (\bar{P} \setminus P)$ which gives $P \subset \bar{P} \setminus P_0$. Since the intersection of two closed sets is closed, $\bar{P} \setminus P_0$ is closed. Since $\bar{P}$ is the smallest safety property containing P, $\bar{P} \setminus P_0 = \bar{P}$, which lead us to a contradiction. So P_L is a liveness property, while, by definition, $\bar{P}$ is a safety property, which ends the proof. □

Appendix E
Pseudo-Extensions

We formalise what is a pseudo extension. Intuitively, it allows to take some additional cares to send and deliver a message of the original algorithm. These precautions can lead to waiting strategies that are not allowed by an extension in the sense of [24]. Furthermore, some execution $e \in execs(\Pi)$ of the original algorithm that had a counterpart in the set of executions of an extension, can have no counterpart in the set of executions of a pseudo-extension. In this case we say that the execution is masked. Intuitively, a fault of such an execution has no counterpart in the pseudo-extension.

A. The Pseudo-Extension Formalism

Notation: Let $\bar{\Pi}$ be a distributed protocol. $\forall \bar{\alpha} \in execs(\bar{\Pi})$ $\forall p, q \in \Psi$, we note $S_p^{\bar{\alpha}} \in \mathcal{P}(\bar{\mathcal{M}}) = send(\bar{\alpha}|_p)$ the set of messages that have been sent by p in $\bar{\alpha}$ and $S^{\bar{\alpha}} = \bigcup_{p\in\Psi} S_p^{\bar{\alpha}}$. In the same manner, we note $R_q^{\bar{\alpha}} \in \mathcal{P}(\bar{\mathcal{M}}) = received(\bar{\alpha}|_p)$ the set of messages that have been received by q in $\bar{\alpha}$ and $R^{\bar{\alpha}} = \bigcup_{q\in\Psi} R_q^{\bar{\alpha}}$. Finally we note $F_p^{\bar{\alpha}} = \{\bar{m} \in S_p^{\bar{\alpha}} | @q \in \Psi, dest(\bar{m}) = q, \bar{m} \in R_q^{\bar{\alpha}}\}$ and $F^{\bar{\alpha}} = \bigcup_{p\in\Psi} F_p^{\bar{\alpha}}$ the set of "in flight" messages.

Definition 21 (Pseudo-Extension). $(\bar{\Pi}, \Pi, \mu_m^i, \tilde{\mu}_m^i, \mu_m^o, \mu_s, XO)$ is called a pseudo-reduction of distributed protocol $\bar{\Pi} = (\bar{\mathcal{S}}, \bar{s}^0, \bar{\mathcal{M}}_A, \bar{\mathcal{I}}, \bar{\mathcal{O}}, \bar{\mathcal{T}})$ to a protocol $\Pi = (\mathcal{S}, s^0, \mathcal{M}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ iff

μ_m^o, μ_m^i are total maps $\mathcal{P}(\bar{\mathcal{M}}) \to \mathcal{P}(\mathcal{M})$, $\tilde{\mu}_m^i$ is a total map $\mathcal{P}(\bar{\mathcal{M}}) \times \bar{\mathcal{S}} \to \mathcal{P}(\mathcal{M})$, μ_s is a total map $\bar{\mathcal{S}} \to \mathcal{S}$, where $\mu_m^i, \tilde{\mu}_m^i, \mu_m^o, \mu_s$ are polynomially computable ($\langle m \rangle_{\sigma_i}$ cannot be obtained from $\langle \bar{m} \rangle_{\sigma_j}$ if the syntactic representation of $\bar{m}$ does not contain the syntactic representation of $\langle m \rangle_{\sigma_i}$) and the following conditions hold:

- X1 $\bar{\mathcal{I}} = \mathcal{I}$, that is, Π accepts the same terminal inputs as $\bar{\Pi}$;
- X2 $\bar{\mathcal{O}} = \mathcal{O} \cup XO$ and $\mathcal{O} \cap XO = \emptyset$, that is, Π produces the same terminal outputs as $\bar{\Pi}$, except XO; (Typically, XO can contain fault or suspecting notification)
- X_3 $\mu_s(\bar{s}^0) = \bar{s}^0$
- X_4' (message correspondence) $\forall m \in \mathcal{M} \exists (\underline{\bar{m}}^i, \underline{\bar{m}}^o) \in \mathcal{P}(\bar{\mathcal{M}})^2 : \mu_m^i(\underline{\bar{m}}^i) = \mu_m^o(\underline{\bar{m}}^o) = \{m\}$, that is, every message of Π has at least one set of messages, counterpart in $\bar{\Pi}$ for sending (o for output) and another for receiving (i for input);
- X_5 (state correspondence) a) $\forall s \in \mathcal{S}, \exists \bar{s} \in \bar{\mathcal{S}} : \mu_s(\bar{s}) = s$, that is, every state of Π has at least one counterpart in $\bar{\mathcal{S}}$.
- X_6' (Homomorphism for correct processes) $\forall \bar{s_1}, \bar{s_2} \in \bar{\mathcal{S}}, \bar{mi}, \bar{mo} \subseteq \bar{M}, ti \subseteq \mathcal{I}, to \subseteq \bar{\mathcal{O}}$,
  $[\bar{\mathcal{T}}(\bar{s_1}, \bar{mi} \cup ti) = (\bar{s_2}, \bar{mo} \cup to)] \implies$
  $[\mathcal{T}(\mu_s(\bar{s_1}), \tilde{\mu}_m^i(\bar{mi} \cup ti, \bar{s_1})) = (\mu_s(\bar{s_2}), \mu_m^o(\bar{mo} \cup (to \setminus XO)))]$
  Let us note that if $\tilde{\mu}_m^i(\bar{mi} \cup ti, \bar{s_1}) = \emptyset$, X_6 implies $(\mu_s(\bar{s_2}), \mu_m^o(\bar{mo} \cup (to \setminus XO))) = (s_1, \emptyset)$.
- X_7 For every $\bar{\alpha} \in execs(\bar{\Pi})$, for every valid behavior $\bar{\beta}_i = \bar{\alpha}|_i$, s. t. $\bar{\gamma} = (i, \bar{I}, \bar{O}) = levent(\bar{\beta}_i)$ $\mu_m^o(\bar{O}) = \mu_m^o(S_i^{\bar{\alpha}} \cup \bar{O}) \setminus \mu_m^o(S_i^{\bar{\alpha}})$.
- X_8 For every $\bar{\alpha} \in execs(\bar{\Pi})$, for every valid behavior $\bar{\beta}_i = \bar{\alpha}|_i$ s. t. $\bar{s} = lstate(\bar{\beta}_i)$ and $\bar{\gamma} = (i, \bar{I}, \bar{O}) = levent(\bar{\beta}_i)$, $\tilde{\mu}_m^i(\bar{I}, \bar{s}) \subset \mu_m^i(R_i^{\bar{\alpha}} \cup \bar{I}) \setminus \mu_m^i(R_i^{\bar{\alpha}}) \subset \mu_m^i(R_i^{\bar{\alpha}} \cup \bar{I})$ (This can be deduced from computability of each mapping that ensures integrity against a bounded adversary.)

If there exists at least one pseudo-reduction from an algorithm $\bar{\Pi}$ to an algorithm Π, we say that $\bar{\Pi}$ is a pseudo-extension of Π.

The condition X_6' is different from the condition X_6 of the extension definition of [24]. Indeed, the way a message is interpreted depends on the state where a message is received. But the condition X_8 claims that, even according to this special interpretation $\tilde{\mu}_m^i$, we cannot receive more messages at state s that we received in all the executions according to the interpretation μ_m^i that does not take into account the state. Thus X_6' and X_8 allows waiting strategies, where we wait for additional pieces of information before considering a message. That is, the correct process can behave as if it receives the messages later.

The condition X_7 stipulates that the sent messages are sent salvo by salvo. When a correct process want to send a message m, it does not send a corresponding set $\underline{\bar{m}}^o$ into several chunks. Nothing prevent the Byzantine to do this.

For any pseudo-reduction $(\bar{\Pi}, \Pi, \mu_m^i, \tilde{\mu}_m^i, \mu_m^o, \mu_s, XO)$ we can construct a pseudo-execution mapping μ_e that maps executions $\bar{\alpha}$ of $\bar{\Pi}$ to (possibly open) executions of Π, browsing the execution $\bar{\alpha}$, and at each prefix $\bar{x}_{k+1} = \bar{x}_k || \bar{\gamma}_{k+1}$ proceeding as follows:

- 1. Start with $\alpha = \emptyset$.
- 2. For each new event $\bar{\gamma}_{k+1} = (i, \bar{I}, \bar{O})$ that occurs between the state of i $\bar{s}_k|_i = \bar{s}_u$ and $\bar{s}_{k+1}|_i = \bar{s}_v$ (if i is correct, $\bar{\mathcal{T}}(\bar{s}_u, \bar{I}) = (\bar{s}_v, \bar{O})$), perform the following steps:
  - (a)
    (a1) If i is correct, compute $I = \tilde{\mu}_{sm}^i(\bar{I}, \bar{s}_u) \subset \mu_m^i(\bar{I} \cup R_i^{\bar{x}_k}) \setminus R_i^{\bar{x}_k}$
    (a1') If i is Byzantine, compute arbitrary $I' \subseteq \mu_m^i(\bar{I} \cup R_i^{\bar{x}_k})$ (A Byzantine process cannot pretend to have received more messages than possible)
    (a2) Compute $O = \mu_m^o(\bar{O} \cup S^{\bar{x}_k}) \setminus \mu_m^o(S^{\bar{x}_k})$ (A Byzantine process cannot pretend to have not sent what it sent). We can remark that if i is correct, $O = \mu_m^o(\bar{O})$.
  - (b) Remove from I the messages b1) whose recipient is not i itself or b2) that have already been received that is remove from I any $m \in M$ with b1) $dest(m) \neq i$ or b2) $RECV(i, m) \in e$.
  - (c) Remove from O the messages that have already been sent, that is remove from O any $m \in M$ with $SEND(k, m, j) \in e$.
  - (d) For each node $j \in \Psi$, compute $O_j := \{m \in O | src(m) = j\}$.
  - (e) If $I_i \neq \emptyset$ or $O_i \neq \emptyset$, append (i, I_i, O_i) to α.
  - (f) For each $j \neq i$ with $O_j \neq \emptyset$, append $(j, \emptyset, O_j)$ to α.
  - (g)
    (g1) If i is correct, the new state reached by i is $\mu_s(\bar{s}_v)$
    (g2) Otherwise i can reach arbitrary state (with polynomially bounded memory).

In the remaining, μ_e refers to this particular mapping defined above.

B. Relevant Properties of Pseudo-Extension

Here, we define some relevant properties that can be necessary and/or sufficient to ensure the desired properties of a pseudo-extension.

a) Integrity: In a pseudo-extension $\bar{\Pi}$ of a distributed algorithm Π, if a set of messages $\underline{\bar{m}} \in \mathcal{P}(\bar{M})$ s. t. $m \in \mu^i_m(\underline{\bar{m}})$ is received with $src(m) = j$, it does not necessarily implies that a set of messages $\underline{\bar{m}}' \in \mathcal{P}(\bar{M})$ s. t. $m \in \mu^o_m(\underline{\bar{m}}')$ has necessarily been sent by j. This property is called $Integrity$. The fact that integrity is ensured or not depends on the pseudo-extension, the interleaving of the messages and the number of faulty nodes.

