**PhD Thesis**

# Securing Distributed Business Process Model Execution

**Mads Frederik Madsen**

**Advisor: Søren Debois**
**Submitted: March 24, 2023**

IT UNIVERSITY OF COPENHAGEN

## Abstract

This PhD thesis investigates the secure execution of distributed business process models. When collaborating actors use distributed process model execution to coordinate and execute a process, they put themself at risk of foul play; malicious collaborators may give false testimony of events in the process, both their own and others, if this is to their benefit. Similarly, they may try to extract secret steps taken by their co-collaborators. In this thesis, I study security properties for preventing and discovering such malicious behaviour.

I present in this thesis 3 main results from 3 papers I have co-written during my PhD project. The papers' relevance to distributed business process execution is demonstrated in the context of the *consistency problem.* In the consistency problem, one must ensure that a process behaves as specified even when executed as distributed partitions.

The first paper shows how to utilise Trusted Execution Environments to translate Byzantine faults to omission faults in arbitrary distributed algorithms. In the setting of distributed business process execution, this translates to a method for preventing malicious collaborators from actively *lying* about which steps they have taken in the process.

The second paper considers the definitions of *equivocation* – acting maliciously different towards two or more co-collaborators – and redefines exactly what it means to prevent equivocation. We define two different kinds of non-equivocation, one strong and one weaker, which captures properties gained from known non-equivocation subsystems. These non-equivocation properties can be used to eliminate active malicious behaviour other than lying in distributed business process execution. They can also be used to make solutions to agreement problems cheaper, solutions which inherently solve the consistency problem, although at the cost of local autonomy of collaborators.

The third and last paper considers passively malicious collaborators, i.e. collaborators who attempt to cheat in the process by simply following the process and passively listen in an attempt to extract secrets. To prevent such behaviour, we define a possibilistic notion of *secrecy* of actions in processes with run-based semantics. The secrecy definition captures under which conditions a collaborator can take a step in the execution of a distributed business process, safe in the knowledge that a specific collaborator cannot know that the action was taken. We then show that this definition of secrecy is computationally hard to determine in some business process models, specifically Dynamic Condition Response graphs, and present a sufficient condition to determine secrecy for some actions as an alternative.

**Resumé**

Denne ph.d.-afhandling undersøger sikre eksekveringer af distribuerede forretningsprocesmodeller. Når aktører der samarbejder bruger, distribuerede forretningsprocessmodeller til at koordinere og udføre en proces, risikerer de, at samarbejdspartnere med ondsindede intentioner viderегiver forkerte oplysninger om hvad, der har fundet sted i systemet, hvis en sådan udlægning er til deres egen fordel. De kan ligeledes forsøge at udlede forretningshemmeligheder ud fra hvilke handlinger, deres samarbejdspartnere tager. I denne afhandling undersøger jeg sikkerhedsegenskaber, der har til formål at forhindre og afsløre ondsindet opførsel som dette.

Jeg præsenterer i denne afhandling 3 hovedresultater fra 3 respektive artikler, som jeg har skrevet under mit ph.d.-projekt. Disse artiklers relevans for eksekvering af distribuerede forretningsprocesser bliver demonstreret i kontekst af *consistency* problemet. Målet i consistency problemet er at opnå, at kompositionen af lokale del-processer og disses udførsler ikke afviger fra en global procesbeskrivelse.

Den første artikel viser, hvordan man kan bruge et Trusted Execution Environment til at oversætte Byzantinske fejl til beskedtab (*omissions*) i arbitrære distribuerede algoritmer. Ift. eksekvering af distribuerede forretningsprocesmodeller kan dette omsættes til en metode, der forhindrer ondsindede samarbejdspartnere i at *lyve*.

Den anden artikel forholder sig til definitioner af tvetydighed (*equivocation*) – at opføre sig ondsindet forskelligt overfor to samarbejdspartnere – og redefinerer præcis, hvad det betyder at forhindre tvetydighed. Tvetydighed er nært beslægtet med at lyve, men det adskiller sig ved, at aktører kan være tvetydige uden at lyve f.eks. ved at kommunikere med en mindre andel af de samarbejdspartnerne, som de burde. Vi definerer to forskellige slags utvetydighed, en stærk og en svagere, som modellerer forskellige egenskaber, fra kendte subsystemer der giver utvetydighed. Disse egenskaber kan bruges til at forhindre ondsindet opførsel, som adskiller sig fra at lyve. De kan også bruges til at gøre løsninger til enighedsproblemer (*agreement problems*) billigere, løsninger som grundlæggende løser consistency-problemet på bekostning af lokal autonomi.

Den tredje og sidste artikel forholder sig til passivt ondsindede samarbejdspartnere, dvs. samarbejdspartnere som forsøger at snyde i processen ved at følge processen, mens de passivt forsøger at udlede forretningshemmeligheder. For at forhindre sådan en opførsel definerer vi en possibilistisk ide om hemmelighed af handlinger i procesmodeller med *run*-baseret semantik, således at en samarbejdspartner kan tage en specifik handling i en distribueret proces, sikker i sin viden om at specifikke samarbejdspartnere ikke ved, at de tager den. Vi viser derefter, at denne definition af hemmelighed er beregningsmæssigt svær at bestemme for nogle forretningsprocesmodeller, specifikt Dynamic Condition Response grafer, så vi præsenterer som alternativ en betingelse, der er tilstrækkelig til at bestemme hemmelighed.

# Acknowledgements

Thanks to Søren Debois for supervising me in my bachelor's thesis, master's thesis and now PhD thesis. Throughout this time Søren has gone above and beyond his duties as a supervisor. He has believed in me, even when I did not believe in myself, and he knows how to bring out the best researcher in me. He has been very gracious in giving me input when I needed it, but likewise, he has known when to tell me to move on and get working I have been able to share my struggles with him throughout this project, and I have enjoyed his good counsel on both academic and personal matters.

Thanks to Holger Stadel Borum for your friendship and collaboration during our bachelor's, master's and PhD studies. I appreciate your advice and support and am very privileged to have it still.

Thanks to Jonas Kastberg Hinrichsen for taking me under his wing when I first started as a PhD student. He welcomed me with open arms and eased my transition from a student to a researcher as much as anyone possibly could.

Thanks to Simone Rasmussen for being steady as a rock when I find myself on shaky ground. I could not have finished this project without her love and support.

# Contents

**Contents** **iv**

**1 Introduction** **1**
1.1 Necessary formalisms and terminology . . . . . . . . . . . . 2
1.2 Related work . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.3 Introducing the Transforming paper . . . . . . . . . . . . . 10
1.4 Introducing the Non-equivocation paper . . . . . . . . . . . 18
1.5 Introducing the Impalpable Differences paper . . . . . . . . 22
1.6 Discussion: Lying in a distributed business process execution 23
1.7 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . 29
1.8 Introductory conclusion . . . . . . . . . . . . . . . . . . . 33

**2 Transforming Byzantine Faults Using a Trusted Execution Environment** **35**

**3 On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems** **44**

**4 Impalpable Differences: Secret Actions in Processes and Concurrent Workflows** **57**

**Bibliography** **78**

# Chapter 1

# Introduction

Collaboration is a fundamental necessity for companies to survive; no company can secure all their necessary competencies and resources without also engaging in collaborations with their customers. And as the world has become more internationally interconnected, and the number of collaborations companies engage in has increased, the collaborations have grown more complex. One way of clarifying collaborations in a systematic and rigorous manner is by using business process models to model collaborative processes.

The rich research area of Business Process Management (BPM) includes ways of executing instances of such processes, making much of the process automatic. Such execution engines have the potential to allow for an increase in collaboration volume; with a formal process and automated communication and recording of the steps taken, companies can handle a larger volume of collaborations. But with such an expansion in collaboration also comes an increased surface of attack: the more collaborative efforts a party engages in, the greater the risks of some collaborator violating the agreed-upon process, possibly with malicious intent. In this thesis I take steps towards *preventing* foul play by such malicious collaborators, thereby eliminating some of the risks that companies must take when engaging in collaboration.

This introductory chapter continues as follows: first, we introduce the formalisms necessary to properly discuss the subject. This includes a basic formal model for a distributed business process, a system model and a formal problem statement. Then we present work related to the research area of secure business processes. This is followed by three sections, each of which describes in greater detail the main results of this thesis, their relevance to the problem statement and their uses. The last part of this chapter is a section on the future work of this topic.

Chapters 2, 3, and 4 are the papers associated with my PhD project and the source of the main results described in this chapter.

## 1.1 Necessary formalisms and terminology

A *business process model* is a process model of one or more enterprises or businesses. A process model consists of a collection of smaller *activities*, *tasks* or *actions* that are related and serve to fulfil the common goal of the process. Each entity able to take an action in a process is referred to as an *agent* or an *actor*. The process model may have different implementations with different semantics. Examples of these include Business Process Model Notation (BPMN) [1], Dynamic Condition Response (DCR) graphs [2], Petri-nets [3], Even-driven Process Chains (EPC) [4], Business Process Execution Language (BPEL) [5], UML activity diagrams [6], and many others. My main focus during my research has been DCR graphs, but we will use a very basic formal process model during this chapter, that is nonetheless representative of the semantics of most process models.

The *execution* or the *run* of a process model is the initialisation of a process model and the consequent execution of the actions of the model by the actors. We may refer to the instance of a process model as simply a process. Each run of a process produces a *trace*: a description of the actions taken in the process and the (partial) order in which they were taken.

A *distributed* business process model is a model where different parts of the model are time- or space-decoupled, i.e. different actors are not temporally or spatially co-located and they must therefore coordinate their actions with each other for the correct execution of the process. This coordination and communication between the actors in a collaborative business process is known as *process choreography*. A system allowing for distributed process executions is called a distributed process execution *engine*. We may refer to them as simply *execution engines* throughout this chapter.

One aspect of creating an execution engine where actions can be taken in a distributed manner is to ensure *consistency*. In classical concurrency theory, consistency refers to the property that concurrent operations preserve the integrity of the system i.e. does not cause the system to enter a state that it could not have entered under sequential operation (see e.g. [7, Ch. 16]). Similarly in execution engines: when actors can take actions in a time- and space-decoupled manner, consistency refers to the

property that the same runs are possible in a sequential implementation. More formally: given a partitioned process model, the concurrent executions of the partitions must behave as the sequential execution of the process. We use the terms *global* process to refer to the entire process, and *local* process to refer to a single partition of the process. Hildebrandt et al. refer to this problem as the *consistency problem* [8], a name that we adopt here. Note that the problem is also described elsewhere, e.g. by van der Aalst and Weske [9], although they focus on extending partitions of the process, and the closely related to the notion of *composition*(see Section 1.2.) Hildebrandt et al. [8] show how to solve the consistency problem for DCR graphs, notably in a synchronous environment with no faults. In this thesis, we take steps towards solving the consistency problem, but in a *malicious* setting: actors may behave in a malicious manner. They may lie, cheat and steal (secrets), and we still require consistent behaviour in the global process. The steps presented in this thesis aim to *prevent* the actors from lying, cheating and stealing, by using trusted subsystems and information flow theory.

Seeing as a large part of this thesis deals with both distributed computing and business process modelling, the term *process* is ambiguous, meaning both a collection of actions as described above and a computing entity in a distributed algorithm. To avoid any ambiguity, we will refer to a process in distributed computing (i.e. a computing entity in a distributed algorithm) as a *processor* (see e.g. [10]), while retaining the meaning of the word process from the world of business process modelling.

### Problem statement

Given a consistency problem in a synchronous system with reliable channels and malicious actors, how can we:

- Prevent the malicious actors from sabotaging consistency by lying about what actions they take?
- Prevent the malicious actors from sabotaging consistency by cheating without lying?
- Prevent the malicious actors from stealing secrets?

Achieving solutions to the above is central to achieving successful *adversarial collaboration.*

Note that we do not attempt to solve the consistency problem per se, but rather prevent malicious actors from acting maliciously in the setting of the problem. If the above is a comprehensive list of all malicious actions in adversarial collaboration, then applying the preventative measures to existing solutions will allow these solutions to become resistant to adversarial behaviour.

My research into the topic began before my PhD, with the paper *Collaboration among Adversaries: Distributed Workflow Execution on a Blockchain* [11]. In that paper, my co-authors and I describe a way to implement an execution engine on the Ethereum blockchain, thereby preventing malicious actors from lying and cheating, since all actions taken are public information. While successful, that solution has several shortcomings. Firstly, even back in 2018, executing even small processes on the Ethereum blockchain was fairly expensive. For a small process with 5 different possible actions, a single regular execution would cost between 6.6 and 16.2 USD, depending on the chosen implementation, at the time of publication. As of March 1st 2023, this has drastically increased to cost between 36.28 and 87.64 USD [12, 13] for the same workflow. Note that the ratio between the implementations has changed due to a change in the ratio of cost between a computation (gas) and the cost of Ethereum currency (ETH). Secondly, the throughput of the number of actions taken per minute is bounded by the time it takes for a block to be published and verified in the blockchain, which results in the number of actions per minute being 1–2. Furthermore, the solution limits local autonomy: an actor cannot simply take an action that they are allowed to take per the semantics of the process, rather they have to attempt to take the action and wait for that fact to either be accepted or rejected on the blockchain. Ultimately, the approach using blockchains served as a proof-of-concept, and this thesis can be viewed as the natural next step in this research area.

