Crime and Punishment in Distributed Byzantine Decision Tasks

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 [24]. Similar results apply to set agreement [8] or lattice agreement [1]. 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 [19]. 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 [9], [28], 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 [2], [5], [13], [15], [21], [22] 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 [5] 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 full version of the paper [12].

II. Related Work

a) Byzantine failures: If a process deviates from a prescribed protocol, it commits a Byzantine failure [24]. The primary technique in tackling Byzantine failures in distributed computing is *masking*, i.e., hiding the effects of these failures [7], [16], [25], [29], [30]. An alternative approach is *detection* of Byzantine failures. Initially, detection of Byzantine failures was incorporated into the design of Byzantine *failure detectors* [17], [23], [26], which were used for solving the consensus [24] problem. Kihlstrom *et al.* [23] 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 [23] and ours: there exists a faulty behavior that we classify as a commission fault, which is not captured by the definition from [23]. Furthermore, Haeberlen *et al.* [20] studied the problem of generic fault detection in distributed systems. They, as the authors of [23], recognize commission faults as a separate class of Byzantine failures. The definition of commission faults given in [20] 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 [20]; our definition classifies such a behavior as a commission fault. The authors of [20] 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 [19], a generic accountability add-on for distributed systems. The definition of "detectably faulty" processes given in [19] 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 [2], [5], [13], [15], [21], [22]. 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 [19] or can be demanded only upon some serious safety violations [10]. 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 [19]. 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 [10] 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 [16]. Casper [6] is an accountability overlay for blockchain systems. Ways to obtain accountability guarantees for specific "PBFT-like" consensus protocols are proposed in [28]. The authors of [28] 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 [11]; however, it only works for protocols where the decision of all processes is expected to be identical. The technique used in [11] 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 [8], lattice agreement [1]). Hence, the transformation presented in [11], 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 $sender(m) \in \Psi$ and a unique receiver $receiver(m) \in \Psi$. 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 $sender(m) = p$ 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 $receiver(m) = p$ 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 $sent(\beta)$ (resp., $received(\beta)$) 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 $sender(m)$ 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 $Corr_\Pi(\alpha)$ 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 $execs(\Pi)$.

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 [18].

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 the sake of simplicity, we slightly abuse the notation and

[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.

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 [10], [11], 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:

- $sender(m) = p$, 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 $\mu_e : execs(\bar{\Pi}_{\mathcal{D}}) \rightarrow execs(\Pi_{\mathcal{D}})$ that satisfies the following:[5] li

- *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 $execs(\Pi_{\mathcal{D}}, t_0)$ 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 $\alpha \in basis$, where $basis \subseteq execs(\Pi_{\mathcal{D}}, t_0)$, 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 [20]. 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 [3], [4], [14], [27]. However, to the best of our knowledge, Definition 3 and the aforementioned

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

μ_e formalism [20] 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 $sender(m) = p$. 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 $\mathcal{J}_m = received(\beta_p)$, 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 $sender(m) = p$. Let β_p be a finite behavior such that $m \mapsto \beta_p$. The *message justification* of m is the $received(\beta_p)$ 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 [23]). 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]

[6]Note that conflicting messages do not necessarily have the same header (as is the case in [23]). This represents the very subtle difference between our definition of equivocation faults and the definition presented in [23].

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 $m \in sent(\beta_p)$ in α if and only if there exists a message $m' \in sent(\beta_p)$ 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 $m \in sent(\beta_p)$ 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 [19].

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 $m \in sent(\beta_p)$ 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.

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

[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.

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;
- 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

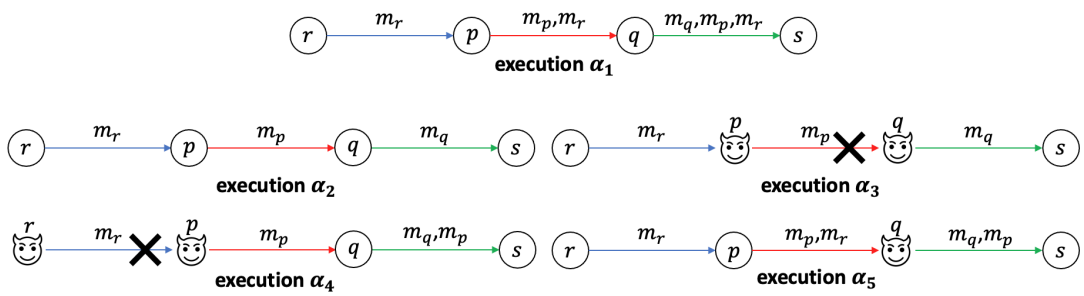

Fig. 1: In execution α_2, process q behaves correctly and sends m_q *without* sending m_p. In execution α_3, process q commits an evasion fault with respect to m_q (note that q cannot send m_p since it has not received m_p). However, these two executions are indistinguishable to process s and, hence, 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.