Definition 22 (Integrity)**.** We say that a pseudo-extension $\bar{\Pi}$ of Π preserves $integrity$ under an environment Env if , $\forall m \in I$, for every $\bar{\alpha} \in execs(\bar{\Pi})$, s. t. a) $RECV(\underline{\bar{m}}, j) \in \bar{\alpha}'$ with $\mu^i_m(\underline{\bar{m}}) = m$ and c) Env holds, then either $\exists \bar{\alpha}' \prec \bar{\alpha}$ s.t. $SEND(src(m), \underline{\bar{m}}', j) \in \bar{\alpha}'$ with $\mu^o_m(\underline{\bar{m}}') = m$ or $src(m)$ is Byzantine.

b) Obligation: In the distributed algorithm Π, if a correct process i sends a message $m \in M$ to another process j, m is eventually received. In a pseudo-extension $\bar{\Pi}$ of Π, if a set of messages $\underline{\bar{m}} \in \mathcal{P}(\bar{M})$ s. t. $m \in \mu^o_m(\underline{\bar{m}})$ is sent, it does not necessarily implies that a set of messages $\underline{\bar{m}}' \in \mathcal{P}(\bar{M})$ s. t. $m \in \mu^i_m(\underline{\bar{m}}')$ will be eventually received by j. This property is called $Obligation$. The fact that obligation is ensured or not depends on the pseudo-extension, the interleaving of the messages and the number of faulty nodes.

Definition 23 (Obligation)**.** We say that a pseudo-extension $\bar{\Pi}$ of Π preserves $obligation$ under an environment Env if , $\forall m \in O$, for every $\bar{\alpha}, \bar{\alpha}' \in execs(\bar{\Pi})$, s. t. a) $SEND(i, \underline{\bar{m}}, j) \in \bar{\alpha}'$ with $m \in \mu^o_m(\underline{\bar{m}})$. b) $\bar{\alpha}' \prec \bar{\alpha}$ c) $\bar{\alpha}$ is infinite d) Env holds, then $RECV(\underline{\bar{m}}', j) \in \bar{\alpha}$ with $\tilde{\mu}^i_m(\underline{\bar{m}}', \bar{s}_j) = m$ for every state $\bar{s}_j$ reached by j after the send of $\underline{\bar{m}}$.

c) Independence: We anticipate execution preservation by defining a useful property to easily ensure a form of execution preservation.

Definition 24 (Independence Criterion)**.** Let $(\bar{x}^{\lambda(q)}, x^q) \in execs(\bar{\Pi}) \times execs(\Pi)$. We say that $(\bar{x}^{\lambda(q)}, x^q)$ verifies independence criterion if :

1) (execution correspondence) $\mu_e(\bar{x}^{\lambda(q)}) = x^q$ (where μ_e is the attached execution mapping of definition 21),
2) (in flight correspondence)
   for each distinct message $m \in (M \cap F^{x^q})$ that is in flight in x^q ($SEND(src(m), m, dest(m)) \in x^q$ and $RECV(dest(m), m) \notin x^q$), there exists a set of messages $\underline{\bar{m}}^i \subset (\mathcal{P}(\bar{M}) \cap F^{\bar{x}^{\lambda(q)}})$ in flight in $\bar{x}^{\lambda(q)}$, s. t. $\tilde{\mu}^i_m(\underline{\bar{m}}^i, s_d) = m$ with $s_d = lstate(\bar{x}^{\lambda(q)})|_{dest(m)}$

Definition 25 (Independence)**.** Let $\bar{\Pi}$ be a pseudo-extension of Π that preserves obligation under an environment Env. We say that $\bar{\Pi}$ allows $independence$ for a $basis \subseteq execs(\Pi)$ under an environment $Env' \subset Env$ if and only if for every $(\bar{x}^{\lambda(q-1)}, x^{q-1}) \in execs(\bar{\Pi}) \times basis$ that verifies independence criterion, for every $I^q \subset F^{x^q}$ for every well-formed event $\gamma^q = (i, I^q, O^q)$ it exists $\bar{y} \in frags(\bar{\Pi})$, s. t. $(\bar{x}^{\lambda(q-1)} || \bar{y}, x^{(q-1)} || \gamma^q)$ still verifies the independence criterion and $(x^{(q-1)} || \gamma^q) \in basis$.

The idea is that a corresponding execution will be able to be built, event by event, using independence criterion as an invariant.

C. Ensured Properties

Here we show that a pseudo-extension preserving integrity, like a one obtained by our transformation τ_{scr}, preserves correctness. Moreover if it also preserves obligation and allows independence, like a one obtained by our transformation τ_{scr}, it preserves a high set of executions, namely the fully-correct one.

a) Correctness preservation: At first, we define "verbose execution" which allows more precision in the proofs.

Definition 26 (Verbose Execution)**.** A verbose execution ve is a sequence of alternating states and events :$\underline{s}^0, \gamma^1, \underline{s}^1, ..., \underline{s}^{k-1}, \gamma_k, \underline{s}^k$, with $\gamma^k = (u(k), I^k_{u(k),\eta(k)}, O^k_{u(k),\eta(k)})$ where :

- $\forall k \in [1, |ve|]$, $u(k) \in \Psi$
- $\forall k \in [1, |ve|]$, $\eta(k) \in \mathsf{N}$
- $\forall k, k' \in [1, |ve|]$, s.t. $u(k) = u(k')$, and $k < k'$: a) $\eta(k) < \eta(k')$ b) if $\eta(k)+1 < \eta(k')$, then $\exists k'' \in]k, k'[$, s.t. $u(k'') = u(k)$ and $\eta(k'') \in]\eta(k), \eta(k')[$
- $\underline{s}^k = (s^k_{p_1}, s^k_{p_2}, ..., s^k_{p_n})$. We note $\underline{s}^k \lceil i = s^k_{p_i}$

We note $ve \lceil p_i$ the verbose execution projected on the process p_i, that is after a) having selected only the events with an index k s. t. $u(k) = p_i$ b) project any s^k on the state of p_i, to obtain the state $\underline{s^k} \lceil i = s^k_{p_i}$ c) delete all the duplicate states that are not separated by an event. We note γ^k the event $(u(k), I^k_{u(k),\eta(k)}, O^k_{u(k),\eta(k)})$

Lemma 13 (Correctness Preservation)**.** Let $\bar{\Pi}$ and Π be two algorithms for which a pseudo-reduction $(\bar{\Pi}, \Pi, \mu_m^i, \tilde{\mu}_m^i, \mu_m^o, \mu_s, XO)$ exists where $\bar{\Pi}$ preserves integrity. Then, if $\bar{e}$ is an execution in which a node i is correct with respect to $\bar{\Pi}$, i is correct in $\mu_e(\bar{e})$ with respect to Π.

Proof. Our induction hypothesis is that i is correct in the prefix of $e\lceil i$ that consists of the first k events of $e\lceil i$ Assume the hypothesis holds up to $k-1$. We want to show it also holds up to k.

We note $\bar{ve} = \underline{\bar{s}}^0, \bar{\gamma}^1, \underline{\bar{s}}^1, ...,$
$\underline{\bar{s}}^{\lambda(q')-1}, \bar{\gamma}^{\lambda(q')}, \underline{\bar{s}}^{\lambda(q')}, ...,$
$\underline{\bar{s}}^{\lambda(q)-1}, \bar{\gamma}^{\lambda(q)}, \underline{\bar{s}}^{\lambda(q)}$, with
$\bar{\gamma}^{\lambda(q')} = (i, \bar{I}_{i,\xi(k-1)}, O_{i,\xi(k-1)})$ and $\bar{\gamma}^{\lambda(q)} = (i, \bar{I}_{i,\xi(k)}, O_{i,\xi(k)})$ and
$ve = \underline{s}^0, \gamma^1, \underline{s}^1, ...,$
$\underline{s}^{q'-1}, \gamma^{q'}, \underline{s}^{q'}, ...,$
$\underline{s}^{q-1}, \gamma^q, \underline{s}^q,$
with $\gamma^{q'} = (i, I_{i,k-1}, O_{i,k-1})$ and $\gamma^q = (i, I_{i,k}, O_{i,k})$
with $\mu_e(\bar{ve}) = ve$.

Let $\xi(k-1)$ be the index of the event $\bar{\gamma}^{\lambda(q')} = (i, \bar{I}^{\lambda(q')}_{i,\xi(k-1)}, \bar{O}^{\lambda(q')}_{i,\xi(k-1)})$ in $\bar{ve}\lceil i$ (with index $\lambda(q')$ in $\bar{e}$) that triggers the event $\gamma^{q'} = (i, I^{q'}_{i,k-1}, O^{q'}_{i,k-1})$ with index $k-1$ in $e\lceil i$ (with index q' in ve).

Let $\xi(k)$ be the index of the event $\bar{\gamma}^{\lambda(q)} = (i, \bar{I}^{\lambda(q)}_{i,\xi(k)}, \bar{O}^{\lambda(q)}_{i,\xi(k)})$ in $\bar{ve}\lceil i$ (with index $\lambda(q)$ in $\bar{e}$) that triggers the event $\gamma^q = (i, I^q_{i,k}, O^q_{i,k})$ with index k in $e\lceil i$ (with index q in ve).

We want to show that $(P_k) : \mathcal{T}(s^{q-1}|_i, I_{i,k}) = (s^q|_i, O_{i,k})$.

At first, by construction, $s^{q'}|_i = s^{q-1}|_i$ (*).

Since no event performed by i occurs between $\gamma^{q'}$ and γ^q (by construction), the rule f has never been triggered between the corresponding $\bar{\gamma}^{\lambda(q')}$ and $\bar{\gamma}^{\lambda(q)}$, which means that at any event $(i, \bar{I}, \bar{O})$ performed by i between the corresponding $\bar{\gamma}^{\lambda(q')}$ and $\bar{\gamma}^{\lambda(q)}$, we have $\tilde{\mu}^i_m(\bar{I}, .) = \emptyset$. Since for every state $s, \mathcal{T}(s, \emptyset) = (s, \emptyset)$, the rule X_6 allows to conclude that $\mu_s(\bar{s}^{\lambda(q)-1}|_i) = \mu_s(\bar{s}^{\lambda(q')}|_i) = s^{q'}|i$ (**) . By applying (*) to (**), we obtain $\mu_s(\bar{s}^{\lambda(q)-1}|_i) = s^{q-1}|i$ (***).