### Process model

Throughout this chapter, we will use an informal notion of a process model where a process consists of (1) a set of actors, (2) a set of actions per actor, that the actor may take at some point in the process (3) a rule restricting when actions are *enabled*, i.e. when that can be taken. A trace of a process run is a sequence of actions taken. Such a process describes a set of possible runs, and so a set of possible traces. We call this set of traces the *language* described by the model. During a run, each actor has access only to their set of actions, the sequence of actions they have

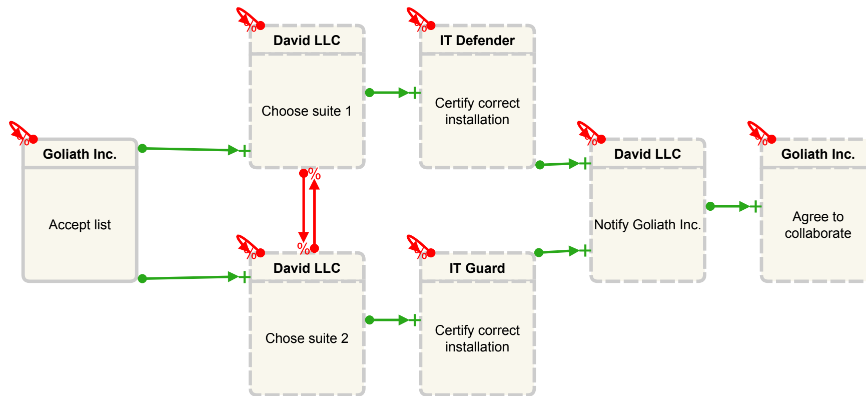



Figure 1.1: DCR graph of the recurring example process.

taken, and which of their actions are currently enabled. This model is very simple but it is useful for discussion about the different main results in the context of adversarial collaboration and the consistency problem.

### Recurring example process

Throughout this chapter, we will use the following example of a process recurringly. Consider two companies, *Goliath Inc.* and *David LLC* ready to enter into some continuous collaboration. The legal and practical details for the collaboration are all settled when *Goliath Inc.*'s security department notices that *David LLC* is using an old and insecure internal security suite. This is not a problem at present, since the security suite is only in place on *David LLC*'s intranet, which only collaborators with *David LLC* have access to. However, since the collaboration will entail that *David LLC* must store several critically confidential documents digitally, *Goliath Inc.* postpones the signing of the collaboration agreement until *David LLC* upgrades their internal security suite, to prevent other collaborators from gaining access to these documents. *David LLC* agrees to this stipulation, but with the addendum that *David LLC* will use one of several named security suites, but keep exactly which one they use confidential. That way, for a malicious collaborator to exploit a zero-day vulnerability, the malicious collaborator would have to keep track of all vulnerabilities of the named security suites. *Goliath Inc.* agrees to this addendum, and they agree on the following process for choosing and implementing a new security suite on

*David LLC*'s intranet:

When *David LLC* has presented an adequate list of security suites to *Goliath Inc.*, *Goliath Inc.* starts the process by accepting the list. Then *David LLC* chooses one of the suites from the list – in this example we consider the list to be of size 2, for the sake of simplicity. When the suite has been installed, the appropriate security company certifies the correct installation: mutually exclusive events which can be taken by either *IT Defender* or *IT Guard*. *David LLC* then notifies *Goliath Inc.* of successful implementation, who then finally signs the collaboration agreement.

It is important to distinguish between the specific notions of *collaboration* and *security* in the example, and the general notions of such that we apply elsewhere in the thesis. We will make explicit when we are referring to these terms in the context of the example.

The process is available as a visualised DCR graph in Figure 1.1. In DCR notation the dashed lines indicate that the action is *excluded*, i.e. cannot be taken (yet). The green arrows indicate that, after the action at the origin has been taken, the action at the destination will no longer be excluded. Likewise, the red arrows indicate that, after the action at the origin is taken, the action at the destination will be excluded. For a more comprehensive explanation of DCR notation, see in Chapter 4, Section 4.1. By the semantics of the DCR notation, then the graph in Figure 1.1 describes a process where *Goliath Inc.* must accept a list before *David LLC* can choose suite 1 or 2, which in turn is followed by first a certification of the chosen suite, then a notification by *David LLC* to *Goliath Inc.*, who can then finally agree to collaborate. In short, it models the recurring example described above.

### TEEs, non-equivocation and BPMs

The first two papers in this thesis are not obviously in the field of BPM. Rather, they contain more general results in the fields of trusted computing and distributed computing that also applies to the field of distributed business process models. They generally study how to limit adversarial behaviour in distributed systems. Specifically, Chapter 2 studies *Trusted Execution Environments* (TEEs), and Chapter 3 studies the trusted computing property *non-equivocation*. Their results apply to adversarial collaboration by the distributed nature of such collaboration, and so we can apply the results from those papers to the research area of securing adversarial collaborations. Their exact application will

be expounded upon in their respective introductions below (Sections 1.3 and 1.4).

They also share an application to *agreement problems* in distributed computing, where a set of processors has to reach agreement on the input values of the processors. This is relevant to the problem of adversarial collaboration since we can achieve a solution to the consistency problem by using an agreement algorithm. Such a solution is achieved by replicating the state of the global business process on each processor, and then using an agreement algorithm to agree on each step of the process. That way we can achieve consistency in the global process, simply by the fact that each processor is aware of the global process state. This is known as the *state machine approach* or *state machine replication* (SMR), and it is well-known that agreement is central to this problem [14]. By using an agreement algorithm that is tolerant to Byzantine faults, we can eliminate malicious behaviour by simply filtering out requests that are not consistent with the shared global state, see e.g. [15]. Note that in this solution to the consistency problem, each actor must necessarily know the full current state of the global process which has consequences for the possibility of secrecy. See Section 1.5 & Section 1.7 for discussions on how consistency and secrecy conflict. Furthermore, by using an SMR solution, the local autonomy of each actor is lessened, since they have to make a request before taking a step in the process (i.e. executing an action), a request that may fail.

## 1.2 Related work

We give in this section a high-level overview of relevant work in the BPM space. For a recent (2021) state-of-the-art literature review on security in business processes, we suggest the excellent work of Abdmeziem et al. [16].

Abdmeziem et al. [16] use a 3x3 matrix system for classifying research in the security of business processes The classification system is built on the observation that there are three different *dimensions* in business process research, and three different *goals* in security research: the informational-, logical- and organisational dimensions and the integrity-, confidentiality- and availability goals, respectively. In the nomenclature of that work, this thesis is situated firmly in the logical dimension of business process research, which considers the control flow logic of the process, rather than the data and artefacts (the informational dimension) or the role delegation (the organisational dimension). The first

two papers of this thesis (Ch. 2 & 3) implicitly study the security goal of integrity of the logical dimension. They contain results that give guarantees about correct control flow logic, i.e. consistency of the global process, in a Byzantine setting. The third paper (Ch. 4) studies the security goal of confidentiality of the logical dimension, i.e. when actions are secret.

We now move away from the classification system of [16].

*Composition & partitioning of business processes* A central part of collaborative business process executions is to ensure correct global behaviour of partitioned processes. It is not necessarily obvious that this is a security concern, but when we consider the classification matrix above, it is easy to see that composition can be classified as an integrity property of the logical business process dimension.

Considering the business process model notation *workflow nets*(see [3]) van der Aalst and Weske [9] show that a public inter-organisational (i.e. collaborative) process can be partitioned into smaller (private) processes, which can then be extended with more actions while still preserving the behaviour of the public process. This is called the *Public-to-Private* (P2P) approach. The only requirement is that the extended private process must be a subclass of the unextended process partition, which intuitively translates into not extending the behaviour of the partition, only limiting it. This approach can allow for an extended application to the notion of secrecy in Section 1.5, since extensions of the public workflow are necessarily unknown to other collaborators.

Hildebrandt et al. [8] present the consistency problem as described previously. The solution presented uses a notion of dependency of actions defined in a way such that preventing the concurrent execution of such dependent actions, together with the appropriate updates to enabledness of action, directly implies global consistency.

Goettelmann et al. [17] uses partitioning similar to the P2P approach to utilise cloud technology for executing business processes while still keeping fragments of the process secret from the cloud provider. This approach, put in simplistic terms, partitions the processes in ways where a confidential fragment of the process is distributed across multiple cloud providers, whereby they cannot deduce the logic of that fragment. Nacer et al. [18] extends this approach by including fake fragments as well, to account for collaborating providers.

*BPM on the blockchain* The research into applications of blockchains in business processes has been extensive. Mendling et al.[19] present challenges and opportunities to this application, with the main take-

away regarding executions being that blockchains can function both as a medium of coordination through message parsing and as a medium ensuring consistency. Henry [20] includes an excellent overview of recent work, classifying current research into 3 areas: (1) Business process execution engines, (2) flexibility (i.e. dynamic changes) and (3) privacy (i.e. confidentiality).

Business process execution engines using blockchains include the work of López-Pintado et al. [21] (BPMN) and Madsen et al. [11] (DCR graphs), while Tran et al.[22] presents a modelling tool for designing BPMN processes and exporting them as separate blockchain smart contracts, which differs from the other approaches by the fact that no single contract will manage all relevant processes. This has a slightly positive impact on the privacy of the collaborators in comparison with the other two, but at the cost of publishing a smart contract.

An interesting other application is presented by Müller et al. [23] where the trust layer in BPMN presented in [24] is used as an analysis tool to analyse trust relationships before mitigating any found issues with blockchain technology.

*BPM and cryptography* Cryptography is a common solution to security issues of confidentiality and integrity, and so it is naturally widely used when solving security issues in business processes as well. These uses include that of Backes et al. [25], which uses cryptographic primitives to ensure confidentiality and integrity for security requirements based on trust relationships. The authors present a method for identifying such relationships when modelling business processes, and then how to protect data with cryptography when trust is not present. As such, they do not consider the confidentiality of activities.

Rohm et al. [26] present the specification language ALMO$T, which can specify secure transactions in business processes. After such specification, the secure transactions can be executed by a framework, guaranteeing confidentiality and integrity of the transaction by way of cryptography. The ALMO$T specification language is highly specific and is not applicable to general business process models.

Carminati et al. [27] suggest a method for encrypting a BPEL [5] process which preserves the execution logic of the process. This allows them to publish the process on a blockchain while preserving the confidentiality of the process.

*Process mining* Process mining – extracting process models from execution logs – is a well-studied area in business processes (see e.g. [28] for a review). Process mining with security considerations includes the

work by Müller et al. [29], which uses a Trusted Executions Environment (TEE) to mine collaborative processes with private sub-process (using the P2P approach described in [9]). The reasoning is that collaborators might be reluctant to share logs of their private sub-processes. To enable process mining under such conditions, a TEE is utilised to keep the logs and private sub-processes confidential to everyone but the appropriate actors.

*BPM and compliance* The European Union's General Data Protection Regulation (GDPR) has been the catalyst for research into how to show that a process complies with such regulation. This implies confidentiality requirements, but from the point of view of the *users* rather than the organisation.

An approach for determining the purpose with a process is presented by Basin et al. [30], which leads to a methodology for auditing compliance with the GDPR in BPMN. With the addition of a dataflow model, such compliance can be shown algorithmically.

Taking a modelling approach, Agostinelli et al. [31] present a set of design patterns based on GDPR, which allows for compliance to be modelled directly into the business process.

*BPM and game theory* Applying game theory to business processes allows for interesting analyses of adversarial collaboration in business processes. It supports analysis of adversarial behaviour where the adversarial collaborator attempts to derail the process or steer it into a state beneficial to them. This game-theoretic approach is introduced by Heindel and Weber [32] and expanded to a solution using secure multiparty computation by Haagensen and Debois [33].

## 1.3 Introducing the Transforming paper

The purpose of this section is to introduce the paper *Transforming byzantine faults using a trusted execution environment* [34], available in Chapter 2.

The main contribution of the paper is a transformation of distributed algorithms. The transformation includes a translation of Byzantine faults to omission faults. That is, after the transformation, a correct processor running the algorithm will act as if it received no message – the message was omitted – when it receives a message from a (Byzantine) faulty processor. The transformation relies on a Trusted Execution Environment (TEE) to act as a Byzantine fault detector. In fact, the paper is an attempt to formalize that this is exactly what a TEE is. The

paper is based on a chapter in a master thesis [35], in which I and my co-authors attempt to show that a TEE is essentially a tool for preventing integrity faults of running code and associated data, which can be considered equivalent to Byzantine faults in a distributed algorithm. The paper expands upon this idea and formalises exactly how a TEE might catch Byzantine faults, and discard messages from faulty processors.

The argument goes as follows: a TEE is a subsystem able to run code with the properties of *integrity*, *confidentiality*, *authenticity* and *remote attestation*. Using the property of remote attestation, which allows an application running inside a TEE to identify and authorise itself to remote TEEs, we can provision TEEs with cryptographic secrets. Such secrets can then be used to cryptographically authenticate messages, thus proving to receiving TEEs that the message originates from an authenticated application in a TEE, and has not been altered since leaving the TEE.

The confidentiality property guarantees that no other code, than the authorised code in the TEE can access the associated data. The integrity property guarantees that if data is changed from somewhere else than the authorised code in the TEE, then the data stops being available. This way, the cryptographic secret provisioned by the remote attestation algorithm becomes unavailable on integrity violations that might lead to a Byzantine fault. Lastly, the authenticity property guarantees that the code running in the TEE has not been altered since compilation.

Putting these concepts together, we can eliminate integrity faults of code, data and messages, simply by moving the data and code running the distributed algorithm on each processor into respective TEEs. Thereby the code cannot change due to the authenticity and integrity guarantees, data cannot change due to the integrity guarantee, messages cannot change (without being noticed) due to the confidentiality of the cryptographic secret, and no maliciously (Byzantine-) faulty processor can imitate a non-faulty processor due to the confidentiality of the cryptographic secret. If a processor receives a message from a processor that has experienced an integrity fault, or if the message itself has experienced one such fault, then the cryptographic authentication will ensure that the receiving processor drops the message, thereby transforming the fault into a message omission.

Note that the transformation is based on several uses of cryptographic authentication, both in the implementation of integrity and confidentiality of the TEE, remote attestation and message authentication. As such, the guarantees are probabilistically rather than information-theoretically

secure, meaning that it relies on standard cryptographic assumptions and that the guarantees can be broken with negligible – i.e. incredibly small – probability.