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 $f = min(\lceil n/3 \rceil - 1, t_0)$, 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 $f = min(\lceil n/3 \rceil - 1, t_0)$ (§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 $current_p$ behavior with each process $p \in \Psi$; initially, $current_p$ 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 $current_p$ 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) $current_p \leftarrow \beta_p$, 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. 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).

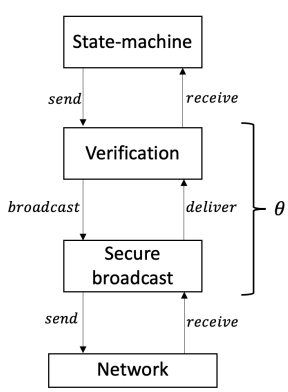

Fig. 2: Overview of the τ_{scr} transformation

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 $receiver(m) = q$, 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 [12].

Algorithm 1 Pseudocode of τ_{scr} - process p

1: **upon** INIT:
2: $seqNum \leftarrow 1$
3: $delivered \leftarrow \emptyset$ ▷ secure-delivered messages
4: $validated \leftarrow \emptyset$ ▷ validated messages
5: $next \leftarrow [0]^n$
6: $receivedMeantime \leftarrow []$ ▷ array of messages
7: $current \leftarrow [\text{empty}]^n$ ▷ behaviors of processes
8: **upon** SEND(m):
9: $M \leftarrow (m, seqNum, receivedMeantime)_{\sigma_p}$
10: $seqNum \leftarrow seqNum + 1$
11: $receivedMeantime \leftarrow []$
12: SECUREBROADCAST(M)
13: **upon** SECUREDELIVER(M):
14: $delivered \leftarrow delivered \cup \{M\}$
15: ▷ The *Validated* function is defined in [12]
16: **upon exists** $(m, sn, recMeantime)_{\sigma_q} \in delivered$ such that $sn = next[q]+1 \wedge Validated(recMeantime) = true$:
17: **if** reception of $recMeantime$ after $current[q]$ results in a valid behavior β_q that sends m **then**
18: $validated \leftarrow validated \cup \{(m, sn)\}$
19: $current[q] \leftarrow \beta_q$
20: **end if**
21: **upon exists** $(m, sn)_{\sigma_q} \in validated$ such that $sn = next[q] + 1$:
22: $next[q] \leftarrow sn$
23: **if** $receiver(m) = p$ **then**
24: RECEIVE(m)
25: $receivedMeantime.append(m)$
26: **end if**
27: **upon exists** $(m, sn)_{\sigma_q}, (m', sn')_{\sigma_q} \in delivered$ such that $sn = sn' \wedge m \neq m'$:
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 $sender(m)$ 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 $f = min(\lceil n/3 \rceil - 1, t_0)$, 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 [12], we define a certain class of transformations producing *pseudo-extensions*. An important feature of the pseudo-extension formalism is that there exists a homomorphism $\mu_e : execs(\bar{\Pi}) \rightarrow execs(\Pi)$ 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; 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.

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 $\Pi_{\mathcal{D}}$. 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, 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 $min(\lceil n/3 \rceil - 1, t_0)$-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, 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 $Corr_{\Pi}(\alpha) = \Psi$, 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 [12]).

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 $f = min(\lceil n/3 \rceil - 1, t_0)$ according to a basis which consists of all fully-correct FIFO executions of $\Pi_{\mathcal{D}}$.*

Proof. For space limitations, the formal proof of the theorem is delegated to the full version of the paper [12]. □

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

Proof. We prove the theorem by using the well-known "double-echo" secure broadcast [5], 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 [12]).
- Randomized distributed protocols in which (1) safety is ensured deterministically, and (2) private channels are not required for liveness (formal treatment given in [12]).
- Sub-quadratic committee-based blockchains (formal treatment given in [12]).

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 [2], [5], [13], [15], [21], [22]. 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.

Acknowledgments

This research is supported in part under Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things", as well as Singapore MOE Grant MOE2018-T2-1-160 (Beyond Worst-Case Analysis).

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

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

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

[9] 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.

[10] 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).

[11] 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.

[12] Civit, P., Gilbert, S., Gramoli, V., Guerraoui, R., Komatovic, J., Milosevic, Z., and Serendinschi, A. Crime and Punishment in Distributed Byzantine Decision Tasks (Extended Version). *IACR Cryptol. ePrint Arch.* (2022), 121.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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).

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.