Now consider the event $\bar{\gamma}^{\lambda(q)} = (i, \bar{I}^{\lambda(q)}_{i,\xi(k)}, \bar{O}^{\lambda(q)}_{i,\xi(k)}) = (i, \bar{I}, \bar{O})$ with the index $\xi(k)$ in $\bar{ve}|_i$. We note $\bar{I}^m = \bar{I} \cap \bar{M}$, $\bar{I}^t = \bar{I} \cap \mathcal{I}$, $\bar{O}^m = \bar{O} \cap \bar{\mathcal{M}}$, $\bar{O}^t = \bar{O} \cap \mathcal{O}$ and $\bar{O}^x = \bar{O} \cap XO$.

We recall that $(i, I_{i,k}O_{i,k})$ is the associated event constructed from $(i, \bar{I}, \bar{O})$ via μ_e.

Since i is correct, $(i, I_{i,k}O_{i,k})$ cannot be produced by another process via rule f. Indeed, if it would be the case, the concerned message $\langle m \rangle_{s_i}$ would have been signed by i itself. So another message $\bar{m}$, storing $\langle m \rangle_{s_i}$ would have been already sent by i and the rule (c) would apply before (f).

We want to show that $\mathcal{T}(s^{q-1}|i, I_{i,k}) = (s^q|i, O_{i,k})$

To do so we use the rules of μ_e to show:

1) $\mu_s(\bar{s}^{\lambda(q)-1}|_i) = s^{q-1}|i$, already shown (***)
2) $\tilde{\mu}^i_m(\bar{I}^{\lambda(q)}_{i,\xi(k)}, \bar{s}^{\lambda(q)-1}|_i)) = I_{i,k}$. This comes from the rule (a1) (the messages removed in rule (b) are ignored in $\mathcal{T}$ since the recipient is not i)
3) $\mu^o_m(\bar{O}^{\lambda(q)}_{i,\xi(k)}) = O_{i,k}$. This comes from the rule (a2) and the hypothesis X_7.
4) $\mu_s(\bar{s}^{\lambda(q)}|_i) = s^{q-1}|i$. This comes from the rule (g1)

Finally we use the condition X_6 to conclude.

□

Also, we show that a pseudo-extension preserving integrity implies an homomorphic transformation according to definition 12.

Lemma 14. Let $\bar{\Pi}$ be a pseudo-extension of distributed protocol Π, for every execution $\bar{\alpha}_2$ of $\bar{\Pi}$ being a prefix of execution $\bar{\alpha}_1$ of $\bar{\Pi}$, $\mu_e(\bar{\alpha}_2)$ is a prefix of $\mu_e(\bar{\alpha}_1)$

Proof. Follows from the way μ_e is constructed (events are always appended, never removed). □

Lemma 15. Let $\bar{\Pi}$ be a pseudo-extension of Π preserving integrity, then $(\bar{\Pi}, \Pi, \mu_e)$ is an homomorphic transformation.

Proof. Correctness preservation is ensured by lemma 13 if integrity is preserved, prefix ordering preservation is ensured by lemma 14. Finally, the condition X'_6 of pseudo-reduction is clearly a special case of homomorphic transition. Thus the three conditions of definition 12 are satisfied. □

b) Execution preservation:

Lemma 16 (Execution Preservation)**.** Let $\bar{\Pi}$ be a pseudo-reduction of Π. Let Env an environment so that a) the pseudo-extension preserves the obligation and the integrity properties of Π, b) every node is benign c) the pseudo-extension allows independence for $basis$. Then for every execution $\alpha \in basis$, there exists an execution $\bar{\alpha}$ of $\bar{\Pi}$ such that

- a) $\mu_e(\bar{\alpha}) = \alpha$ (modulo duplicate messages sent by faulty nodes in α)
- b) $i \in Corr_{\bar{\Pi}}(\bar{\alpha}) \iff i \in Corr_{\Pi}(\alpha)$

Proof. We use independence criterion (assumed to be verified for $\alpha = \emptyset$, $\bar{\alpha} = \emptyset$) as invariant. Then we construct $\bar{\alpha}$ from α recursively event by event. □

D. Preservation of Safety and Liveness Properties

Lemma 17 (Preservation of Safety)**.** Let Π be a distributed algorithm and $\bar{\Pi}$ be a pseudo-extension of Π preserving integrity. Let P be a safety distributed property ensured by Π with t_0'-resiliency, then for every execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t_0'$, $\mu_e(\bar{\alpha})$ does not violate P.

Proof. The safety is preserved because of correctness preservation. Indeed , $i \in corr(\bar{\alpha}, \bar{\Pi}) \implies i \in Corr_{\Pi}(\alpha = \mu_e(\bar{e}))$. Thus $|corr_{\Pi}(\alpha = \mu_e(\bar{\alpha}))| \geq |corr_{\bar{\Pi}}(\bar{\alpha})| \geq n - t_0'$. So for every execution $\bar{\alpha}$ with $|corr(\bar{\Pi}, \bar{\alpha})| \geq n - t_0'$, safety is preserved since the P is ensured with t_0'-resiliency by the original algorithm Π.

□

Lemma 18 (Preservation of Safety and Liveness)**.** Let Π be a distributed algorithm and $\bar{\Pi}$ be a pseudo-extension ensuring both integrity and obligation. Let P be a distributed property ensured by Π with t_0'-resiliency. Then, the following hold:

- If P is a safety distributed property of Π, then for every execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t_0'$, $\mu_e(\bar{\alpha})$ does not violate P.
- If P is a liveness distributed property of Π and $t_0' \leq f$, with factor $f = \mathsf{min}(\lceil n/3 \rceil - 1, t_0)$ then for every infinite execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t_0'$, $\mathsf{gen}(\mu_e(\bar{\alpha}), \Pi)$ is in P.

Proof. The safety is preserved because of lemma 17. Let show that liveness is also preserved.

Let $\bar{\alpha}$ be an infinite execution of $\bar{\Pi}$ corresponding to an execution $\alpha = \mu_e(\bar{\alpha})$ of Π. We want to show that: either α is infinite and contains the send and reception of an infinite set of messages from correct processes or e is finite but every correct process reached a state so that the liveness property is ensured if they stay at this state forever. Then, since safety is preserved and Π ensure liveness, the liveness will be preserved. We show it by contradiction.

Let us assume α contains a finite number of messages received by correct nodes. Let α_1 the prefix of α where the last from-correct-to-correct message is received. Let $\bar{\alpha}_1$ be a prefix of $\bar{\alpha}$, s. t. $\mu_e(\bar{\alpha}_1) = \alpha_1$. If the liveness property is not in ensured in α_1, and no messages from correct to correct is in flight, we can remove all the in flight messages from α_1 to obtain α_1' where a deadlock occured. Thus Π does not ensure liveness which would lead to a contradiction. Thus it exists at least one from-correct-to-correct message m in flight in α. Thus it exists a set $\underline{\bar{m}}^o$ of messages that have been sent in $\bar{\alpha}_1$, s. t. $m \in \mu_m(\underline{\bar{m}}^o)$. Because of obligation, a set $\underline{\bar{m}}^i$, s. t. $m \in \mu_m^i(\underline{\bar{m}}^i)$ will be eventually delivered by correct process in $\bar{\alpha}$. Finally m will be eventually delivered in $\mu_e(\bar{\alpha})$. Here we got the contradiction.

□

Appendix F
Secure-Broadcast-based transformation

A. Secure-Broadcast-Based Pseudo-Extension

Here, we formalise our transformation.

1) hierarchical composition: Our solution is the result of the hierarchical composition of several modules. We present the hierarchical composition and then the different modules. Finally, we present the result of the composition which is our transformation. The choice of composition is only didactic, the definition of the transformation is self-content but it can be easier to understand it as the composition of simple modules.

Definition 27 (Compatibility)**.** Let $\Pi_A = (\mathcal{S}_A, s_0^A, \mathcal{M}_A, \mathcal{I}_A, \mathcal{O}_A, \mathcal{T}_A)$ and $\Pi_B = (\mathcal{S}_B, s_0^B, \mathcal{M}_B, \mathcal{I}_B, \mathcal{O}_B, \mathcal{T}_B)$. Π_B is *pluggable* on Π_A iff $\mathcal{M}_A \subseteq \mathcal{I}_B$ and $\mathcal{O}_B \subseteq \mathcal{M}_A$.

The intuition is that B need to be able to accept a message sent by A as a terminal input ($\mathcal{M}_A \subseteq \mathcal{I}_B$), and each terminal output of B has to be interpreted as a delivered message by A ($\mathcal{O}_B \subseteq \mathcal{M}_A$).

Definition 28 (Hierarchical Composition)**.** Let $\Pi_A = (\mathcal{S}_A, s_0^A, \mathcal{M}_A, \mathcal{I}_A, \mathcal{O}_A, \mathcal{T}_A)$ and $\Pi_B = (\mathcal{S}_B, s_0^B, \mathcal{M}_B, \mathcal{I}_B, \mathcal{O}_B, \mathcal{T}_B)$, s. t Π_B is *pluggable* on Π_A. We define $\Pi_A \otimes \Pi_B = (\mathcal{S}_{A\otimes B}, s_0^{A\otimes B}, \mathcal{M}_{A\otimes B}, \mathcal{I}_{A\otimes B}, \mathcal{O}_{A\otimes B}, \mathcal{T}_{A\otimes B})$ such that:

- $\mathcal{M}_{A\otimes B} = \mathcal{M}_B$

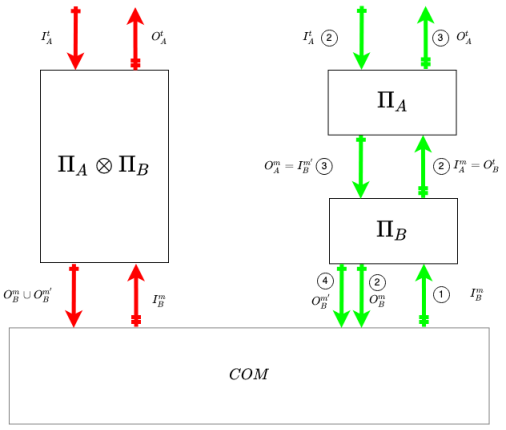

Fig. 3: Hierarchical composition: each intermediate set of messages represented by a circle with a number k is computed from the ones with a number $k-1$

- $\mathcal{I}_{A\otimes B} = \mathcal{I}_A$
- $\mathcal{O}_{A\otimes B} = \mathcal{O}_A$
- $\mathcal{S}_{A\otimes B} = \mathcal{S}_A \times \mathcal{S}_B$ (cartesian product)
- $s_0^{A\otimes B} = (s_0^A, s_0^B)$
- $\mathcal{T}_{A\otimes B}((s_A, s_B), in)$ is computed as follows:
  1) $(s'_B, out_B) := \mathcal{T}_B(s_B, in)$
  2) $out_B^t = out_B \cap \mathcal{O}_B$, $out_B^m = out_B \cap \mathcal{M}_B$
  3) $(s'_A, out_A) := \mathcal{T}_A(s_A, out_B^t)$
  4) $out_A^t = out_A \cap \mathcal{O}_A$, $out_A^m = out_A \cap \mathcal{M}_A$
  5) $(s''_B, out'_B) = \mathcal{T}_B(s'_B, out_A^m)$
  6) $\mathcal{T}_{A\otimes B}((s_A, s_B), in) = ((s'_A, s''_B), out_A^t \cup out_B \cup out'_B)$

2) The modules: Now we present the different modules. We stress that the verification module presented in definition 33 is slightly different from the "original" transformation presented in algorithms 1 and 3

Secure-Broadcast: A secure-broadcast is the multi-shot version of Reliable-broadcast. This is a communication primitive that ensure Integrity, Uniformity, Obligation and Source Order, as long $f < t$ and Integrity and Source Order even if $f > t$.