### Relation to adversarial collaboration

The ability to translate Byzantine faults into omission faults is quite powerful for the use of adversarial collaboration. Consider an adversarial collaboration that utilises a distributed process execution engine. If this execution engine has been transformed using a TEE to translate Byzantine fault, then every time an actor receives a message of an action taken by another actor, then they are guaranteed that the sending actor has actually recorded the action in their own execution engine, and so cannot repudiate their actions. In effect, it prevents the malicious actor from *lying*, by guaranteeing the consistency of all local processes.

Consider the example in Section 1.1. The major threats to *Goliath Inc.* and *David LLC* by Byzantine faults in the execution engine are the following:

1. If *David LLC* believes that *Goliath Inc.* has accepted the list of security suites, but *Goliath Inc.* has not, then *David LLC* might spend considerable resources on implementing a suite that is unacceptable.
2. If *David LLC* believes that one of the security companies has installed their suite, but they have not, then *David LLC* might accidentally leak *Goliath Inc.*'s confidential information.
3. If *Goliath Inc.* believes that *David LLC* has been certified due to notification of such, but *David LLC* actually did not receive such certification, then *Goliath Inc.* might similarly risk sending confidential information to *David LLC* and so risk leakage of this information.

By eliminating Byzantine faults, the actors can trust when the execution engine informs them that an action has been taken by some other actor. This actively eliminates all of the above threats. It even allows the actors to transitively apply this guarantee: when *Goliath Inc.* sees that *David LLC* has taken the action of notifying *Goliath Inc.* of certification, *Goliath Inc.* is guaranteed that not only did *David LLC* take this action, but the prerequisites for this action have also been taken! So *Goliath Inc.* is guaranteed that either *IT Guard* or *IT Defender* did, in fact, certify a security suite for *David LLC*'s intranet.

Aside from the use of the transformation directly on the distributed process execution engine layer, the transformation also has uses in the implementation details of the execution engine. As described in Section 1.1, another way of achieving consistency is by the use of agreement algorithms. However, such agreement algorithms can be computationally expensive: they require a lot of communication and a lot of replicated processors to ensure fault tolerance, i.e. that no malicious actor(s) can break non-repudiation in this case. However, by eliminating Byzantine faults, such a solution can be achieved more cheaply.

Where the transformation does not help, is if a malicious actor figures out a way to act correctly according to the prescribed process, but still cheat in some way. For instance by using omissions to break consistency, since the transformation does not handle omission faults. An example of this can be found in Section 1.4.

#### Vulnerability: replay attacks

The transformation does not always protect against replay attacks. This vulnerability is not mentioned in the paper. After the remote attestation step, message authentication in the paper is achieved by a generic non-interactive message authentication code (MAC) algorithm. The importance of non-interactiveness comes from the fact that the transformation aims to add no additional communication steps after remote attestation, and so the MAC must be computed without communicating with the receiving processor. However, the paper does not examine how this leaves the transformation vulnerable to replay attacks for distributed algorithms where identical messages are sent from different processors or at different times.

Consider a distributed algorithm in a synchronous system, in which all processors broadcast some value in the first synchronous round. Then, in round 2, the processors forward the lowest value it received in round 1. Since the processors simply forward values in round 2, repetition of identical messages does occur. If a Byzantine faulty processor in a transformed version of the algorithm then fails after the send step in the first round, but before receiving any messages, the processor can simply forward an arbitrary or maliciously chosen value in round 2. And since the messages are correctly MAC'ed, they are indistinguishable from messages sent by a non-faulty processor, which circumvents the translation of faults.

As is apparent from the above example, a Byzantine faulty processor can break the guarantees of a transformed general omission fault-tolerant

algorithm by replaying messages, if that algorithm uses identical messages at different times or from different processors. We suggest three different ways to solve this vulnerability: (1) the underlying algorithm could be changed to make messages identifiably unique. In the above example, this could be achieved simply by appending the processor identifier to each message. That way, the faulty processor could not replay messages to appear as correct messages from the faulty processor without corrupting the authentication code. This does not fix the vulnerability for all systems, however, e.g. systems where malicious processors have control over the channels.

(2) In that case, one could solve the issue by changing the use of remote attestation. Rather than provisioning the processors with a common cryptographic secret, the remote attestation step could be used to provision the processors with a public key infrastructure (PKI)i and sign the messages with a private key, rather than using a MAC. By doing so, a replayed message would be easily identifiable as originating from another processor than the processor replaying the message. (2.1) If the TEE implementation in question is *Intel® software guard extensions* (SGX), then the remote attestation based on Intel® EPID enables processors in establishing secure channels protected by the same guarantees as the TEE itself [36]. However, a PKI is a lot more complex and requires several communication steps to provision, so it may not useful in all cases.

(3) A third possibility is to alter the transformation to allow for each message in the original algorithm to be extended with a 2-step challenge-response protocol: first, the sender requests a nonce – a random number – from the receiver, who supplies such a nonce. Then the nonce is appended to the original message before the authentication code is generated and appended and the entire sequence is sent. This is a standard way of preventing replay attacks, and more information can be found e.g. [37, Ch. 3]. Of course, (3) changes the translation of fault from being a 1-round, $n = f + 1$ translation to being a 3-round, $n = f + 1$ translation, since the non-interactiveness of the transformation is sacrificed.

#### Vulnerability: TEE

Since the time of publication of the paper, many new vulnerabilities to most TEE implementations have been discovered, including the three major hardware technologies that support TEE implementations, namely AMD's Platform Security Processor (SP), ARM's TrustZone and In-

tel's Software Guard Extension (SGX). Some good overviews of vulnerabilities in different TEE implementations include Van Bulck et al. [38] who present 35 vulnerabilities across Intel, RISC-V [39] and Sancus [40] TEEs, Khalid and Masood [41] who present 5 Intel TEE vulnerabilities, 5 ARM TEE vulnerabilities, and 3 AMD TEE vulnerabilities, and Muñoz et al. [42] who present a variety of attacks on 48 different TEE implementations, classified under software-attacks, side-channel attacks and architectural attacks.

Considering that the transformation requires all of the security guarantees a TEE provides, i.e. integrity, confidentiality, authenticity and remote attestation, any attack on a TEE will break the guarantees of the transformation. In the following two major vulnerabilities on Intel SGX are outlined, one for integrity and one for confidentiality.

*Integrity* Plundervolt [43, 44] is an exploit on the integrity of Intel SGX. It utilises an undocumented API for controlling voltage to the Intel CPU, to induce relatively predictable errors in integrity-protected code. Errors that crucially are not detected by the SGX framework. The attack works by requesting a lower-than-expected voltage to the CPU (i.e. by *undervolting*) just before calling integrity-protected code. By doing so an attacker can cause predictable errors in some CPU instructions while maintaining correct behaviour for the rest. Notably, an attacker can with relative predictability cause a bit-flip in the third byte of a multiplication instruction.

If this bit represents whether an action is enabled in an execution engine, an attacker could be allowed to execute an action despite it being disabled in the global state. Consider the example in 1.1. By using Plundervolt, an adversarial *David LLC* could notify *Goliath Inc*. that they had successfully had a new security suite implemented, without having done so, and no security company has provided a certificate, since Plundervolt could allow execution of disabled actions. *Goliath Inc*., believing that such a lie is impossible due to the protection of the underlying TEE, would then accept the word of *David LLC* and terminate the process.

*Confidentiality* SGXpectre [45] is a confidentially exploit on Intel SGX, based on the Spectre [46] and Meltdown [47] exploits on Intel CPUs. SGXpectre abuses the speculative execution of Intel CPUs to access confidential data and leak it via side-channel attacks. A common example of one such side-channel attack is a timing attack on cache-state changes.

By forcing a speculative execution of a specific function, an attacker can force SGX to load a cryptographic key. Then, having carefully con-

trolled the contents of a specific cache, they can note the loading time, thus deducing how much of the cache changed. By repeating this exploit with different contents in the cache, an attacker can eventually deduce the loaded secret.

Using SGXpectre, an attacker could extract the key for signing the messages in the transformation. By doing so, they can circumvent the TEE entirely, and sign any message as being free from Byzantine faults. Any adversarial collaborator could use this to circumvent the integrity guarantee of the transformed execution engine and execute any action despite the enabledness of that action.

*Vulnerability effect* Both of the described exploits require, at least partial, physical access to the device under attack. Unfortunately, this is exactly the adversarial model for the transformation: the adversaries are running their own code on their own machines. Together with the sheer amount of TEE vulnerabilities found in recent years, this indicates that one cannot implicitly trust the guarantees of a TEE. It should be noted that the exploits described above have been patched, and are no longer viable, and so has most other serious TEE vulnerabilities (see e.g. [42]). Regardless, it seems that TEE-technology is not yet mature enough to secure critical systems, so one should use the transformation of the paper with care.

### Discussion: Integrity- vs. Byzantine faults

The paper presents an argument of how a TEE can transform integrity faults of code, data and messages into message omissions. However, this is not necessarily the same as transforming all *Byzantine* faults, as suggested by the title. In a system model with a Byzantine fault model, we generally consider *arbitrary* faults possible [7, Ch. 2], so that the Byzantine fault model encompasses both the general omission-[1] and crash fault models – see Figure 1.2 for an Euler diagram of the fault models.

Usually, we refer to a Byzantine fault as one that is not encompassed by the general omission fault model, so when we say that we translate a Byzantine fault to an omission fault, it means that we translate an arbitrary fault outside the omission fault model to one inside it. The only assumption we make is that Byzantine faults cannot simulate the guessing of cryptographic secrets. Notable exceptions to this definition of Byzantine faults include [48] and [49], both of which define a Byzantine fault as

[1] The general omission fault model allows faulty processors to omit messages they are supposed to send, and that they have received.

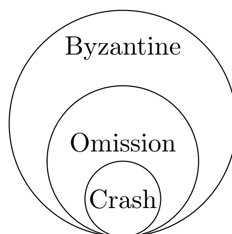



Figure 1.2: Euler diagram of the traditional fault models.

"a fault presenting different symptoms to different observers" – almost exactly the definition of equivocation, see Section 1.4 and Chapter 3. Since this definition does not include faults outside the general omission fault model which presents consistent symptoms to all observers, we will use the traditional definition. Regardless, it is not immediately clear that *all* Byzantine faults are integrity faults, and that our claim to translate Byzantine faults is, in fact, achieved by using a TEE.

An advantage of using the traditional definition of Byzantine faults is in this definition a Byzantine fault is equivalent to one or more faulty messages[2], at least in traditional distributed system models where communication is only available via message-parsing. In [50], this is described as "[a faulty processor], beside failing to send the required messages, may also send false and contradictory messages, even according to some malevolent plan." This equivalence is because the behaviour of a processor is exactly defined by the messages it sends since this is the only way for the processor to communicate; to deviate from the expected behaviour is exactly to deviate from sending the expected messages since this *is* the expected behaviour[3].

Since the fault of omitting a message is included in the general omission fault model, and so excluded from being a Byzantine fault, the only unexpected behaviour left is the sending of messages, i.e. messages inconsistent with the behaviour specified in the algorithm. And this is exactly what the transformation prevents: any unexpected and unauthorised changes to integrity-protected data and code. The cryptographic authentication even allows processors to see changes made to the message during transmission. The only thing that the transformation cannot

[2]To be precise: a faulty message is a message that is not specified as correct behaviour of a non-faulty processor.

[3]We will leave aside discussion of whether a processor can be considered faulty while it behaves in a non-faulty manner. In this thesis, a faulty processor is a processor that behaves faulty.

protect against is faults in the implementation or the hardware of the TEE, and faults in the implementation of the algorithm, all of which we assume to be without faults. These are reasonable assumptions: the former because a TEE is a trusted computing device – if one cannot trust a trusted subsystem in trusted computing to behave as expected, then both the practise and the subsystem becomes useless – and the latter because if one cannot trust the implementation of an algorithm to be correct of, then surely the properties of the algorithm cannot be expected to still hold; imagine if the merge sort algorithm guaranteed performance of $O(n \cdot log\ n)$, even if the implementation did not correctly behave as the specification.

There are, however, real-world use cases where the above assumptions are not as reasonable. For instance, [15] suggests that a use-case for agreement algorithms is in critical systems, where each processor could run different implementations of the critical system, and for each output, the processors could run an agreement algorithm. That way, one could eliminate faults due to *faulty implementation*. As such, it is important to note that this is outside the scope of use of the transformation, due to our basic assumption of correct implementation. Similarly, we showed various vulnerabilities of TEEs, which seems to indicate that the trusted computing device in question is, in fact, not trustworthy.

## 1.4 Introducing the Non-equivocation paper

Here we introduce the paper *On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems* [51], available in Chapter 3. The paper revolves around the notion of *equivocation*: when a faulty processor acts differently towards two or more non-faulty processors. The idea of equivocation can be found back in 1980 in the seminal paper *Reaching Agreement in the Presence of Faults* by Pease et al. [10], which also introduces the problem of *interactive consistency*, showed how cryptographic signatures eliminate the need for duplication in synchronous agreement systems, and is the paper which inspired the famous *The Byzantine Generals Problem* [52]. In the first paragraph of Reaching Agreement, we find the following quote "a bad processor might report one value to a given processor and another value to some other processors", explicitly defining equivocation as the fault to be aware of in Byzantine systems, but without naming the fault *equivocation*.

The term *equivocation* was, to the best of my knowledge, not given to this kind of faulty behaviour before 2007, when it was introduced in the

paper *Attested Append-Only Memory: Making Adversaries Stick to Their Word* [53]. This paper introduces a small trusted cryptographic component which eliminates the possibility for a faulty processor to equivocate, called *Attested Append-Only Memory* (A2M). This component, and those similar to it, essentially allows a processor to digitally associate an index in a monotonic increasing counter to one, and only one, message. This ensures that a processor that is supposed to send a message as a part of distributed algorithm, can choose only one message to send since any other would be associated with a different index in the monotonic counter. This protects against equivocation, as a faulty processor can no longer "report one value to a given processor and another value to some other processors".