Definition 29. A secure-broadcast algorithm: Let M be a set of (potentially signed) messages. A secure-broadcast algorithm for M is an algorithm $A_{scr}(\mathcal{M}) = (\mathcal{S}_{src}, s_0^{src}, \mathcal{M}_{src}, \mathcal{I}_{src}, \mathcal{O}_{src}, \mathcal{T}_{src})$ such that

- $\mathcal{I}_{src} = \{secure\text{-}bcast(\langle m, s\rangle_{\sigma_{src(m)}}) | q \in \Psi, m \in M, s \in \mathsf{N}\}$,
- $\mathcal{O}_{src} = \{secure\text{-}deliver(\langle m, s\rangle_{\sigma_{src(m)}}) | q \in \Psi, m \in M, s \in \mathsf{N}\}$,
- $\forall m \in \mathcal{M}$, $\exists \underline{m}^i \subset \mathcal{M}_{src}$, s. t. the reception of $\underline{m}^i$ triggers $secure\text{-}deliver(\langle m, s\rangle_{\sigma_q})$
- $\forall m \in \mathcal{M}$, $\exists \underline{m}^o \subset \mathcal{M}_{src}$ the reception of $secure\text{-}bcast(\langle m, s\rangle_{\sigma_q})$ triggers the send of $\underline{m}^o$.
- Each execution of $A_{scr}(\mathcal{M})$ ensures Integrity, Uniformity, Obligation and Source Order:
  - (Integrity) If a correct process p secure-deliver a message $\langle m, s\rangle_{\sigma_q}$ with q correct, then q has secure-broadcast $\langle m, s\rangle_{\sigma_q}$.
  - (Uniformity) If a correct process p secure-delivers a message $\langle m, s\rangle_{\sigma_q}$, then every correct process eventually secure-delivers $\langle m, s\rangle_{\sigma_q}$. Furthermore, no correct process secure-delivers $\langle m', s\rangle_{\sigma_q}$ with $m' \neq m$.
  - (Obligation) If a correct process p secure-broadcasts a message $\langle m, s\rangle_{\sigma_q}$, then every correct process eventually secure-delivers $\langle m, s\rangle_{\sigma_q}$.
  - (Source Order) If a correct process secure-delivers $\langle m, s\rangle_{\sigma_q}$ then p already secure-delivered a message $\langle m', s'\rangle_{\sigma_q}$ for every $s' \in [1, s[$. A correct process never secure-delivers $\langle m, s\rangle_{\sigma_q}$ and $\langle m', s\rangle_{\sigma_q}$ with $m \neq m'$

Consistent Causal Past Justification: The aim of theses sequence of definitions is to introduce the valid predicate $Valid_\Pi$, that accepts a set of messages only if it represents a fully-consistent execution of the distributed algorithm Π.

At first we define executions *matching* a set of messages $\underline{m}$, that is an execution that only handles messages in $\underline{m}$.

Definition 30 (Matching Executions)**.** Let Π be a distributed algorithm with $\mathcal{M}$ as set of messages. Let $\underline{m} \in \mathcal{P}(\mathcal{M})$. We note $Execs_\Pi(\underline{m}) \subset Execs_\Pi$ the set of fully-correct execution that only handle messages in $\underline{m}$, no more, no less, that is $\forall \alpha \in Execs_\Pi(\underline{m}), send(\alpha) = \underline{m}$. An execution $\alpha \in Execs_\Pi(\underline{m})$ is said to be *matching* $\underline{m}$.

Then we define consistent causal past justification (CCPJ) of a message m, which is a set of messages $\underline{m}$, s. t. it exists a fully-correct execution matching $\underline{m}$ that justifies the send of m.

Definition 31 (Consistent Causal Past Justification)**.** Let Π be a distributed algorithm with $\mathcal{M}$ as set of messages. A consistent causal past justification (CCPJ) of $m \in \mathcal{M}$ is a set of messages $\underline{M'} = \underline{M} \cup I$, so that $\exists \alpha \in Execs_\Pi(\underline{M})$, $O \in \mathcal{P}(\mathcal{M})$ and $\alpha' \in Execs_\Pi(\underline{M} \cup I \cup O)$, so that $\alpha' = \alpha||(i, I, O)$ with $m \in O$.

The set of CCPJs of m for an algorithm Π is noted $CCPJ_\Pi(m)$

Finally, we can define the valid predicate $Valid_\Pi$.

Definition 32 (Valid Predicate (for the valid check))**.** Let Π be a distributed algorithm with $\mathcal{M}$ as set of messages. We note $Valid_\Pi : \mathcal{P}(\mathcal{M}) \to \{true, false\}$ the predicate so that for every $log \in \mathcal{P}(\mathcal{M})$, $Valid_\Pi(log)$ returns $true$ if and only if for every $m \in log$, $\exists log' \subset log$ so that $log' \in CCPJ_\Pi(m)$. This definition does not precise if this predicate can be computed in a decent time-frame for all distributed algorithms.

This predicate plays the role of a filter after the secure-delivery of a certain set of messages. This set will be able to be taken into account only if it is validated by the the valid predicate $Valid_\Pi$. Hence, a correct process will reach state $\bar{s}$ of the transformation corresponding to a state of the original algorithm s only if it stored a fully-correct execution e of the original algorithm that justifies this state s, and so regardless the number of Byzantine processes. We call this property valid-enabling.

Definition 33. Let $\Pi = (\mathcal{S}_\Pi, s_0^\Pi, \mathcal{M}_\Pi, \mathcal{I}_\Pi, \mathcal{O}_\Pi, \mathcal{T}_\Pi)$ a distributed algorithm. A valid-check algorithm for Π, noted $V(\Pi)$ is an algorithm $(\mathcal{S}_v, s_0^v, \mathcal{M}_v, \mathcal{I}_v, \mathcal{O}_v, \mathcal{T}_v)$, such that:

- $\mathcal{I}_v = \{valid\text{-}bcast(m) | m \in \mathcal{M}_\Pi\}$,
- $\mathcal{O}_v = \{valid\text{-}deliver(m) | m \in \mathcal{M}_\Pi\}$,
- $\mathcal{M}_v = \{secure\text{-}bcast(\langle m, s\rangle_{\sigma_q}), secure\text{-}deliver(\langle m, s\rangle_{\sigma_{src(m)}}) | m \in \mathcal{M}_\Pi\}$
- each state in $\mathcal{S}_v$ stores an incremental sequence number sn starting at 0 and a register r with the attributes $r.validated$ and $r.scr\text{-}delivered$ with $type(r.validated) = type(r.scr\text{-}delivered) = \mathcal{P}(\mathcal{M}_\Pi)$ and $type(sn) = \mathsf{N}$
- $\mathcal{T}_v(s_v, in)$ is computed as follows :
  - $in^t = in \cap \mathcal{I}_v$, $in^m = in \cap \mathcal{M}_v$
  - If $|in^t| = k > 0$ and $in^t = \{valid\text{-}bcast(m_0), ..., valid\text{-}bcast(m_{k-1})\}$, $out_1 = \{scr\text{-}bcast(\langle m_k, \sigma_v.sn + k\rangle_{\sigma_q}) | valid\text{-}bcast(m_k) \in I^t\}$, otherwise $out_1 = \emptyset$.
  - $r'.scr\text{-}delivered = r.scr\text{-}delivered \cup$ $\{\langle m, s\rangle_{\sigma_{src(m)}} | scr\text{-}deliver(\langle m, s\rangle_{\sigma_{src(m)}}) \in in^m\}$
  - $r'.validated = r.validated \cup scr_{max}$ where scr_{max} is the maximal subset of $r'.scr\text{-}delivered$ s. t. $valid_\Pi(log)$ with $log = \{m | \langle m, s\rangle_{\sigma_{src(m)}} \in r.validated \cup scr_{max}\ \}$
  - $sn' = sn + |in^t|$
  - $out_2 = r'.validated \setminus r.validated$
  - $\mathcal{T}_v(s_v, in) = (s'_v, out_1 \cup out_2)$ with s'_v the evaluation of r' and sn'.

a) Adaptor: The role of the adaptor is allowing a hierarchical composition between the original algorithm and the valid-check module composed with the secure-broadcast module.

Definition 34 (Adaptor)**.** Let $\Pi = (\mathcal{S}_\Pi, s_0^\Pi, \mathcal{M}_\Pi, \mathcal{I}_\Pi, \mathcal{O}_\Pi, \mathcal{T}_\Pi)$ be a distributed algorithm. An adaptor for Π is an algorithm $A_{adpt}(\Pi) = (\mathcal{S}_{adpt}, s_0^{adpt}, \mathcal{M}_{adpt}, \mathcal{I}_{adpt}, \mathcal{O}_{adpt}, \mathcal{T}_{adpt})$ s. t.

- $\mathcal{M}_{adpt} = \{valid\text{-}bcast(m), secure\text{-}deliver(m) | m \in M_\Pi\}$
- $\mathcal{I}_{adpt} = \mathcal{O}_{adpt} = \mathcal{M}_\Pi$
- $\mathcal{S}_{adpt} = \{s_0^{adapt}\}$
- $\mathcal{T}_{adapt}(s, in)$ is performed as follows:
  - $in^t = in \cap \mathcal{I}_{adpt}$ and $in^m = in \cap \mathcal{M}_{adpt}$
  - $out_1 = \{valid\text{-}bcast(m) | m \in out^t\}$ and $out_2 = \{m | valid\text{-}deliver(m) \in in^m\}$
  - $\mathcal{T}_{adpt}(s, in) = (s, out_1 \cup out_2)$

3) The transformation τ_{scr} obtained by composition of our modules: The transformation consists in secure-broadcasting every message that are normally sent in Π and receiving a message m' only of it has been both secure-delivered and justified by a fully-correct execution via the valid predicate.