Another way of preventing equivocation is by the use of different kinds of broadcast channels, see e.g. [54], such that when the processor sends a message on a channel, that message is delivered to several processors. This eliminates the possibility for a faulty processor to send different messages to different processors, simply because the processor only has access to the one channel. Here the non-equivocation property – the property that prevents equivocation – is located, not with the *processors*, but with the *channels*. And that has a quite profound effect: when the non-equivocation property is located with the processor a faulty processor can still choose whom to send the message to, and, more importantly, whom not to send it to. When the non-equivocation property is located on the channels, however, a faulty processor has only the choice between sending a message on that channel, and not sending it. It can no longer try to violate system properties by tactically choosing specific processors not to send values to. It is this difference that the paper in Chapter 3 examines.

In the paper my co-author and I identify the properties *strong* and *weak* non-equivocation: in strong non-equivocation, a faulty processor can only choose to send a message to all recipients or send no message at all, while in weak non-equivocation a faulty processor is allowed to choose the recipients of each message. For ease of reasoning, we define the dual of these properties: *strong equivocation* is when a processor sends at least two distinct messages, and *weak equivocation* is when a processor sends only one distinct message but omits to send that message to at least one processor. In this terminology, weak non-equivocation prevents strong equivocation, and strong non-equivocation prevents both weak and strong equivocation. The differences between the non-equivocation properties are most distinct in synchronous system models with reliable

channels since the difference is the omission of certain messages – something which only happens to *faulty* processors in this system model. Note that this is also the system model of the global consistency problem in Section 1.1. We show in the paper how the non-equivocation properties allow for different fault tolerances and amounts of communication for different agreement problems; strong non-equivocation allows for the lowest of both.

Considering the definition of weak equivocation, it seems reasonable to think that weak equivocation is a kind of message omission. It is important to understand the distinction. When a processor commits a message omission, the message it is (not) sending is the message that is appropriate according to the algorithm. When a processor commits weak equivocation, there are no restrictions on what message the processor is (not) sending. In other words, non-equivocation does not prevent a processor from lying; a faulty processor can freely choose the message that they want to send.

Agreement and equivocation are closely linked. The act of reporting two different messages to two different processors, when you are supposed to report the same message, is an attack on agreement. The two processors were supposed to agree on the message that they received, which they no longer do. It is, therefore, no surprise that we show that subsystems providing the properties also provide solutions to well-known agreement problems: strong non-equivocation solves Byzantine broadcast[4] [52], and weak non-equivocation solves crusader agreement [55]. This is not to say that (non-)equivocation has no interest outside research into agreement protocols, rather, this shows that non-equivocation can be used in systems where agreement can be used to achieve some other goal, for example in distributed process execution.

### Relation to adversarial collaboration

Intuitively, there seem to be two uses of non-equivocation in adversarial collaboration. The first is as a tool to eliminate equivocation directly in an existing solution to the consistency problem with a weaker adversary model, such as the one in [8]. The other is to implement an agreement algorithm to handle communication in the system, as described in Section 1.1, and then use non-equivocation tools to increase the fault tolerance of those systems, thereby making the solution robust against more malicious actors. Unfortunately, what follows now is an argument

[4]This should certainly be no surprise, since the inspiration to the property came from broadcast channels

that to guarantee fairness and consistency with non-equivocation, one must also solve an agreement problem, even with the use of a TEE as described in Section 1.3. The basic intuition is that, due to well-chosen omissions, an adversary can break consistency of a global process by not letting other actors know that their actions are no longer enabled.

At first glance, strong non-equivocation seems to handle malicious omitting of messages – the malicious actor can only choose to inform the correct set of actors when they take an action, or no one at all, which looks equivalent to not taking the action at all. So using a subsystem providing strong non-equivocation together with a TEE to prohibit the possibility of lying, as described in Section 1.3 and Chapter 2, seems to prevent a malicious actor from acting maliciously *at all*. This is, unfortunately, not the case.

Recall the process model from Section 1.1, and consider a process with the actors $A$ and $B$, such that $A$'s actions are $\{a, c, d\}$ and $B$'s actions are $\{b, d\}$. The restrictions of the process are such that it produces the language $\{a, ab, ac, abd, acd\}$. Assume that taking action $d$ brings some value to the actor taking it, and $A$ and $B$ are competing to take the action. It is clear that $A$ must inform $B$ when taking the action $a$, since this makes $b$ enabled. However, if $A$ does not – they instead choose to inform no one, which is allowed under strong non-equivocation – they can continue to take the actions $c$ and then $d$, with $B$ never having the possibility to do so. Put in security terms, $A$ starves $B$ by attacking availability of $b$. Even worse, a malicious actor could also attack the global consistency of the system, forcing illegal runs: consider a process with the language $\{a, b, ac, bc\}$, and the actors $A$ and $B$, where $A$'s actions are $\{a, c\}$ and $B$'s actions are $\{b, c\}$. By taking action $a$, and omitting to tell $B$, $A$ can enable $B$ to still take $b$ and then $c$, giving the run $abc$, which is not part of the language, thus breaking global consistency.

These problems arise from the fact that, even if the TEE enforces correct local behaviour, the strong non-equivocation property cannot enforce correct global behaviour, since messages of changing enabledness of other actions can still be omitted[5]. This eventually reduces to the problem of two processors achieving coordination for a single value, when messages can be omitted. This problem is known as the *two generals problem* or the *two lovers problem*, and is known to be impossible [56, 57] and [58, Ch.3].

[5]Neither can weak non-equivocation, since it restricts the actors strictly less than strong non-equivocation

One way of circumventing this impossibility is to include more than two processors and use an omission-resilient uniform agreement algorithm (which requires more than two processors [59]), which is to say, use an approach like the SMR one described in Section 1.1. So, by applying the non-equivocation properties directly to an execution engine, we must more or less still solve an agreement problem. And here, one might utilise non-equivocation subsystems to reduce the fault tolerance for a cheaper and more resilient implementation, as shown in the paper in Chapter 3.

## 1.5 Introducing the Impalpable Differences paper

In this section, we present the paper *Impalpable Differences: Secret Actions in Processes and Concurrent Workflows*, available in Chapter 4. The paper is currently under peer review for publication. This section does not include a section on its relevance to adversarial collaboration as the relevance is self-evident.

The paper examines when an actor in a distributed process execution can consider an action secret. Recall the example in Section 1.1. A vital part of that process is to keep secret if *David LLC* chooses suite 1 or 2. So when can we consider such an action secret?

The paper presents a possibilistic secrecy definition of actions in process models with run-based semantics. The term possibilistic refers to the type of inferences we allow the adversary, namely only *possibilistic inferences*, meaning that we model an adversary as only knowing something if they can deduce it without reservations. This is opposed to *probabilistic inferences*, where we allow an adversary to know something if they can infer that it holds except for some negligible probability.

To define secrecy of actions, we first define an indistinguishability relation on runs: when can an actor not distinguish between two distinct runs of a process? This, of course, depends on what we allow the actor to know in each run. If they know the sequence of actions in the run[6], then they can distinguish all distinct runs from each other. If, on the other hand, they know nothing in any run, then all runs are indistinguishable. This notion of what an actor knows is captured in an *observation function*. Each actor in a process has their own observation function, limited by a universe of possible observations, and the subset of actions they can observe.

[6] as actors do in when using an agreement algorithm each time they take an action, see Section 1.1

An obvious observation function for an actor in the process model presented in Section 1.1, would be the sequence of the sets of enabled and disabled actions that the actor experiences throughout the run, along with the actions they take themselves. E.g. if taking action $a$ makes $a$ disabled, then an actor taking the action $a$ records that they took $a$ and that $a$ is now disabled. When another actor then takes an action that makes $a$ enabled again, then the first actor records that $a$ has become enabled again.

It is now easy to see when two runs are indistinguishable, namely when they produce the same observation. And then secrecy is relatively straightforward: if all runs where $a$ is taken at least once are indistinguishable from some runs where $a$ is not taken, then $a$ can be considered secret from the observing actor.

We apply the secrecy definition to DCR graphs and show that determining indistinguishability, in general, is infeasible by a reduction of *reachability*, i.e. determining if an action can ever be enabled, which is known to be a computationally hard problem [60]. We then go on to show a sufficient condition of when actions in a DCR graph are secret to some actor, namely when the action is non-trivially isomorphic with another action in the graph, and neither is observable by the actor in question. It is then easy to see that choice of suites 1 and 2 are secret to *Goliath Inc.* in Figure 1.1 since there exists an isomorphism where *Choose suite 1* maps to *Choose suite 2*, and vice versa, and the two certification actions map to each other in a similar manner.

Unfortunately, secrecy is directly at odds with the solution guaranteeing consistency based on an agreement algorithm: by requiring agreement by all actors on all actions taken, no action can be secret, since all runs are distinguishable from each other. We leave a further investigation of this issue as future work.

## 1.6 Discussion: Lying in a distributed business process execution

In Section 1.3 and Chapter 2 we present a result on how to prevent lying in a distributed system. This is transferable to prevent lying in a distributed business process execution, but some major distinctions need to be made.

First, we must define what lying is. For the purposes of this discussion, we can consider a lie any announcement of a falsehood. Doing so, we disregard the motivation behind the lie which is important for

more fine-grained discussions on lying in computer systems, see e.g. [61] and [62, Ch. 17], but the definition suffices for this discussion: in a distributed system, a message is either truthful and correct or it is a lie

When a malicious actor lies in a distributed business process execution, it is different from when a malicious processor lies in a generic distributed algorithm. A distributed business process is usually tightly coupled with non-digital actions; actual people are doing something and recording it in the process execution engine, and other people then react to that action by taking other actions and so forth, often with computers taking only a few or no automated actions during a run. This is more rarely the case with other distributed algorithms, e.g. agreement algorithms. While the inputs to an agreement algorithm can be provided both by computer systems (e.g. sensors), or people, the intermediate steps between input and output are rarely observed by people, and even more rarely observed *during* the execution of the algorithm. Sure, you can run an agreement algorithm with people acting as the processors of the algorithm, but this is surely an exercise with few practical use cases. For distributed business process models on the other hand, the tight coupling between a run of a business process and the real world has impacts on how lies may be executed and caught in a distributed business process execution.

Firstly, the tight coupling allows for malicious actors to lie in a new manner that is difficult to detect, namely by acting non-maliciously in the distributed business process execution engine, but not letting it represent their real-world actions. Doing so, they might act as if they have taken some action in the execution engine but actually have not taken this action, or even taken some other action instead, in the real world. This is a problem that extends beyond distributed business process execution to any problem where we attempt to use computers to represent and coordinate the state of real-world objects and behaviour. It is especially prevalent in blockchain supply chain solutions. We will discuss this issue below.

Secondly, the tight coupling allows us to verify some actions. When an actor indicates they have taken an action in the real world, it allows us to go check if this is actually the case. Similarly, if an actor attempts to hide that they have taken an action in the process, we might be able to observe them taking the action in any case. This added verifiability can mitigate some of the added threat from point 2, and even some of the more traditional threats from lying by acting maliciously inside the execution engine.

### The digital-real-world disconnect

It is important to distinguish between taking a step in a process in the execution engine of that process, versus taking a step in that process in the real world. Consider the example process from Section 1.1. After *David LLC* has chosen suite 1, *IT Guard* could easily claim to have installed the suite correctly, without actually having done so, thereby undermining the entire purpose of the process. This disconnect between what is recorded digitally vs. what has happened in the real world is well-known. The disconnect often crops up as a challenge for blockchain implementations of supply chains, see e.g. [63][7], [64][8] and [65][9]. Going forward, we will refer to the disconnect as the *digital-real-world disconnect.*

The digital-real-world disconnect is a major obstacle to secure and trustworthy implementations of process execution engines when the collaborators might be adversarial. Even if a benign collaborator can trust the execution engine to ensure that no actor can act maliciously, they might be deceived into taking steps that they would otherwise have preferred not to, by a malicious collaborator who indicated that they took steps in the process they did not, in fact, take in the real world. In the above example, *Goliath Inc.* could be a victim of such a scheme if *David LLC*, in malicious collaboration with *IT Guard*, took the step of notifying *Goliath Inc.* of the correct installation, while they had no installation made in the real world.

The digital-real-world disconnect does not make process execution engines useless as tools for collaboration among adversaries, however. Taking the point of verifiability in distributed business process execution into account, at least the easily verifiable actions can be considered relatively safe from lies via the digital-real-world disconnect. Furthermore, actions that exist only in the process model, i.e. that have no real-world counterpart, can likewise be considered secure. This includes, for instance, the actions that deal only with the coordination of the process itself – the *choreography.*

[7] "block chain veracity is reliant on appropriate audit processes to verify each transactional record to ensure it is accurate at the time it is entered into the blockchain"

[8] "one of the primary challenges of blockchain supply chains, and indeed blockchain generally, [is] the problem of the quality of data entered into a blockchain"

[9] "Assuring integrity of input data is a difficult task"

### Circumventing the digital-real-world disconnect

One way to increase the usefulness of a process execution engine in the face of the digital-real-world disconnect is to implement the process in terms of giving information to other parties. That way, if the execution engine, and the information in it, is trustworthy, then taking an action in the process *becomes* taking the action in the real world by way of the execution engine.

One benefit of multi-actor business processes is that the actions that are important to multiple actors generally are part of the choreography and deal with coordinating the order of those actions. In the example in Section 1.1, the entire process is part of the choreography, since all actions affect, or are affected by, the actions of other actors, and so require coordination.

A *coordination model* in distributed computing refers to a computational model where we describe protocols as the messages exchanged by the processors, rather than the computational steps taken by each processor [66, Ch. 13]. Applying this concept to business process models, a *coordination process* is a process described in terms of the communication between the actors in that process. It is immediately clear that not every part of a collaboration can be rewritten into a coordination process. E.g. in the Public-to-Private approach (P2P) [9], where actors share a public part of a business process, but have their own private extension of this public part, the actions that do not affect the public parts of the process require no communication with other actors.

In the example in Section 1.1, *David LLC* could have an entire internal process for choosing between suites 1 and 2, which then could not be modelled as a coordination process[10]. However, the choreography *must*, by definition, be expressible as the communication between the actors in the process, since this is what they necessarily represent. And so we can model this crucial part of the process as a pure information exchange, thereby circumventing the digital-real-world disconnect, since they have been merged with their real-world counterpart.

This is the way we have designed the example process in Section 1.1, so let us use that as an apt example: when *Goliath Inc*. takes the action *Accept list*, and that fact becomes known to *David LLC* via the execution engine, then *Goliath Inc*. has for all intents and purposes accepted the list in the real world. They might change their mind, and try to recall their acceptance, but they cannot repudiate that they did, in fact,

[10] Unless one also models the internal departments of *David LLC* as actors in the process.

issue an acceptance in the first place. The same holds for all the other actions in the example process: when *David LLC* notifies *Goliath Inc.* of certification, then taking that action is *the same thing* as notifying *Goliath Inc.* of certification in the real world. The execution engine simply becomes the platform for notification.

Perhaps the actions of the security companies are the most non-trivial examples of actions modelled as information – installing a security suite is not just an exchange of information between collaborators. This is why the actions have been modelled as *certifications*: a certificate is a formal document attesting some fact, and so it is, in fact, pure information given to other parties: "*IT Guard* hereby attests that *David LLC* has had our security suite correctly installed". However, the usefulness of this modelling method also stops here. While the certification itself may be pure information and coordination, it still refers to actions having been taken in the real world. These real-world actions are not modelled in the business process model and so one might get the impression that it is safe from the digital-real-world disconnect; it is not. A certification is a guarantee given by a collaborator that some action has been taken in the real world, but one should only trust such a certification as far as one trusts the actor giving it and the risk they run of being caught in a lie.

It seems that modelling actions as information exchange is no silver bullet against the digital-real-world disconnect. If the information exchange action becomes a *substitute* for a action in the real world, then a malicious actor will still be able to lie about the real-world part of that action. By modelling actions as information exchange, we can only make a guarantee about what actors *say* they do, not what they actually *do*. However, this part of the digital-real-world disconnect extends beyond digitalisation and is rooted in all collaborative efforts, regardless of whether they use a digital medium such as an execution engine for coordination. In an adversarial collaboration where coordination is handled by e.g. all collaborators meeting weekly and coordinating their efforts, the problem of handling discrepancies between what an actor says and what they actually do still exists. We can still mitigate this risk, however, by utilising the verifiability of actions, and trusted third parties.

One of the benefits of the coupling between an action in a business process model and the real world is that we can verify whether an action has actually taken place. In the recurring example from Section 1.1, *Goliath Inc.* can verify that the same flaws are not present after being

notified by *David LLC* that a new security suite has been installed. Similarly, when a company requires payment for goods, the company can usually verify payment before finalising the transfer of goods. When an insuree makes a claim to their insurance company through their digital portal, the insurance company usually requires proof of the claim, e.g. pictures or a police report, which can be uploaded through their portal.

However, not all actions are easily verifiable. In the recurring example, it is not easily verifiable to either *Goliath Inc.* or *David LLC* if a suite has been installed *correctly*. For the company requiring payment, it may be that requiring payment before transferring goods is prohibited by law. And a claim to the insurance company could require expert knowledge to determine the exact size of the compensation. In these cases, the actors of the processes can use trusted third parties to eliminate any discrepancies between what an actor says they do, and what they actually do. This is the purpose of the certification actions in the recurring example. The insurance company could, for instance, use a mechanic to verify the extent of the damages, if the claim was regarding a car accident. In such cases, the actor substitutes the difficult-to-verify action with a certification from a trusted third party. It is, of course, vital that such a third party is actually trustworthy since any malicious behaviour negates the guarantees of the process.

In conclusion, there are 3 ways of mitigating the risk of lying by utilising the digital-real-world disconnect in a distributed business process execution. First, actions can be modelled as pure information exchange, allowing for the use of the results in this thesis to secure the actor from lying about the action. One should take care to not simply move the problem into the real world again by doing so, though. Secondly, by verifying easily verifiable actions, one can gap the disconnect and ensure no discrepancies between what an actor says they do and what they actually do. Finally, if neither of the other two possibilities is possible, one can outsource the action to some trustworthy third party. It should be noted that both verifying actions and using trusted third parties reduce the usefulness of distributed business process execution engines to a degree; both mitigations require more real-world work and lessen automation. As such, the cost of mitigating risks from the digital-real-world disconnect must be weighed against the cost of potential adversarial behaviour by collaborators. One question that is left is how far one can push the idea of modelling collaboration as an information exchange before either of the other two possibilities are necessary. We leave this as future work.

## 1.7 Future work

In this thesis, we take steps to prevent malicious behaviour in the execution of a distributed business process. However, the steps taken are by no means exhaustive. They rely on strong assumptions, that may not always hold true for practical purposes. We have likewise left several avenues of research for the future. In this section, we present and discuss four major questions left unanswered in this thesis, which can serve as a point of departure for future research into the subject:

1. How can we prevent malicious behaviour in partially synchronous or asynchronous distributed business model executions? Which tools can we reuse, and where do we have to invent new tools?
2. Can malicious actors act maliciously in other ways than lying, cheating and stealing?
3. Does modelling the choreography as information flow allow for further circumvention of the digital-real-world disconnect?
4. How exactly are consistency and secrecy opposed, and how far can we push each without violating the other?

#### Partially synchronous and asynchronous systems

The first and most significant assumption made in this thesis is the synchronicity of the system model. The synchronous system model is generally considered a strong assumption due to its impractical, and sometimes even impossible, implementation details. As such, synchrony is a major limitation to the usefulness of the results in this thesis. Fortunately, not all the results presented requires the strong assumption of complete synchrony: the transformation of Byzantine faults using a TEE in Chapter 2 is independent of synchrony.

Similarly, the notion of secret actions in Chapter 4 is defined without any assumptions of synchrony. It is important to factor such assumptions into the observation notion, however, since observing a run in an asynchronous system may lead to lost or reordered observations of run fragments. This translates into the observation function being inherently non-deterministic, i.e. a random function, in such a system model. While this, of course, *must* be accounted for in specific uses of the definition, it does not present a problem for most systems where the observation in a synchronous version of the system is still possible in the asynchronous system; if an action is secret for a deterministic observation function,

then, intuitively, it must also be secret in a randomized version of that function, when all images in the random function must contain their counterpart in the deterministic function. This means, however, that one should take care not to assume an action to be secret when this is modelled under asynchronous assumptions, since these actions may not be secret in systems with a high degree of synchrony. It may also be prudent to examine further probabilistic notions of secrecy for use in asynchronous systems since there clearly is an element of probability by virtue of the random function.

The results regarding non-equivocation in Chapter 3, have less use in asynchronous or partially synchronous systems. Inherently, the difference between strong non-equivocation and weak non-equivocation lies in the possibility for actors to omit messages in the weak version and not in the strong one. And it is imperative to the analysis of weak non-equivocation that a processor can determine that another processor is faulty by the omission of a message. But if messages can be delayed indefinitely, as they can in asynchronous systems, then this is indistinguishable from the actor omitting the message, making both the distinction and the analysis of the properties invalid. A need for further study of how these properties behave for distributed business process models in asynchronous and partially synchronous systems is indicated.

### Other malicious actions

The second significant assumption we have made is that the described malicious actions are exhaustive: that a malicious actor can only lie, cheat by equivocation, or attempt to steal secrets. However, it may be that this list is, in fact, not exhaustive. Heindel and Weber [32] describe a model of a malicious actor that follows the prescribed process and conserves consistency, but whose goal is to achieve a specific outcome in the process. By choosing their action appropriately, such a malicious actor may force other actors to make suboptimal choices, thereby achieving their goal. Such malicious behaviour is not explicitly covered under any of the assumptions for malicious behaviour. However, there does seem to be a link to the definition of secrecy of actions: both types of malicious actors follow the described process. Also, for a malicious actor to force other actors to take certain choices, they must be aware of which action the other actors take, or at least the effect such actions have on the process as a whole. For two or more actions that are secret by virtue of being isomorphic as described in Chapter 4, Section 4.3, this may hold true. Actions that are secret for other reasons, though, could obscure

the process state as a whole to the malicious actor, and so may actually be a way to prevent or limit such behaviour. Further analysis of the interaction between action secrecy and such malicious behaviour looks to be a promising avenue of research.

In Sections 1.3 and 1.4, we saw that neither the TEE transformation, nor the non-equivocation properties could alone prevent active malicious behaviour of actors in an execution engine. Even the composition of the transformation with the strong non-equivocation guarantee was vulnerable to an attack on consistency where the adversary omitted messages that are supposed to update the enabledness of actions of other actors, due to the two generals problem. However, in the event of the invention of an execution engine that is resilient to (malicious) omissions, the results of Chapters 2 and 3 are immediately applicable.

Other malicious behaviours that may not have been accounted for, include malicious behaviour that exploits the concurrency of actions. It may be that there exists an attack on the described countermeasures for malicious behaviour if the malicious actor can make use of the inherent concurrency of actions. This is related to the notion of synchrony since concurrency implies some asynchrony in the system. We have discounted such attacks since concurrency must be handled by all solutions to the consistency problem. However, current solutions do not include asynchrony and do not suppose malicious behaviour, so it may be that this assumption is too strong.

### Extending the modelling of choreographies

In Section 1.6, we described how one could model the choreography as an exchange of information to circumvent the digital-real-world disconnect. This has some limitations, as described, and must be used with caution. The idea of modelling actions as a flow of information may also have important benefits, that can be extended and used beyond the choreography. Recall that the benefit of modelling actions this way is to decouple them from, or preferably even entirely merge them with, their real-world counterpart. As such it may be beneficial to the actors of processes with a lot of repetition or with many instances of the same process to apply this modelling to actions outside the choreography. A formal investigation into this way of modelling may yield important results in both creating more performant and more secure processes.

As an aside, there seems to exist a curious relationship between the choreography, information flow of a process, and secrecy of actions. Part of the research that did not make it into the paper in Chapter 4, was

the definition of *total secrecy* of actions. The definition in Chapter 4, Section 3, implies that an observing actor cannot determine if an action has been taken in any run where the action actually *has* been taken. The observing actor is allowed to determine that an action has *not* been taken in runs where it has, in fact, not been taken (e.g. the empty run). Total secrecy, on the other hand, implies that an actor can *never* determine if an action had been taken *or not*. The total secrecy property implies that *no information flows* from the totally secret action to the observing actor. This, in turn, implies that such actions cannot exist in the choreography between the two actors, since, by definition, information flows between actions in the choreography.

**Consistency vs. secrecy**

In Section 1.5, we describe how the secrecy of an action and the consistency of a global process using an agreement algorithm are directly at odds. This is due to the fact that, if an action is secret, then determining if an actor has taken that action must be impossible. Meanwhile, by using an agreement algorithm (see 1.1, actions are published when taken. This dichotomy between consistency and secrecy seems to extend partly beyond agreement algorithms: when an action is secret, it means that the amount of information flowing from one actor to another is limited or lacking when taking the secret action. Consistency, on the other hand, implies that information *must* flow between at least some actions in the process, otherwise the rules of the global process cannot be preserved.

One way to conserve some secrecy and still use agreement algorithms is to require only agreement on the public part of a process (assuming a P2P approach). There may be no need to ensure consistency of actions not affecting the public part of the process, since these only affect the actor's private process. Due to the digital-real-world disconnect, each actor can do as they please for such actions anyway; there is no need to communicate any information, the only actor that has any interest in verifying the action is the actor taking the action, and if a trusted third party is made to take the action, then the action invariably becomes part of the public part of the process. So the secrecy of such local actions may be possible to achieve by using a solution with an agreement algorithm. However, there are plenty of examples where we are more interested in keeping the actions that are part of public process secret. For instance, when *David LLC* chooses which security suite to protect its infrastructure. Other ways of circumventing this discordance could prove beneficial.

In Chapter 4, the assumption is that consistency is guaranteed: the actors cannot circumvent the rules of the global process. This naturally limits which actions can be secret to those where small amounts of information are required to flow between actors for consistency to be preserved. However, this leaves the question of how weakening the requirements of consistency might improve secrecy: if, for instance, we allow the system to enter inconsistent states that are guaranteed to eventually become consistent again at a later time, can we increase the number of secret actions? Might we even be able to achieve total secrecy of actions in the choreography? What other reasonable weakening of consistency might be beneficial to secrecy, and what exactly is the relationship between the two properties, in information flow terms?

## 1.8 Introductory conclusion

Thank you for following me this far. I trust that you have gotten a good overview of the results of the papers that make up this thesis, and how they relate to securing business process model executions. For good measure, I will summarise the major points here.

The main result of the paper in Chapter 2 is a transformation of omission-resistant distributed algorithms to Byzantine ones, by using a TEE. The transformation applies to adversarial collaboration by preventing malicious actors from actively lying about which actions they have taken. This application disregards the notion of the digital-real-world disconnect and does not prohibit malicious actors from attacking consistency by omitting messages. Another application of the transformation is the fact that it can reduce the cost of agreement algorithms – which can be used to solve the consistency problem – and increase their fault tolerance. However, the transformation includes some vulnerabilities and should be used with care.

The main result of the paper in Chapter 3 is an analysis of two different kinds of non-equivocation, and their application to agreement algorithms. At first glance, non-equivocation seems to have an application directly to adversarial collaboration, but we show that this application is rather negligible, due to the inherent two generals problem present whenever malicious actors can omit messages. However, their application to agreement algorithms makes them useful for any execution engine that uses this approach.