Definition 35 (Secure Transformation τ_{scr})**.** For every distributed algorithm Π, we note $\tau_{scr}(\Pi)$ the algorithm $\bar{\Pi} = \Pi \otimes A_{adpt}(\Pi) \otimes V(\Pi) \otimes A_{scr}(\mathcal{M}_\Pi)$

We explicit the obtained algorithm so that the definition 35 is self-contained and does not depend on the hierarchical composition definition:

- $\bar{\mathcal{M}}_{\bar{\Pi}} = \mathcal{M}_{scr}(\Pi)$
- $\bar{\mathcal{I}}_{\bar{\Pi}} = \mathcal{I}_{\Pi}$ and $\mathcal{O}_{\bar{\Pi}} = \mathcal{O}_{\Pi}$
- $\bar{\mathcal{S}}_{\bar{\Pi}} = \mathcal{S}_{\Pi} \times \mathcal{S}_{adpt}(\Pi) \times \mathcal{S}_{V}(\Pi) \times \mathcal{S}_{scr}(\Pi)$
- $\bar{s}_0^{\bar{\Pi}} = (s_0^{\Pi}, s_0^{adpt}(\Pi), s_0^{V}(\Pi), s_0^{scr}(\Pi))$

We explicit the obtained transition function
$\bar{\mathcal{T}}_{\bar{\Pi}}(\bar{s}^1 = (s_\pi^1, s_{adpt}^1, s_V^1, s_{scr}^1), \bar{in}) :$

1) $\bar{in}^m = \bar{in} \cap \bar{\mathcal{M}}_{\bar{\Pi}}$ and $\bar{in}^t = \bar{in} \cap \bar{in}$
2) $\mathcal{T}_{scr}(s_{scr}^1, \bar{in}^m) = (s_{scr}^2, \bar{out}_{scast} \cup \bar{in}_{sdel})$ with $\bar{out}_{scast} \subset \mathcal{M}_{scr}$ and $\bar{in}_{sdel} \subset \mathcal{O}_{scr}$
3) $\mathcal{T}_v(s_v^1, \bar{in}_{sdel}) = (s_v^2, \bar{in}_{vdel})$ with $\bar{in}_{vdel} \subset (\mathcal{O}_v \cap \mathcal{M}_{adpt})$ and $s_v^2.valid\text{-}delivered = s_v^1.valid\text{-}delivered \cup \bar{in}_{vdel}$
4) $\mathcal{T}_{adpt}(s_{adpt}^1, \bar{in}_{vdel}) = (s_{adpt}^2, \bar{in}_{adpt})$ with $\bar{in}_{adpt} \subset \mathcal{O}_{adpt}$
5) $\mathcal{T}_{\Pi}(s_\pi^1, \bar{in}_{adpt} \cup \bar{in}^t) = (s_\pi^2, IO_{react} \cup \bar{out}^t)$ with $\bar{out}^t \subset \bar{\mathcal{O}}_{\bar{\Pi}}$ and $IO_{react} \subset \mathcal{M}_{\Pi}$
6) $\mathcal{T}_{adpt}(s_{adpt}^2, IO_{react}) = (s_{adpt}^3, \bar{in}_{vcast})$; $\mathcal{T}_v(s_v^2, \bar{in}_{vcast}) = (s_v^3, \bar{in}_{scast})$; $\mathcal{T}_{scr}(s_{scr}^2, \bar{in}_{scast}) = (s_{scr}^3, \bar{out}_{react})$
7) $\bar{\mathcal{T}}_{\bar{\Pi}}(\bar{s}^1 = (s_\pi^1, s_{adpt}^1, s_V^1, s_{scr}^1), \bar{in}) =$
   $(\bar{s}^2 = (s_\pi^2, s_{adpt}^3, s_V^3, s_{scr}^3), \bar{out}_{scast} \cup \bar{out}_{react} \cup \bar{out}^t)$

The messages IO_{react} of step 5 is an output of the module which represents the original algorithm and is interpreted as an input by the adaptor that will trigger the secure-broadcast of IO_{react}. The obligation of secure-broadcast ensure that every message of IO_{react} will be eventually (secure) delivered by the other correct process. We say that τ_{scr} preserves obligation, which is defined in definition 23 of appendix E.

The messages in_{adpt} of step 5 have been secure-delivered. The integrity property of the secure-broadcast ensures that each message in_{adpt} has been "sent" (secure-broadcast) by the appearing source if it is correct. We say that τ_{scr} preserves integrity, which is defined in definition 22 of appendix E.

In the appendix F, via some results of the appendix E, we show at lemma 24 that τ_{scr} ensures the same properties of safety and liveness of the original algorithm as long as the number of faulty processes is bounded by $n/3$. This is intuitive since, nothing changed excepting that each message is secure-broadcast and that a message has to be secure-delivered to be taken into account.

The messages in_{adpt} of step 5 have been validated. Every time the module Π visits a sequence of states $\vec{s} = (s^0, s^1, ..., s^n)$, the module $V(\Pi)$ stores in its register r, at attribute $r.validated$, a fully-correct execution of Π that justifies the state $\vec{s}$. We call this property, *valid enabling*.

Because valid enabling and integrity are ensured even if $f > t$, if a set of correct processes reach a non-safe state $\underline{s} = (s_1, ..., s_k)$, they respectively store a fully-correct execution α_k that justifies s_k. We show in lemma 27 in appendix F, that these executions are enough to compute a proof of culpability against $t + 1$ Byzantine processes.

This is intuitive since otherwise, it would mean that t or less Byzantine processes can tackle the safety property of the original algorithm.

B. The Transformation τ_{scr} Ensures Integrity, Obligation and Allows Independence

Lemma 19 (τ_{scr} generates Pseudo-Extension)**.** For every distributed algorithm Π, the algorithm $\bar{\Pi} = \tau_{scr}(\Pi) = \Pi \otimes A_{adpt}(\Pi) \otimes V(\Pi) \otimes A_{scr}(\Pi)$ obtained via secure-broadcast-based transformation is a pseudo-extension of Π.

Proof.

- X_1 and X_2 are immediate
- X_3 and $X_5 : \bar{\mathcal{S}}_{\bar{\Pi}} = \mathcal{S}_{\Pi} \times \mathcal{S}_{adpt}(\Pi) \times \mathcal{S}_{V}(\Pi) \times \mathcal{S}_{scr}(\Pi)$. We note $\mu_s : (s_\pi, s_1, s_2, s_3) \in \bar{\mathcal{S}}_{\bar{\Pi}} \to s_\pi \in \mathcal{S}_{\Pi}$.
- X_4 μ_m^i and μ_m^o are defined with mapping function of the secure-broadcast
- X_6 Let $\bar{\mathcal{T}}_{\bar{\Pi}}(\bar{s}^1 = (s_\pi^1, s_{adpt}^1, s_V^1, s_{scr}^1), \bar{in}) =$
  $(\bar{s}^2 = (s_\pi^2, s_{adpt}^3, s_V^3, s_{scr}^3), \bar{out}_{scast} \cup \bar{out}_{react} \cup \bar{out}^t)$ as explicated in step 7 of transition function of the transformation.
  We need to verify $\alpha(\mu_s(\bar{s}^1), \mu_{sm}^i(\bar{I}, \bar{s}^1)) = (\mu_s \bar{s}^2), \mu_m^o(\bar{out}_{scast} \cup \bar{out}_{react} \cup \bar{out}^t))$
  The step 5 and μ_s definition ensures the correspondence for the first component.
  Then we have in_{adpt} (step 5) that corresponds to the new valid-delivered messages. We can define $\tilde{mu}_{sm}^i(\bar{in}^m, \bar{s}) = in_{adpt}$ where in_{adpt}, is computed with the step 2, 3 and 4.
  Finally $\mu_m^o(\bar{out}_{scast} \cup \bar{out}_{react} \cup \bar{out}^t)) = \mu_m^o(\bar{out}_{react} \cup \bar{out}^t))$, since $\bar{out}_{scast}$ does not contain new secure-broadcasted messages. Then $\bar{out}^t$ is given immediately by $\mathcal{T}_{\Pi}$ (step 5). Thereafter $\bar{out}_{react}$ to ti_{react} with the secure broadcast transformation μ_m^o.

Finally every condition is ensured. □

Lemma 20 (τ_{scr} Preserves Integrity)**.** Let Π be a distributed protocol. The secure-broadcast-based pseudo-extension $\bar{\Pi} = \tau_{scr}(\Pi)$ preserves $Integrity$ as long as the adversary is computationally bounded.

Proof. Immediate with the integrity property of secure-broadcast, which holds as long as the adversary is computationally bounded. □

Lemma 21. $(\tau_{scr}(\Pi), \Pi, \mu_e)$ is an homomorphic transformation.

Proof. We know that $\tau_{scr}(\Pi)$ is a pseudo-extension by lemma 19, that ensures integrity by lemma 20. Then we can apply the lemma 15. □

Lemma 22 (τ_{scr} Preserves Obligation)**.** Let Π be a distributed protocol. The secure-broadcast-based pseudo-extension $\bar{\Pi} = \tau_{scr}(\Pi)$ preserves $Obligation$ as long as the $f < t$.

Proof. Let $Env = \{(f < t)\}$. Let $m \in O$, $\bar{\alpha}, \bar{\alpha}' \in execs(\bar{A})$, s. t. a) $SEND(i, \underline{\bar{m}}, j) \in \bar{\alpha}'$ with $\mu^o_m(\underline{\bar{m}}) = m$. b) $\bar{\alpha}' \prec \bar{\alpha}$ c) $\bar{\alpha}$ is infinite d) Env holds.

Since $\underline{\bar{m}}$ has been sent by a correct process i, i secure-delivered a consistent causal past justification of m, note $ccpj$. Because of the uniformity property of secure-broadcast, which is ensured as long as $f < t$, $ccpj$ and m will be eventually secure-delivered by all correct nodes. Since $ccpj$ is a consistent causal past justification of m, both $ccpj$ and m will be validate by all correct processes, namely $j = dest(m)$.

Finally, $RECV(\underline{\bar{m}}', j) \in \bar{\alpha}$ with $\mu^i_m(\underline{\bar{m}}') = m$. □

Definition 36 (FIFO Execution)**.** An execution α is FIFO if for every messages (m, m') with same source and same destination ($src(m) = src(m')$ and $dest(m) = dest(m')$), if m' is sent after m, then m' is not received before m.

Lemma 23 (τ_{scr} Allows Independence)**.** Let Π be a distributed protocol. The secure-broadcast-based pseudo-extension $\bar{\Pi} = \tau_{scr}(\Pi)$ allows $independence$ for fully-correct FIFO execution as basis as long as $f \leq t$ and the adversary is bounded.

Proof. We assume the independence criterion holds for $(x^{(q-1)}, \bar{x}^{\lambda(q-1)})$ (noted $IC(q-1)$).

Let $I^q \subset F^{x^{q-1}}$ s. t. for every $m \in I_q$, $dest(m) = i$. Let $O^q \in \mathcal{P}(M)$, s. t. $\gamma^q = (i, I^q, O^q)$ and $(x^{(q-1)}||\gamma^q)|_i$ is a correct behavior.

By assumption, every message $m \in I_q$ is ready to be delivered, that is it exists a strict causal past justification $J(m)$ of m that has been valid-delivered by i (1) (otherwise, the assumption would not hold).

For every message in m, for every message $m'' \in J(m)$, s. t. $dest(m'') = j$, with j correct, j has valid delivered m'' (2). Indeed, if was not the case, j would not have valid-broadcast $m''' \in J(m)$, triggered by m'' and $J(m)$ would not be complete.

We note $J = \bigcup_{m \in I^q} J(m)$.

We build $\bar{y}_1$ s. t. every message in $J(m)$ is valid-delivered by every correct process (3). This is possible because of uniformity and (1). Moreover, $\mu_e(\bar{y}_1) = \emptyset$ because of (2).