The main result of the paper in Chapter 4 is a rigorous definition of when an action is secret. This is directly applicable to adversarial

collaboration since it prevents malicious actors from extracting business secrets by observing the collaborative process.

Below follow the papers I have written during the course of my PhD project, from which stems the main results described in this introduction. I hope that you will enjoy reading them.

# Chapter 2

# Transforming Byzantine Faults Using a Trusted Execution Environment

# Transforming byzantine faults using a trusted execution environment

Mads Frederik Madsen, Mikkel Gaub, Malthe Ettrup Kirkbro, and Søren Debois
Department of Computer Science
IT University of Copenhagen
{mfrm,mikg,maek,debois}@itu.dk

***Abstract*—We present a general transformation of general omission resilient distributed algorithms into byzantine fault ones. The transformation uses the guarantees of integrity and confidentiality provided by a trusted execution environment to implement a byzantine failure detector. Correct processes in a transformed algorithm will operate as if byzantine faulty processes have crashed or their messages were dropped. The transformation adds no additional messages between processes, except for a pre-compute step, and the increase in states of the algorithm is linearly bounded: it is a 1-round, $n = f + 1$ translation, making no assumptions of determinism.**

***Index Terms*—Byzantine faults, translation, TEE, TPM, SGX**

## I. Introduction

In the byzantine fault model, we know that $3f+1$ processes are necessary to solve the consensus problem [1]–[3], while $2f + 1$ processes are sufficient to solve the problem in the crash-fault model [3]. By *solve*, we mean with some relaxation of either liveness, asynchrony or determinism, to circumvent FLP impossibility [4]. Using hybrid fault models, where *some* subsystems are assumed to fail only by crashing, new solutions have achieved $2f + 1$ fault tolerance [5]–[10]. The trusted subsystem prevents a process from *equivocating*, the action of sending contradicting messages, which increases the fault tolerance from $3f + 1$ to $2f + 1$ [5,11].

These small trusted subsystems take different forms. Most notable are the attested append-only memory (A2M) identified by Chun et. al. in [5], and the authenticated monotonic counter (TrInc) identified by Levin et. al. in [11]. The solutions using these trusted subsystems make the common assumption that the component, in fact, does not experience byzantine faults, but fails only by crashing.

We show how any algorithm tolerant to general omission faults and unreliable channels can be transformed into a byzantine fault-tolerant algorithm. Note that while the transformation works regardless of synchrony, it does require the underlying algorithm to be resilient to omission faults *also* in the synchronous setting. It is also assumed that the transformed algorithm is correct before the transformation takes place.

The transformation operates by moving the entirety of the algorithm into a trusted execution environment (TEE). We do not assume that the TEE is free from byzantine faults, but rather use its integrity and confidentiality guarantees. Intuitively, the integrity guarantee simulates a perfect local byzantine fault detector oracle. which can be used to translate any such fault to a crash or a message omission. Meanwhile, the confidentiality guarantee ensures that, given a shared secret between the integrity protected processes, no process can falsely pass itself off as being integrity protected.

The transformation works on both synchronous (but omission resilient) and asynchronous algorithms; transforms both deterministic and randomised algorithms; introduces no overhead on the number of messages past pre-computation; imposes only a small constant overhead on message size; and preserves the algorithm's fault tolerance. To our knowledge, no transformation with these properties currently exists.

The overhead of transformations are traditionally measured in rounds and fault tolerance (e.g. [12]–[15]). E.g. a 4-round, $n = 3f + 1$ transformation converts one round of message send/receives in the algorithm into four and requires that at most $\lfloor \frac{n-1}{3} \rfloor$ processes exhibits faults. (A translated algorithm will have the lower fault tolerance of the transformation and the original.) We say that a transformation is a crash to byzantine transformation if it can transform a crash fault-tolerant algorithm to be byzantine fault-tolerant and that the transformation translates byzantine faults to crash faults if a byzantine fault on one process presents as a crash fault to the transformed algorithms on other, correct, processes. Using these terms, this paper presents a 1-round, $n = f + 1$ general omission to byzantine transformation.

This paper comprises 5 sections: in Section II we present related work; in Section III we give our system model; in Section IV we present and prove correct our core transformation; and in Section V we give an example application.

## II. Related work

Automatically transforming or translating algorithms to have fault tolerance in stronger fault models is a well-known idea. Our improvement is to extend such transformations with the use of trusted hardware, as trusted hardware has shown some promise in eliminating byzantine faults. E.g., Attested Append-only Memory [5] and Trusted Incrementers [11], which eliminate equivocation in byzantine consensus systems; the PoeT consensus algorithm for Hyperledger Sawtooth [16], where mining is replaced with a proof of elapsed time from trusted hardware; and by the blockchain-based cloud service provider iExec [17], where trusted hardware secures confidentiality of data computed at untrusted remote actors.

Transformations generally focus on the translation of faults and are therefore referred to as translations. Translations are

distinguished by the synchrony of the system in which they are applied, since they, usually, cannot translate faults in both synchronous and asynchronous systems.

*Synchronous translations.* These include the composable translations of [12], where the composition of auxiliary translations results in two crash-to-byzantine translations: a 4-round, $n = 4f + 1$ translation and a 6-round, $n = 3f + 1$ translation. The translations differ in their broadcast primitives (reliable, validated, etc.). These results were improved upon in [13], which presents 2, 3, and 4-round translations, with fault tolerance related to the round-increase: $n > \max(6f - 3, 3f)$, $n > \max(4f - 2, 3f)$ and $n = 3f + 1$, respectively. Later [18] proposed a translation assuming each process has a unique cryptographic signature and that at most one in any replicated pair of a process fail simultaneously, achieving under these assumptions a crash-to-authenticated-byzantine translation.

None of these translations caters to randomized algorithms: they all rely on comparing deterministic behaviour to check whether a process is exhibiting byzantine faults.

*Asynchronous translations.* In this setting, [19] proposes a crash-to-byzantine translation, by building a reliable broadcast primitive which filters messages to present byzantine faults as crashes. The protocol requires three broadcast rounds and has a fault tolerance of $n = 3f + 1$. Coan [14] builds upon this result and proposes a 2-round, $n = 4f + 1$ translation, and a 3-round, $n = 3f + 1$ translation; both making asynchronous crash fault-tolerant algorithms into byzantine fault-tolerant ones. Ho et. al. [20] proposes an asynchronous crash to byzantine translation using an ordered[1], authenticated and reliable broadcast primitive, yielding a fault tolerance of $n = 2f + 1$ if the primitive is built using cryptography, or $n = 3f + 1$ if it is built without cryptography. The broadcast primitive requires 3 rounds for each original message.

Lastly, our translation is related to that of Clement et. al. in [15], where it is shown that the non-equivocation given by a trusted subsystem is not enough to ensure the $2f + 1$ lower bound in the hybrid fault model, but that transferable authentication—e.g., digital signatures—is needed as well. In this setting, they create a 1-round, $n = f + 1$ crash to byzantine translation using a trusted subsystem that ensures non-equivocation and transferable authentication. Our translation distinguishes itself on the following points: their translation can translate asynchronous algorithms, while ours can translate both synchronous and asynchronous algorithms; they assume that the trusted subsystem can only fail by crashing, while we assume only the integrity and confidentiality of a TEE; their translation requires the algorithm to be deterministic, as their primitive requires simulation of the sending process' underlying state machine by the receiving process, while ours require no such simulation, and thus can translate both randomized and deterministic algorithms; each sent message from a process in their translation includes all previously received and sent messages to enable complete simulation of the sending process by the receiving process, making the message size grow linearly with the number of rounds. In our transformation, the messages have the constant overhead of a Message Authentication Code tag (MAC).

[1] *FIFO*-ordered per sending process, and not ordered between distinct processes, since this would make the construction of such a primitive reduce to a consensus problem.

## III. System Model

We will now present our system model, which is related to that of [20], where the traditional system model is extended with several state machines per process, each with separate state and progression. We will not, however, model channels as separate state machines, but instead, use the abstraction of channels as passive components for delivering messages.

### A. Processes and channels

A system comprises *processes*, *state machines* and *channels*: A distributed algorithm consists of $n$ processes $\{p_0, \ldots, p_{n-1}\}$, each containing a set of state machines. State machines on the same process may be connected with reliable FIFO-channels; whereas state machines on separate processes can be connected only with unreliable channels, which may drop, reorder, duplicate, corrupt, or redirect messages. We make no synchrony assumptions about channels and processes, as we solely use the assumptions of the underlying system.

"State machines" here are any kind of automaton that consumes inputs and produces outputs. A simple example is Mealy Machines; however, we emphasise that our transformation also applies to more powerful machines such as pushdown automata or Turing machines. Note that some of the machines introduced by our translation—notably the wrapper machines and the machines appending MAC codes to messages— can *not* be adequately represented as finite state machines, but require more powerful computational models.

Formally, a *state machine* in this paper consists of a set of states, transitions between the states, and an initial state. (When state, transitions, and alphabet of actions are all finite, the machine is a Mealy Machine.) Transitions are on the form $(s_0, t_{in}) \rightarrow (s_1, t_{out})$, where $s_0$ and $s_1$ are, respectively, the beginning and end states of the transition. $t_{in}$ and $t_{out}$ are tuples of the form $(B, c)$, where $B$ is an arbitrary length bit string, and $c$ is a named channel. $t_{in}$ is an input message on channel $c$ required for the transition to initiate, and $t_{out}$ is the output message produced and sent via channel $c$ as a consequence of the transition. We use $\emptyset$ to indicate that a transition does not require an input or an output.

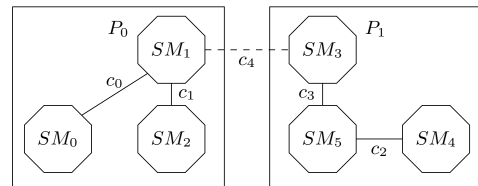



Fig. 1. Simple example of processes and channels in our system model.

Figure 1 shows an example of processes and channels. This simple setup has 2 processes, $P_0$ with the state machines $SM_0, \ldots SM_2$ and $P_1$ with state machines $SM_3, \ldots SM_5$. The

state machines of $P_0$ are connected with the reliable channels $c_0$ and $c_1$, while the state machines of $P_1$ are connected with the reliable channels $c_2$ and $c_3$. The processes are connected through the unreliable channel $c_4$ between $SM_1$ and $SM_3$.

This model is inherently non-deterministic: state machines of a process can be in different states across otherwise identical runs. However, we do not rely on the determinism of an algorithm in our transformation, and our transformation does not introduce non-determinism.

### B. Faults

As mentioned, channels between processes are unreliable: messages may be dropped from these channels. We use channel omissions as a proxy for all omission faults in the system, including send- and receive omissions. Note that this convention makes it impossible to detect where the message omission has taken place, a key problem in general omission resilient systems [21]. Formally, we model a crash failure by an $\varepsilon$-transition to a special state $s_e$, with no outgoing transitions. In this state, a state machine can neither transition away, receive inputs, nor create outputs, altogether behaving as if crashed.

To model byzantine faults, we must allow arbitrary behaviour. To this end, we model a byzantine fault of a process as the removal, addition or substitution of any number of that process' state machines with arbitrary replacement machines. An initially faulty process can be perfectly approximated by such a substitution occurring at the very beginning before any communications or internal actions take place.

We make conventional assumptions about cryptography: a byzantine fault cannot produce faults requiring the simulation of secrets—so no byzantine fault can produce new messages with correct Message Authentication Codes (MACs).

### C. TEE guarantees

A TEE is a subsystem[2] providing Authenticity, Integrity, Confidentiality and Remote Attestation [22]–[24]. **Authenticity** is the property that a program saved to persistent storage and later loaded into a TEE, loads successfully into the TEE only if it is the unaltered original program. In other words, the program cannot be changed after it has been compiled into a TEE compliant binary. **Integrity** is the property that only a program running in a TEE can change the data in the memory of a TEE, and the program can only change the data in the TEE that has been allocated to it. This integrity property still holds while the data resides outside the TEE, e.g. in persistent storage. We presently work with a weaker integrity property: unauthorised changes to data inside the TEE cause the immediate loss of all cryptographic secrets in that TEE. We choose this weaker model because nothing prevents a byzantine process from trying to impersonate a newly crashed process. In the worst case, the imposter process would replicate the crashed process' state, making it indistinguishable from the crashed process from the point of view of other correct processes, except for any confidential secrets the crashed process might have had. **Confidentiality** is the property that data created in the TEE can only be read by a program running inside the TEE, both during program execution and when residing in persistent storage. Furthermore, a program inside the TEE can only read its own confidential data. **Remote Attestation** is the property that a program running inside a TEE can *prove* its Authenticity to remote hosts. We will assume that the Remote Attestation property enables a confidential and integrity-protected exchange of symmetric cryptographic keys between TEE programs, as seen in e.g. [25].

[2]Note, we are not referring to the implementation of a TEE, which can be achieved in several ways, but the properties implementations have in common.

For ease of modelling, we let Integrity encompass Authenticity, i.e. we view programs as data generated by an application running inside a TEE, which means that we will be given Authenticity as a by-product of Integrity.

We note that for practical TEE implementations, the Integrity property is up to common assumptions about cryptography, e.g., the TEE implementation *Intel® Software Guard Extensions* (SGX) detects integrity violations except with negligible probability under the assumption that AES128 is a random permutation [26]. Our notion of Integrity conforms to the one of SGX, where the processor will halt entirely on an integrity fault. Nothing prevents a byzantine process from trying to impersonate a crashed process, but the integrity property will ensure that the impostor process will not be able to replicate the confidentiality-protected secrets.

To model these properties, we need three more concepts:

1) An integrity protected area of each process.
2) A state machine ($SM_c$) for each such area in which all cryptographic secrets resides.
3) An attestation process ($P_{RA}$) and an attestation state machine ($SM_A$). An $SM_A$ resides on each process, and together with $P_{RA}$ enables Remote Attestation.

The state machine $SM_c$ enjoys the Confidentiality property: so only $SM_c$ may access cryptographic secrets residing on the process. We assume that no other process, including ones arising from byzantine faults, can "guess" these secrets. The integrity protected area enjoys the Integrity property in the sense that if it encounters a byzantine fault—if one of its state machines is substituted—the $SM_c$ machine disappears.

Details of Remote Attestation varies with the implementation, e.g., the GlobalPlatform standard has no standardised Remote Attestation mechanism, but supports different implementations [27,28]. We assume that $SM_A$ and $P_{RA}$ are able to perfectly attest to state machines in the integrity protected area, i.e. uniquely identify the state machines and their states and verify that they are located in an integrity protected area. Moreover, we assume that, as part of that attestation, the remote attestation process is able to securely provision $SM_c$ with a shared symmetric secret. These assumptions are supported by, e.g., Remote Attestation in SGX [29,30], which relies on trusting an *Intel® Attestation Server* (IAS).

As some implementations of TEEs have limited access to hardware peripherals, we do not allow channels from the integrity protected area of a process to another process. Instead, we introduce state machines outside the integrity protected area that has a reliable channel into the integrity protected area, and

an unreliable channel to another process. We name these state machines *wrapper state machines* or simply *wrappers*.

### D. Weakening of channels

To allow failing processes arbitrary behaviour, we allow correct state machines to receive messages from all state machines on faulty processes. Messages from faulty processes can be received on any and all channels by state machines on correct processes. The only exception we allow is that faulty state machines outside the integrity protected area cannot send messages on channels with both endpoints inside an integrity protected area. This is a realistic assumption: communication between modules in the same TEE is usually implemented by shared memory, which also resides inside the TEE, and thus is protected from tampering by modules outside the TEE.

### E. Integrity violations vs. byzantine faults on SGX

We explain in more detail, using Intel SGX as an example, exactly in what sense we can construct a byzantine failure detector using SGX primitives. The primary protection against integrity violations in SGX is an integrity tree with the root stored in SRAM on-die [26]. All integrity-protected memory is MAC'ed in blocks collected in a Merkle-tree, with the root of this stored in on the CPU itself. When reading data from memory, MACs are verified against the Merkel tree: If they do not verify, the CPU halts, requiring a physical reset.

However, Gueron notes in [26] that, both the CPU *and its caches* are trusted components, i.e., reside within the "Trust boundary perimeter", which results in some confidentiality attacks [31]–[33]. It follows that SGX might not protect against faults taking place in the CPU cache, nor any affecting the CPU itself. To the best of our knowledge, the integrity attacks that exists on SGX requires a vulnerability in the program running in the enclave [34,35], which we assume is not the case for the algorithm to be transformed.

Regardless, this mechanism protects against physical and software attacks on memory: No process, including other SGX processes, can (except with negligible probability) modify data in the integrity protected memory without the CPU halting [29]. This is an argument that *SGX can detect and protect against byzantine faults that change code and main memory after process initiation*. Note that a "byzantine fault" here may not "guess" secrets encapsulated in SGX.

## IV. TRANSFORMATION

We now present the transformation of general omission tolerant algorithms to byzantine fault-tolerant algorithms, under the assumption that the algorithm handles unreliable channels.

The core idea of our transformation is to move the entire set of state machines into the integrity protected area, thus guaranteeing the integrity of the algorithm. However, we will need to ensure the integrity of all messages between the protected areas, both on the unprotected areas of the processes and while in transit on the channels. To this end, the transformation involves modifying both processes and individual state machines. On the process-level, we move constituent state machines into the integrity protected area; on the individual state machine level, we sign messages from the state machines to ensure integrity during transmission.

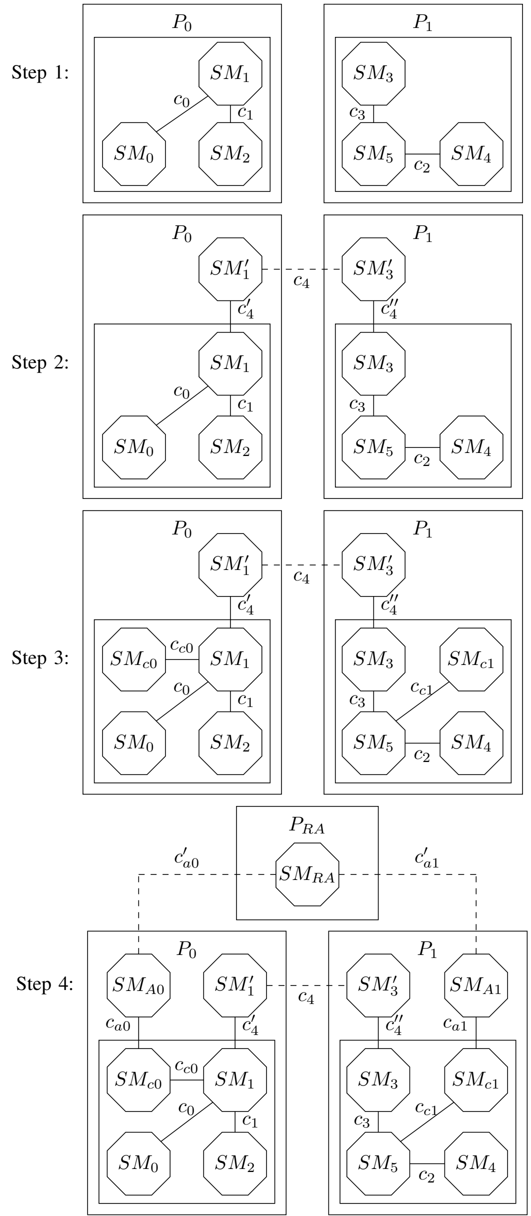



Fig. 2. Visualisation of the process-level transformation of Figure 1.

The transform is in 6 steps, where steps 1–4 adds integrity protection and remote attestation; and 5–6 implements byzantine failure detectors by validating such attestations.

### A. Stage 1: Integrity protection & remote attestation

These are steps 1–4:

1) Encapsulate state machines in their process' integrity protected area.
2) Add wrapper machines and connect them to the integrity protected state machines, and to each other.
3) Add $SM_c$, a machine hosting secrets, to each process.
4) Add $SM_A$ to all processes, and connect them to $P_{RA}$.

Note that the machines added in steps 2–4 are not finite. We visualise these steps for a very simple protocol in Figure 2.

*Step 1* moves the state machines of each process into the integrity protected area of that process, visualised in Figure 2 as a box inside the processes. We temporarily remove channels to external processes. *Step 2* adds back the missing channels via a "wrapper" state machine residing *outside* the integrity protected area of a process. For each (unreliable) inter-process channel, we add (1) a wrapper at each endpoint, (2) an (unreliable) channel with endpoints in the wrappers and (3) a (reliable) channel from each end-point wrapper to the original state machine end-point. *Step 3* adds a cryptographic state machine $SM_c$ to all integrity protected areas. $SM_c$ will store any secrets, will MAC messages to be transmitted, and will be cleared of secrets if a byzantine fault is detected in the integrity protected area. *Step 4* adds remote attestation: A machine $SM_A$ on all processes and a $P_{RA}$ process.

### B. Stage 2: Detecting byzantine failures

All processes must first complete a pre-compute step to participate in the transformed algorithm. In the pre-compute step, the processes are (a) remotely attested and (b) provisioned with a shared cryptographic secret. All processes will share the same secret after completing step (b). These steps do not presuppose synchrony or eventual delivery: Any message of the remote attestation being infinitely delayed is equivalent to the first message of the underlying algorithm being infinitely delayed. With this shared key, a Message Authentication Code (MAC) is appended to messages, which verifies to other processes that the message was constructed by a non-failed integrity protected state machine. Because we assume that byzantine faults manifest as the loss of cryptographic secrets, a correct MAC guarantees that the message originated from a correct process. Thus, steps 5–6:

5) MAC each outbound inter-process message sent from within the integrity-protected area. We transform each transition $(s_0, t_{in}) \rightarrow (s_1, (B, c))$ into the two transitions:

$$(s_0, t_{in}) \rightarrow (s', (B, c_c))$$
$$(s', (B|| \mathsf{MAC}(B), c_c)) \rightarrow (s_1, (B|| \mathsf{MAC}(B), c))$$

Here $c_c$ is a reliable channel to $SM_c$, $s'$ is a new state, where the state machines wait for a reply from $SM_c$, and $B|| \mathsf{MAC}(B)$ is $B$ appended with a valid MAC tag.

6) Verify the MAC on each inbound message. Each transition of the form $(s_0, (B, c)) \rightarrow (s_1, t_{out})$ becomes:

$$(s_0, (B||MAC(B), c)) \rightarrow (s', (B||MAC(B), c_c))$$
$$(s', ("1", c_c)) \rightarrow (s_1, t_{out})$$
$$(s', ("0", c_c)) \rightarrow (s_0, \emptyset)$$

Here $s'$ is an intermediate state in which we await a reply from $SM_c$. If $SM_c$ replies that the MAC is valid ("1") we proceed to $s_1$; otherwise, the MAC is invalid, and we return to the state $s_0$. In the latter case, the machine behaves *as if* no message was ever received.

Note that the machines added in steps 5 and 6 are not finite. Step 5 and 6 compose; if a transition has inter-process messages as both input and output, the transition is transformed by step 5, and the resulting transition with the inter-process output is then transformed by step 6. Note that if $SM_c$ handles requests in order, neither step 5 nor step 6 introduce new non-determinism, because both steps use a waiting state, such that the requesting state machine must receive a response from $SM_c$ before continuing. The entirety of the transformation introduces no new non-determinism since each external send and receive have simply been broken up into smaller atomic steps, during which the state machine is waiting.

### C. Overhead

The number of new states and transitions are both linearly bounded by the number of messages to/from state machines on other processes. Each inter-process message send adds one state and one transition; each inter-process message receival adds one state and two transitions.

Counting messages, each inter-process message in the original system incurs additional intra-process messages in the transformed system for MAC'ing (2), for the sending and receiving wrappers (2), and for MAC verification (2); altogether 6 messages. Intra-process messages incur no overhead.

An algorithm with $m_e$ inter-process (external) messages and $m_i$ intra-process messages, the transformed system will use $m_e$ inter-process messages and $6m_e + m_i$ intra-process messages, plus the messages for the initial remote attestation and provisioning pre-compute step. Note that the overhead is all intra-process: barring the initial remote attestation, the number of inter-process messages, messages *between* processes, does not increase. In practice, each inter-process message has only the small constant-size overhead of a MAC. We leave an empirical investigation of overhead as future work.

### D. Correctness

We show how a byzantine fault in any part of the process will be translated to either perpetual or sporadic message omission. A failed process with perpetual message omission seems crashed to all other processes since it cannot in any way affect the other processes in the system. We call this failure mode *constant omission*. Note that a single byzantine fault might affect more than one state machine and thus several different types of state machines on the same process.

*Integrity protected state machines.* Recall that a byzantine fault consists of the removal, addition, or replacement of machines; and that such a fault causes the removal of the $SM_c$ state machine containing secrets of the integrity protected area. With the lost secrets assumed unguessable by new state machines, no machine in the failed process will be able to produce MACs (Step 5). State machines of correct processes will, therefore, behave as if they received no messages from the failed process (Step 6): all correct non-wrapper state machines experience constant omission from the faulty process. Note that this case includes the cryptographic machine $SM_c$.

*Wrapper state machines.* A *replacement* for a wrapper machine cannot fake correct MACs and so behaves equivalently to an unreliable channel: the failed machine can drop, reorder, corrupt and redirect the messages. The original algorithm is assumed to handle unreliable channels, hence also these faults, except malicious corruptions, which is handled by the MAC. *Removing* or *adding* machines are special cases of replacements. Altogether a byzantine fault in wrapper machines translates to ordinary channel faults or constant omission.

*Remote attestation state machines.* We assume that remote attestation is atomic, and do not consider faults *during* that attestation process. We proceed to consider the two cases of the remote attestation machine $SM_A$ experiencing a byzantine fault (1) before and (2) after the attestation completes.

(1) If the remote attestation machine suffers a byzantine fault *before* being remotely attested, it can drop, reorder, corrupt or redirect messages. However, it cannot create new valid messages in this process, nor can it read the shared secret that is provisioned to the cryptographic state machine as a result of the attestation. As such, a fault in the remote attestation machine can at worst have the consequence that either the cryptographic state machine is provisioned with a wrong secret, or not provisioned at all. Both cases translate to constant omission: any message sent from the integrity protected area will be dropped by any correct receiving process. (2) If the remote attestation state machine experiences a byzantine fault *after* the remote attestation process has completed, that fault is equivalent to a byzantine fault in a wrapper state machine, since the remote attestation state machine resides outside the integrity protected area, and does not have special privileges.

Clearly, an unhandled byzantine fault in $P_{RA}$ violates all of the former guarantees, as byzantine processes can now be provisioned with the shared secret. Therefore, the remote attestation process must be protected in a way that guarantees that only correct processes are provisioned with the shared secret. Such mechanisms exist [36]–[38].

Except for byzantine faults in the remote attestation process ($P_{RA}$ and $SM_{RA}$), we have seen that on all ordinary processes, the transformation translates byzantine faults to omissions, constant omissions, unreliable channels and crashes.

## V. Transformation example

We exemplify the transformation with the classic central-server mutual exclusion algorithm (CSME), see, e.g. [39]. In mutual exclusion, a collection of processes share access to one or more resources, referred to as the *critical section* (CS). To prevent data-races and race-conditions, a mutual exclusion algorithm ensures that at most a single correct process has access to the critical section at any given time, that is, at most one correct process may execute in the CS at a time (safety) and any requests to enter and exit the critical section eventually succeeds (liveness). As an aside, this example demonstrates that the translation has relevance outside consensus problems.