Then we build $\bar{y}_2$ where every message in I is valid-delivered by every correct process except i (4). Here again this is possible because of (3) and uniformity of secure-broadcast and here again $\mu_e(\bar{y}_2) = \emptyset$ since the correct nodes valid-deliver messages whose they are not the recipient.

Then $\bar{\gamma} = (i, \bar{I}, \bar{O})$ occurs s. t. i valid delivers $I^q = \tilde{\mu}^i_m(\bar{I}, lstate(\bar{x}^{\lambda(q-1)}||\bar{y}_1||\bar{y}_2))$. This is possible because of $(IC(q-1))$.

Then we build $\bar{y}_3$ so that for every $m' \in O^q$, $\underline{\bar{m}'}^i$ are in flight after $\bar{y}_3$, while no additional message have been sent in $\mu_e(\bar{y}_3) = \emptyset$. The secure-broadcast allows this. Since i is correct, $J \cup I^q$ is causal past justification for O^q (5). Moreover $J \cup I^q$ has been valid delivered by every correct process (6) because of (3) and (4). Thus (5) and (6) implies that every message in O^q is ready to be delivered by the corresponding recipient (7).

We note $\bar{y} = \bar{y}_1||\bar{y}_2||\bar{\gamma}||\bar{y}_3$.

We have $((x^{(q-1)}||\gamma^q), \bar{x}^{\lambda}(q-1)||\bar{y})$ that verifies the first independence criterion condition of $(IC(q))$ because $\mu_e(\lambda(q-1)||\bar{y}) = \mu_e(\lambda(q-1))||\mu_e(\bar{y}_1)||\mu_e(\bar{y}_2||\bar{\gamma}||\mu_e(\bar{y})_3) = \mu_e(\lambda(q-1))||\mu_e(\bar{\gamma}) = x^{q-1}||\gamma^q$. Moreover $((x^{(q-1)}||\gamma^q), \bar{x}^{\lambda}(q-1)||\bar{y})$ verifies the second independence criterion condition of $(IC(q))$ because of (7) and $(IC(q))$.

Thus, the secure-broadcast base pseudo-extension allows independence.

□

C. Properties of τ_{scr}

1) properties preservation: Here we show that τ_{scr} still ensures the safety and liveness properties of the original algorithm.

Lemma 24 (Property Preservation of τ_{scr})**.** Let Π be a distributed algorithm and $\bar{\Pi} = \tau_{scr}(\Pi)$. Let P be a distributed property ensured by Π with t'_0-resiliency. Then, the following hold:

- If P is a safety distributed property of Π, then for every execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t'_0$, $\mu_e(\bar{\alpha})$ does not violate P.
- If P is a liveness distributed property of Π and $t'_0 \leq f$, with factor $f = \mathit{min}(\lceil n/3 \rceil - 1, t_0)$ then for every infinite execution $\bar{\alpha}$ of $\bar{\Pi}$ with $|Corr_{\bar{\Pi}}(\bar{\alpha})| \geq |\Psi| - t'_0$, $\mathit{gen}(\mu_e(\bar{\alpha}), \Pi)$ is in P.

Proof. The secure-broadcast based transformation ensures integrity regardless the number of Byzantine nodes, thus lemma 17 can be applied. Moreover, it ensures obligation as long as the number of Byzantine nodes is bounded by $n/3$, thus lemma 18 can be applied. □

2) Accountability: Here we show that τ_{scr} ensures accountability.

Lemma 25 (Justification)**.** Let $\bar{\alpha}$ be an execution of $\bar{\Pi}$ such that $\alpha = \mu_e(\bar{\alpha})$. For every correct process $i \in Corr_{\bar{\Pi}}(\bar{\alpha})$ that visited the sequence of states $\overrightarrow{s}_i = (s_i^1, ..., s_i^k)$ during its corresponding behavior $\beta_i = \alpha|_i$, i stores a fully-correct execution α' that justifies $\overrightarrow{s}_i$, i. e. $\beta_i = \alpha'|_i$ and $corr_\Pi(\alpha')$.

Proof. By construction of valid-check module, valid enabling is preserved even if $f > t$. □

Lemma 26 (Disseminated Proof)**.** Let $\bar{\alpha}$ be an execution of $\bar{\Pi}$ such that $\alpha = \mu_e(\bar{\alpha})$ violates a safety distributed property P of Π and let Π ensure P with t'_0-resiliency. then there exists $i, j \in C$, s. t. the union of their valid-delivered log contains at least $t'_0 + 1$ pairs of mutant messages from as many different Byzantine processes.

Proof. Because of lemma 25, it exists a set of correct processes $C \subset Corr_{\bar{\Pi}}(\bar{\alpha})$, s. t. for every $i \in C$, i stores a fully-correct execution α_i justifying the sequence of visited states $\overrightarrow{s}_i = (s_i^0, ..., s_i^n)$ during α_i. By contradiction, we assume the lemma no to be true. We will build an execution $\alpha' \in execs(\Pi)$ that would violate the safety distributed-property with less than t'_0 Byzantine faults, which will be in contradiction with t'_0-resiliency of Π for safety property P. For every correct $i \in C \subset Corr_{\bar{\Pi}}(\bar{\alpha})$, we have $Corr_\Pi(\alpha_i) = \Psi$ because of valid enabling. We note $\beta_i = \alpha_i|_i$, the behavior that i should have in the execution α_i of the original algorithm. We fix B, $|B| \leq t'_0$ that will play the role of Byzantine processes. We note $K = \Psi \setminus B$ (Korrect). For every (really) correct process k in K, for every fully-correct behavior stored , the sent messages are not conflicting and respect a common-prefix, that is $\forall i, j, k \in K, \beta_{k,i} = \alpha_i|_k$, $\beta_{k,j} = \alpha_j|_k$, either $\beta_{k,i} \leq \beta_{k,j}$ or $\beta_{k,j} \leq \beta_{k,i}$.

We construct α^* as follows: The Byzantine processes in B behave with each correct i as they did in execution α_i, using the behavior $\beta_{b,i} = \alpha_i|_b$ for each $b \in B$. In α^*, each correct process $i \in K$ behaves as they did in α_i, using $\beta_i = \alpha_i|_i$. Since every member of K has the same behavior in each α_i, nothing change in the communication between members of K. Then each correct process i visit the sequence of states $\overrightarrow{s}_i = (s_i^0, ..., s_i^n)$ corresponding to the behavior β_i so that α^* violates the safety property. But this is no possible since we assumed the original algorithm ensure P with t'_0-resiliency and $|corr_\Pi(\alpha^*)| = |K| \geq |\Psi| - t$. Thus there is a contradiction which prove that the assumption was not true. Finally, we conclude that at least $t'_0 + 1$ mutant messages from as many Byzantine processes are stored in the union of a pair of fully-correct executions (α_i, α_j) stored by two correct processes $i, j \in C$. □

When two correct nodes i and j reach states s_i and s_j, they store fully-correct execution $e_i, e_j \in execs(\Pi)$ that justify s_i and s_j. A way to centralise the potential proofs of culpability is to broadcast e_i after the decision. In fact, each message $\langle m, sn_q \rangle_q$ secure-delivered by process i, will be echoed to j (and vice-versa). If process j stores the message (instead of ignoring it because it already secure-delivered $\langle m', sn_q \rangle_q$ with $m' \neq m$), the correct processes will be able to centralise the proof. We slightly abuse the notation τ_{scr} to refer to the extension of τ_{scr} where mutant messages are stored.

Lemma 27 (Accountability)**.** Let $\bar{\alpha}$ ($|corr_{\bar{\Pi}}(\bar{\alpha})| \geq 2$) be an execution of $\bar{\Pi}$ such that $\alpha = \mu_e(\bar{\alpha})$ violates a safety distributed property P of Π and let Π ensure P with t'_0-resiliency, then every correct process eventually stores $t'_0 + 1$ pair of mutant messages from as many Byzantine processes.

Proof. The pairs of mutant messages are stored in the union of log of two correct processes according to lemma 26. Thus these two correct processes just have to broadcast their log to everybody. □

3) syntactic correspondence:

Lemma 28 (Syntactic Correspondence)**.** Let Π be a distributed algorithm. Let $\bar{\Pi} = \tau_{scr}(\Pi)$. Then, the following hold:
- Let $\bar{\alpha}$ be an execution of $\bar{\Pi}$. If a process $p \in \Psi$ is correct in $\bar{\alpha}$, then p is correct in $\mu_e(\bar{\alpha})$.
- For every fully-correct FIFO execution α, there exists an execution $\bar{\alpha}$ of $\bar{\Pi}$ such that $\alpha = \mu_e(\bar{\alpha})$.

Proof. Let Π be a distributed algorithm. We know that $\tau_{scr}(\Pi)$ is a pseudo-extension of Π according to lemma 19. Then we know, it preserves integrity according to lemma 20 and obligation according to lemma 22. Additionally, we know that it allows independence according to lemma 23. Then, we can apply the lemma 13 and 16. This conclude the proof. □

The lemma 28 shows that our transformation is not arbitrary, since the pseudo-extension solves the problem with the same way as the original one.

Theorem 10 (Generic Accountable Transformation)**.** *Let Π be a non-synchronous t_0-resilient distributed protocol . Then, $\tau_{scr}(\Pi)$ is a general accountable counterpart of Π with factor $f = \mathsf{min}(\lceil n/3 \rceil - 1, t_0)$ according to basis that consists in the set of fully-correct FIFO executions. Furthermore, if Π solves a task $\mathcal{D}$, then $\tau_{scr}(\Pi)$ is an accountable counterpart of Π with factor $f = \mathsf{min}(\lceil n/3 \rceil - 1, t_0)$.*

Proof. $\tau_{scr}(\Pi)$ is a general accountable counterpart of Π by conjunction of lemma 21, 24, 27 and 28. Furthermore, if Π solves a task $\mathcal{D}$, it is an accountable counterpart of Π by theorem 8 □

Appendix G
τ_{scr} Application To Randomized Distributed Protocols

The Theorem 10 also holds for randomized distributed protocols if:

- safety is ensured *deterministically*, i.e., if a randomized distributed protocol Π_{rand} ensures a safety distributed property P with t_0-resiliency, then there does not exist an execution of Π_{rand} with less than $t_0 + 1$ faulty processes that violates P (e.g., [6]), and
- any liveness distributed property does *not* require the existence of private channels (e.g., private channels are required for liveness in [9]).

This section is devoted to the application of τ_{scr} to such randomized distributed protocols.

A. Preliminaries

A discrete probabilistic space over a countable set S is denoted by $(S, 2^S, \eta)$, where η is a discrete probability measure, that is, $\eta(\emptyset) = 0$, for each $C \subset S, \eta(S) = \sum_{c \in C} \eta(\{c\})$ and $\eta(S) = 1$. We define $\mathsf{Disc}(S)$ to be the set of discrete probability measures on S. In the remainder of the section, we omit the set notation for a measure of a singleton set. For a discrete probability measure η on a set S, $\mathsf{supp}(\eta)$ denotes the support of η, that is, the set of elements $s \in S$ such that $\eta(s) \neq 0$.

B. Randomized Distributed Protocols

A randomized protocol $\Pi_p = (\mathcal{S}_p, s_0^p, \mathcal{M}, \mathcal{I}_p, \mathcal{O}_p, \mathcal{RT}_p)$ is assigned to a process $p \in \Psi$, where $\mathcal{S}_p$ represents a set of countable states p can take (we refer to $\mathcal{S}_p$ as the state set of Π_p), $s_0^p \in \mathcal{S}_p$ is an initial state of p, $\mathcal{M}_p$ is a countable set of messages p can send or receive, $\mathcal{I}_p$ is a countable set of internal events p can observe, $\mathcal{O}_p$ is a countable set of internal events p can produce and $\mathcal{RT}_p : \mathcal{S}_p \times P(\mathcal{M}_p \cup \mathcal{I}_p) \to \mathsf{Disc}(\mathcal{S}_p \times P(\mathcal{M}_p \cup \mathcal{O}_p))$, that is, the new state, the events produced and the new messages sent are randomly chosen according to the discrete probability law.

Since the next state is not deterministically computed, we extend the notion of events given in §III. A *randomized event* is an element $(p, I, s, O) \in \Psi \times \mathcal{I}_p \times \mathcal{S}_p \times \mathcal{O}_p$. We denote by $\mathsf{REvents}(\Pi_p)$ the sent of randomized events of Π_p. We say that $\beta_p = (p, I_1, s_1, O_1), (p, I_2, s_2, O_2), ...$ is *valid* according to Π_p if and only if it conforms to the assigned protocol Π_p, for all $i \geq 1$, $(s_i, O_i) \in \mathsf{supp}(\mathcal{RT}_p(s_{i-1}, I_i))$ with $s_0 = s_0^p$. We denote by $\mathsf{RExecs}(\Pi)$ the set of randomized executions of a randomized distributed protocol $\Pi = (\Pi_p, \Pi_q, ..., \Pi_z)$ and by $\mathsf{RExecs}(\Pi, t_0)$ the set of randomized executions with at most t_0 Byzantine processes. A randomized distributed protocol Π ensures a safety distributed property P *determinstically* (or with probability 1) with t_0-resiliency if and only if no execution $\alpha \in \mathsf{RExecs}(\Pi, t_0)$ violates P.

C. τ_{scr} Obtains Accountability

We now prove that τ_{scr} obtains accountability for randomized distributed protocols which (1) ensure safety deterministically, and (2) do not require private channels for liveness.

a) Ensuring Safety: It is easy to see that lemma 17 can be applied to randomized distributed protocol that ensures safety deterministically. The proof is the same. Then, integrity is still trivially preserved by the transformation because of the use of signature by the source, thus the lemma 20 is still true and hence the safety part of lemma 24 is still true.

b) Ensuring Liveness: As long as the actual number of Byzantine processes is less than or equal to $n/3$, the obligation property of τ_{scr} is ensured (see Lemma 22). Thus, any liveness property P_L ensured in a randomized distributed protocol Π_{rand} is preserved in $\tau_{scr}(\Pi_{rand})$. Importantly, knowledge of certain exchanged messages cannot help the adversary to tackle liveness, since Π_{rand} (by assumption) does not require private channels to ensure liveness.

c) Ensuring Accountability: The following lemma shows that correct processes store enough information to allow for accountability. Therefore, whenever a safety distributed property is violated, correct process simply need to exchange their information, thus ensuring accountability.

Lemma 29 (Randomized Disseminated Proof)**.** Let Π_{rand} be a randomized distributed protocol that ensures a safety distributed property P with t_0-resiliency. Let $\bar{\alpha}$ be an execution of $\bar{\Pi}_{rand}$ such that $\alpha = \mu_e(\bar{\alpha})$ violates P, where

$\bar{\Pi}_{rand} = \tau_{scr}(\Pi_{rand})$. Then, there exist processes i and j that are correct in $\bar{\alpha}$ such that the union of their valid-delivered log contains at least $t_0 + 1$ pairs of conflicting messages from as many distinct Byzantine processes.

Proof. The proof is identical to the proof of Lemma 26. □

D. Second Look at The "Private Channels" Assumption

This section informally discusses a simple way to ensure accountability for a majority of practical randomized distributed protocols that *do* rely on private channels to ensure liveness. A majority of such randomized distributed protocols require that messages have to be kept secret for a single round and have the following structure:

- Round 1: Each process i sends a message $m^1_{ij} = (\mathsf{ROUND}^1, \mathsf{content}^1_{ij})$ to each process j through a private channel.
- Round 2: Upon the reception of a set S^1_j of messages of the form m^1_{kj} from a set K^1_j of $n - t_0$ processes, each process j computes and sends $m^2_{j\ell}$ to process ℓ, for every process ℓ.

Our informal accountability transformation changes the original protocol in the following manner:

- Round 1: Each process i secure-broadcasts a message $\bar{m}^1_{ij} = (\mathsf{ROUND}^1, \mathsf{cypher}(\mathsf{content}^1_{ij}, pk'_j)$ to each process j, where $\mathsf{cypher}(\mathsf{content}^1_{ij}, pk'_j)$ can only be read by j (a nonce nonce^1_{ij} can be added if the content is too small). Hence, the adversary cannot adapt its messages to the content of the messages in private channels.
- Round 2: Upon the secure-delivery of a set $\bar{S}^1_j$ of messages of the form $\bar{m}^1_{kj}$ from a set K^1_j of $n - t_0$ processes, each process j a) computes $\mathsf{reveal}^{12}_{ij} = \mathsf{content}^1_{ij}$ (with the potential nonce nonce^1_{ij}, if needed), and b) computes $m^2_{j\ell}$, for every process ℓ. Then, the process secure-broadcasts the revelation $R^1_j = (\mathsf{reveal}^{12}_{ij})_{i \in K^1_j}$, and, finally, secure-broadcasts every message of the form $m^2_{j\ell}$. The messages $m^2_{j\ell}$ can be secure-delivered after the secure-delivery of the revelation which can be compared with the secure-delivered messages in $\bar{S}^1_j$.

Appendix H
τ_{scr} Adaption For Committee-Based (Permissionless) Blockchains

In this section, we show how our τ_{scr} transformation can be adapted for committee-based blockchains, i.e., for permissionless distributed protocols. The aim of such distributed protocol traditionally is to obtain a sub-quadratic communication complexity in order to ensure scalability. Therefore, a suitable accountable version of these distributed protocols should remain sub-quadratic.

A. Committee-Based Blockchains

We assume a distributed protocol Π_{comm} for a committee-based blockchain. Π_{comm} operates among a (dynamic) set Ψ of $|\Psi| = n$ processes with t (out of n) processes being Byzantine. During an execution of Π_{comm}, committees are randomly elected through a sub-quadratic subprotocol $\Pi^{election}_{comm}$ (e.g., [28]) with $o(n^2)$ communication complexity that ensures safety and liveness with high probability as long as $t < n/3$.

Let a committee K with $|K| = s = O(\mathsf{log}(n))$ processes be elected in an execution $\alpha \in \mathsf{execs}(\Pi_{comm})$. Committee K is supposed to executed a distributed subprotocol $\Pi^{K,decision}_{comm}$ that solves a decision task with $(s/3)$-resiliency before outputting the sequence of "decided" values to the rest of the system through a distributed subprotocol $\Pi^{K,diffusion}_{comm}$. We assume a mildly adaptive adversary that is not able to corrupt a new group of processes in less time than needed for an election of a new committee [33]. Moreover, we assume that the expected communication complexity of $\Pi^{K,decision}_{comm}$ is $\tilde{O}(s^2)$. For example, protocols presented in [30], [37] solve randomized multi-value Byzantine agreement and Byzantine lattice agreement, respectively, with such complexity.

Finally, the communication complexity of Π_{comm} is $\chi(\Pi_{comm}) = \chi(\Pi^{K,decision}_{comm}) + \chi(\Pi^{K,diffusion}_{comm}) + \chi(\Pi^{election}_{comm}) = \tilde{O}(s^2) + O(s \cdot n) + o(n^2) = o(n^2)$.

B. General Accountable Counterpart

a) Problem: If the processes are unlucky, it is possible for newly elected committee K to contain $t_K > 2s/3$ Byzantine processes. If the non-elected processes (i.e., processes from the $\Psi \setminus K$) are passive, i.e., they do not communicate to each other (except during an execution of $\Pi^{election}_{comm}$), it is impossible to avoid a "non-accountable" bi-simulation done by the committee [34]. However, if non-elected processes are able to communicate, then there exists a solution.

b) Solution: We create a distributed protocol $\bar{\Pi}_{comm}$ as follows. We demand K to run $\bar{\Pi}^{K,decision}_{comm} = \tau_{scr}(\Pi^{K,decision}_{comm})$ such that (1) the dynamic randomized secure broadcast primitive is used [21], [22], and (2) clients are able to deliver messages from the secure broadcast primitive. Note that the diffusion is performed by the secure broadcast primitive, i.e., $\bar{\Pi}^{K,decision}_{comm}$ replaces both $\Pi^{K,decision}_{comm}$ and $\Pi^{K,decision}_{comm}$. Hence, either all correct clients share a consistent common view or all correct clients eventually store a proof of culpability of (at least) $s/3$ Byzantine processes[20].

[20]Recall that we use a secure broadcast primitive that allows duplication (in order to allow accountability).

The obtained communication complexity of $\bar{\Pi}_{comm}$ is $\chi(\bar{\Pi}_{comm}) = \chi(\bar{\Pi}_{comm}^{K,decision}) + \chi(\bar{\Pi}_{comm}^{election}) = \tilde{O}(s^2) \cdot O(n\log(n)) + o(n^2) = \tilde{O}(n\log^3(n)) + o(n^2) = o(n^2)$. Hence, $\bar{\Pi}_{comm}$ is a sub-quadratic distributed protocol for a committee-based blockchain that, as long as the number of Byzantine processes is bounded by $n/3$, ensures accountability whenever safety is violated due to a corrupted committee.

Importantly, we assume that $t < n/3$ in order to (1) be able to agree on a committee with a sub-quadratic distributed protocol, and (2) be able to ensure duplication of the secure broadcast primitive with a sub-quadratic distributed protocol, while allowing $t_K > 2s/3$.

Appendix I
τ_{scr} Application To PBFT

In this section, we present how our τ_{scr} transformation is applied to PBFT [10]. First, we present the preliminaries of the PBFT protocol (Appendix I-A). Then, we present an example of an execution of PBFT and we show how our τ_{scr} transformation provides the accountability guarantees in this case (Appendix I-B)

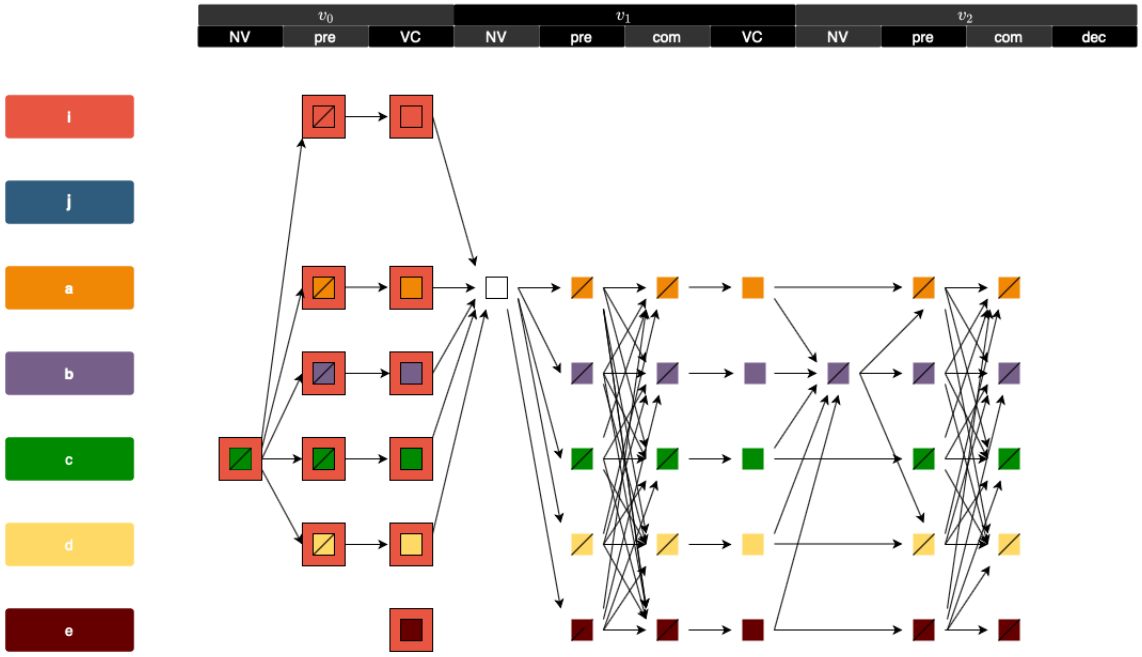

Fig. 4: Messages secure-delivered by process i: Process i cannot valid-deliver messages sent by processes a, b, c, d, e (even though i has previously secure-delivered them) until it receives the NEW-VIEW message from a.

A. Preliminaries of PBFT

For the simplicity of the presentation, we use the original notation of Castro and Liskov [10] in the rest of this section.

In PBFT, the replicas move through a succession of configurations called *views* that are numbered consecutively. In a view, one replica is the *primary* and the others are *backups*. View changes are carried out whenever the primary is suspected to have failed.

When the primary of the view v receives a client request m, it assigns a sequence number sn to this request and starts a three-phase protocol to atomically broadcast the request to the replicas. The three phases are *pre-prepare*, *prepare*, and *commit*. If, at the second phase of a view v, a process i receives enough messages supporting the message m for sequence number sn, it satisfies the predicate $prepared(m, v, sn, i)$, which allows it to broadcast a (m, v, sn)-commit message to every process. If a process receives $n - t_0$ (m, v, sn)-commit messages with the same triplet (m, v, sn), then it commits the message m for the sequence number sn irrevocably, which is denoted by the predicate $committed\text{-}local(m, v, sn, j)$. PBFT uses a garbage collector and snapshots to reclaim memory. However, we consider, for simplicity, that PBFT is given infinite memory in order to illustrate our example without any issues potentially raised by the garbage collector (e.g., a "part of" a proof of misbehavior has been garbage-collected, thus accountability cannot be achieved).

The view change is scheduled as follows: a backup maintains a timer at each view. If the backup timer expires in view $v - 1$, the backup starts a view change to move the system to view v, stops accepting messages (other than CHECKPOINT, VIEW-CHANGE and NEW-VIEW messages) and broadcasts a (v, P)-VIEW-CHANGE message to all replicas, where P is a

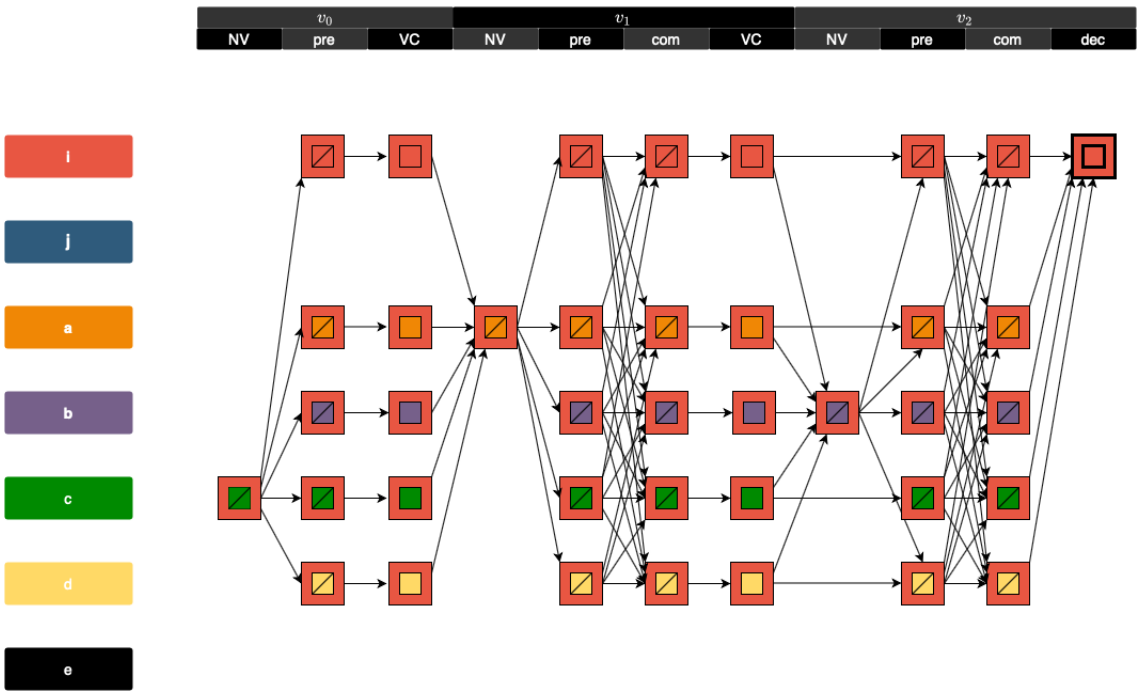

Fig. 5: Messages secure-delivered by process i: Process i valid-delivers all the messages secure-delivered in Figure 4 once it receives the NEW-VIEW message from a. Moreover, i commits the *cmd* command.

set containing the messages that the backup prepared in the past. If message $m \in P$, we say that the backup *propagated* m from view $v-1$ to view v. Otherwise, if it is faulty, it may omit to include these messages in P. When the new primary designated for view v receives $n-t_0$ valid VIEW-CHANGE messages for view v from other replicas, it broadcasts a (v, V, O)-NEW-VIEW message to all other replicas, where V is a set containing the valid (v, P)-VIEW-CHANGE messages received by the primary and O contains the set of messages propagated to the view v. Before committing any message at view v, a process waits for $(n-t_0)$ VIEW-CHANGE messages for all the views preceding v.

If a primary receives a set V of $n-t_0$ VIEW-CHANGE omitting m for sequence number *sn*, it can build a NEW-VIEW message, including these set V, to pre-prepare the command cmd^{null} at the sequence number *sn*. If cmd^{null} is committed at the sequence number *sn*, it means that no command has to be executed at this sequence number.

B. Example of an execution of PBFT

We first introduce the legend that explains how appendix I and fig. 6 should be interpreted. Then, we give an example of an execution.

a) Legend: Each process is represented by a color. A small square represents a message which is secure-delivered by a corresponding process. If a small square is framed with a frame, then the message is also valid-delivered (i.e., received at the state-machine level) by the process. We use different colors of frames to emphasize that the commands carried by the messages are different (we use the orange frame for *cmd* and the blue frame for *cmd′*). If the square is crossed by a line, then the crossed square represents a message that is broadcast in the original algorithm.

b) Example of an execution: Let $\Psi = \{i, j, a, b, c, d, e\}$ be the set of processes in the system. The leader (i.e., the primary) of view v_0 is c. The leader of view v_1 is a and the leader of view v_2 is b.

We take a closer look at the log of secure-delivered messages (the *secureDelivered* set from Algorithm 1) and the log of valid-delivered messages (the *validDelivered* set from Algorithm 1) of process i (figs. 4 and 5). From Figure 5 we see that processes from the $Q_i^0 = \{i, a, b, c, d\}$ set prepare the command *cmd*, but do fail to commit the command. Hence, they propagate the prepared command *cmd* to the leader of the next view v_1, which is process a. Process a is not successful in ensuring that *cmd* is committed in view v_1. Thus, *cmd* is propagated (by processes a, b, c, d, e) to the leader of view v_2 - process b. Finally, process b ensures that process i commits *cmd* in view v_2.

c) Accountability: Let process j commit a command *cmd′*, where *cmd′* $\neq$ *cmd* (the log of valid-delivered messages of j is presented in Figure 6) in the same execution. Recall that process i commits the *cmd* command (due to its log of valid-delivered messages from Figure 5).

Given Figure 6, we conclude that all the processes from the $Q_j^0 = \{j, a, b, c, e\}$ set has prepared the cmd' command. Since i commits cmd, it is clear that there are more than $t_0 < n/3$ faulty processes in the system (otherwise, the disagreement would not occur). All process from the $G = Q_i^0 \cap Q_j^0 = \{a, b, c\}$ are faulty since they prepared two different commands in view v_0 (i.e., they equivocated). Moreover, the proof of culpability for every process that belongs to G is stored at the union of the valid-delivered logs of i and j. Once they indeed exchange their logs, all processes from the set G become detected.

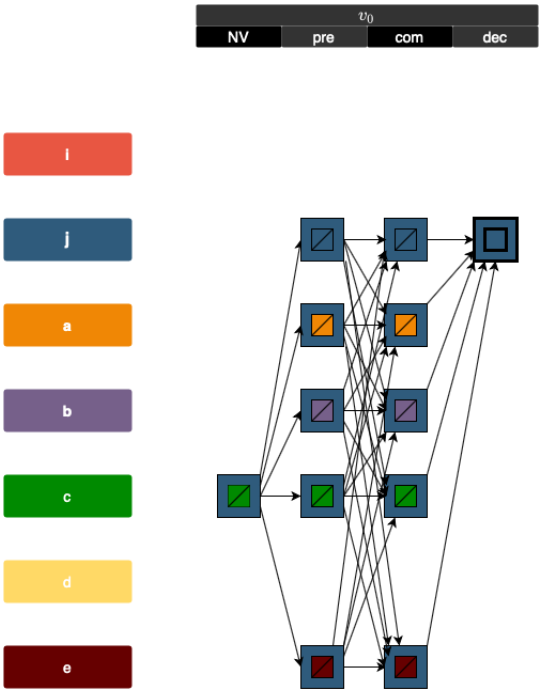

Fig. 6: Process j commits the cmd' command.