This algorithm does not provide liveness under process crashes: If the server crashes, no process can gain access tokens, so no requests for access succeed; if a client crashes with the token, the token is lost. However, safety is still guaranteed: no process can access the critical section without a token. Under byzantine faults, neither safety nor liveness is guaranteed. E.g., the server could distribute multiple tokens.

We now apply the transformation from Section IV to CSME, obtaining an algorithm which provides the same guarantees (safety) under byzantine faults as the untransformed does under crash faults. First, we model the algorithm as Mealy Machines, channels and processes. We model two different state machines: a client state machine, and a server state machine. The modelling is more straight-forward with pushdown automata, but we continue with Mealy Machines for clarity.

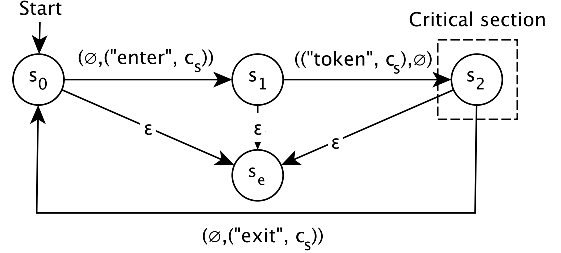



Fig. 3. Client state machine from the central server algorithm for mutual exclusion. Note that the channel $c_c$ represents the unreliable channel to the server, and is different across client instances.

The client state machine is modelled in Figure 3. It is in one of four states: not having requested access to the CS ($s_0$), waiting for the token from the server ($s_1$), being in the CS ($s_2$), or have crashed ($s_e$).

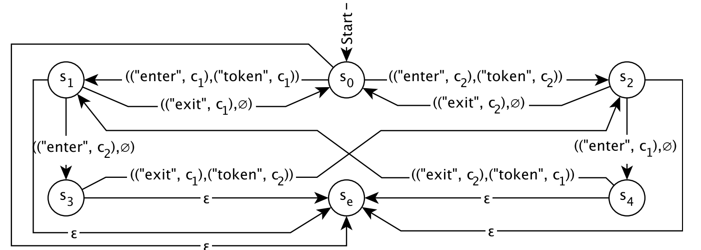



Fig. 4. Server state machine from the central server algorithm for mutual exclusion. Note that this server can only handle two client processes.

Figure 4 shows a model of the server state machine in an algorithm with two client processes. It encompasses 6 states:

$s_0$ no client has the token, no requests have been received
$s_1$ client 1 has requested the token
$s_2$ client 2 has requested the token
$s_3$ client 1 has the token, client 2 has requested the token
$s_4$ client 2 has the token, client 1 has requested the token
$s_e$ the state machine has crashed.

We will assume a CSME system with two client processes $P_1$ and $P_2$ each running an instance of the state machine from Figure 3, with an unreliable channel to the server process $P_0$, which runs an instance of the state machine from Figure 4.

We first show how the state machines are transformed to have their messages MAC'ed by $SM_c$ and to have $SM_c$ verify the messages they receive. This is done by applying the state machine transformation described in Section IV.

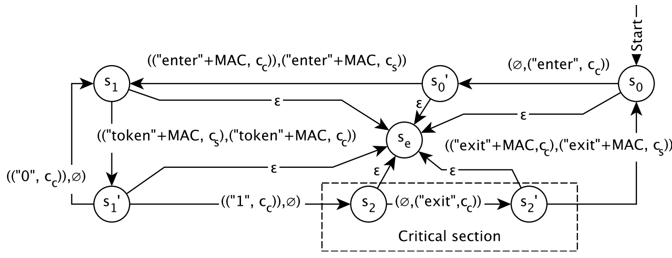



Fig. 5. Client state machine in CSME, after it has been transformed to handle byzantine faults.

Figure 5 shows the transformed client state machine. To enter the critical section, a request is sent to $SM_c$ over the inter-process channel $c_c$ for MAC'ing and transitions to $s_0'$. When the MAC'ed message is returned, it is sent to the server process. The state machine is now at $s_1$, which is equivalent to $s_1$ in the untransformed state machine. $c_s$ is now a channel to/from the wrapper state machine. When a token is received from $c_s$, the message's MAC is sent to $SM_c$ for verification. If the MAC is correct, the state machine will transition to $s_2$, which is in the critical section and is equivalent to $s_2$ in the untransformed state machine. A similar process is followed when exiting the critical section: the exit message is sent to $SM_c$ for MAC'ing, and the returned message is sent to the wrapper state machine for redirection to the server process. Figure 9 shows the transformed server state machine, omitting $s_e$ for readability. Figures 6 and 7 show the processes before and after the transformation, respectively.

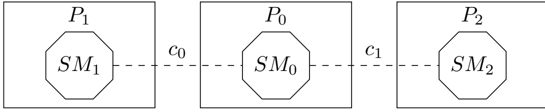



Fig. 6. Processes and channels in the central server algorithm for mutual exclusion, with one server and two clients. $SM_0$ is an instance of Figure 4, while $SM_1$ and $SM_2$ are instances of Figure 3.

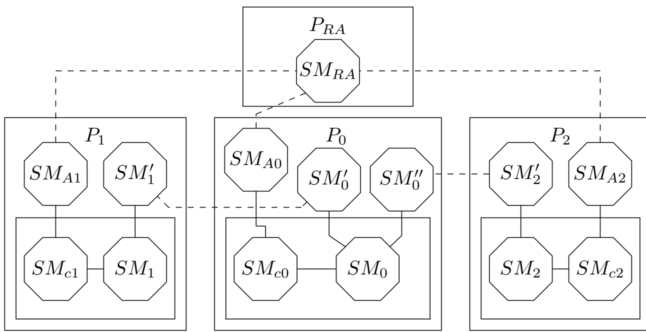



Fig. 7. Processes and channels in the central server algorithm for mutual exclusion, after they have been transformed to handle byzantine faults.

In the transformed CSME algorithm, correct processes exhibit the *same behaviour* under byzantine faults as they did under crash faults in the original algorithm. The example shows that the translation directly inherits fault resilience properties of the original algorithm: CSME has safety but not liveness under crash and omission faults, so the transformed CSME algorithm also has safety but not liveness, but under the stronger fault model of byzantine faults and omission faults.

Consider the fault where the server state machines try to serve tokens on any request, without waiting for the token to be released by the client holding it, violating safety. Model this byzantine fault as $SM_0$ has been exchanged with $SM_{byz}$ (see Figure 8), which serves a token to any request by the clients. Under this byzantine fault, none of the tokens can be MAC'ed. Thereby, any correct client process (Figure 5) will get stuck in a loop between $s_1$ and $s_1'$, when $SM_c$ rejects the token as it has not been MAC'ed. This is behaviour is equivalent to the server process crashing: a token will never be accepted, and the client will be unable to enter the CS.

While the transformation does seem quite complex in this example, we emphasize that the transformation can be fully automated by applying the 6 steps described in Section IV.

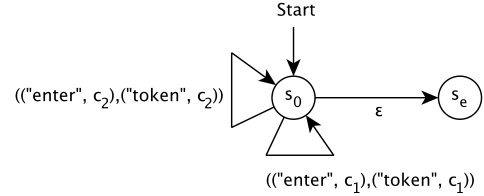



Fig. 8. The byzantine fault state machine $SM_{byz}$ on the server process.

## VI. Conclusion

We have given a 1-round, $n = f + 1$ transformation which translates byzantine faults to crashes and omission. The transformation relies on the integrity and confidentiality guarantees of a trusted execution environment and requires remote attestation. Using this model, we have shown how, by the properties of a TEE, the transformation translates the byzantine faults into either crashes, unreliable channels or omission faults. This ultimately makes an argument for the use of TEEs as trusted subsystems, and for the validity of using TEEs as the subsystem running other small trusted subsystems, by use of the presented transformation. Future work includes investigating performance of this technique when applied to more complex distributed algorithms; and implementing automated variants of this translation for protocol specification languages such as Session types, SDL or UML-variants.

## References

[1] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[2] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[3] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, no. 4, pp. 824–840, 1985.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, 1985.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested Append-only Memory: Making Adversaries Stick to Their Word," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. ACM, 2007, pp. 189–204.

[6] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "EBAWA: Efficient Byzantine agreement for wide-area networks," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium On*. IEEE, 2010, pp. 10–19.

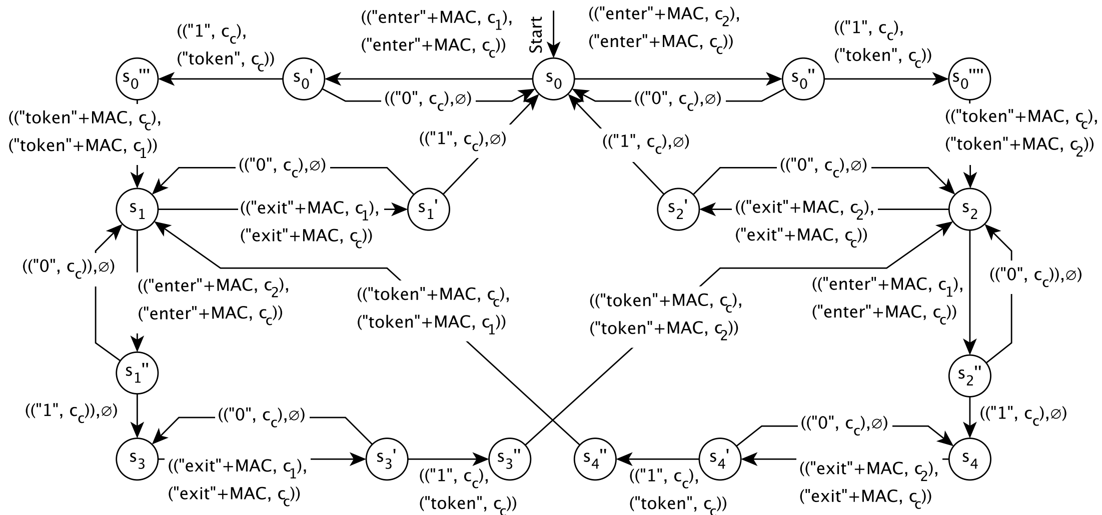



Fig. 9. Server state machine in the central server algorithm for mutual exclusion, after it has been transformed to handle byzantine faults. Note that $s_e$ has been omitted for improved readability.


[7] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 295–308.

[8] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[9] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable Byzantine Consensus via Hardware-assisted Secret Sharing," *preprint arXiv:1612.04997*, 2016.

[10] J. Behl, T. Distler, and R. Kapitza, "Hybrids on Steroids: SGX-Based High Performance BFT," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. ACM, 2017, pp. 222–237.

[11] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small Trusted Hardware for Large Distributed Systems." in *NSDI*, vol. 9, 2009, pp. 1–14.

[12] G. Neiger and S. Toueg, "Automatically increasing the fault-tolerance of distributed algorithms," *Journal of Algorithms*, vol. 11, no. 3, pp. 374–419, Sep. 1990.

[13] R. A. Bazzi and G. Neiger, "Simplifying Fault-Tolerance: Providing the Abstraction of Crash Failures," Georgia Institute of Technology, Technical Report, 1993.

[14] B. A. Coan, "A compiler that increases the fault tolerance of asynchronous protocols," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1541–1553, 1988.

[15] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues, "On the (limited) power of non-equivocation," in *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*. ACM Press, 2012, p. 301.

[16] Intel, "Hyperledger Sawtooth," Jun. 2019. [Online]. Available: https://sawtooth.hyperledger.org/

[17] iExec, "iExec Documentation," Jun. 2019. [Online]. Available: https://docs.iex.ec/

[18] D. Mpoeleng, P. Ezhilchelvan, and N. Speirs, "From crash tolerance to authenticated byzantine tolerance: A structured approach, the cost and benefits," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, Jun. 2003, pp. 227–236.

[19] G. Bracha, "Asynchronous Byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[20] C. Ho, D. Dolev, and R. van Renesse, "Making Distributed Applications Robust," *International Conference On Principles Of Distributed Systems*, vol. 4878, pp. 232–246, 2007.

[21] K. J. Perry and S. Toueg, "Distributed agreement in the presence of processor and communication faults," *IEEE Transactions on Software Engineering*, no. 3, pp. 477–482, 1986.

[22] GlobalPlatform, Inc., "Introduction to Trusted Execution Environments," May 2018.

[23] Intel, "Intel(R) Software Guard Extensions Programming Reference, Revision 2," October 2014.

[24] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64.

[25] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, vol. 13, 2013, p. 7.

[26] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," Intel Development Center, Israel, Tech. Rep. 204, 2016.

[27] GlobalPlatform, Inc., "GlobalPlatform Technology TEE System Architecture Version 1.2," Nov. 2018.

[28] C. Shepherd, R. N. Akram, and K. Markantonakis, "Establishing mutually trusted channels for remote sensing devices with trusted execution environments," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017, p. 7.

[29] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[30] Intel, "Intel(R) Software Guard Extensions Developer Reference for Linux OS."

[31] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security - EuroSec'17*. Belgrade, Serbia: ACM Press, 2017, pp. 1–6.

[32] F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*. USENIX Association, 2017, p. 12.

[33] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGX-PECTRE Attacks: Leaking Enclave Secrets via Speculative Execution," *arXiv preprint arXiv:1802.09085*, 2018.

[34] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in Darkness: Return-oriented Programming against Secure Enclaves," p. 19, 2017.

[35] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX," p. 16, 2018.

[36] F. Stumpf, O. Tafreschi, P. Roder, and C. Eckert, "A Robust Integrity Reporting Protocol for Remote Attestation," in *Future Wireless Networks and Information Systems*. Springer Berlin Heidelberg, 2006.

[37] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing - STC '08*. ACM Press, 2008.

[38] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, Jun. 2011.

[39] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Mar. 2004.

# Chapter 3

# On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems

# On the Subject of Non-Equivocation

## Defining Non-Equivocation in Synchronous Agreement Systems

Mads Frederik Madsen
IT University of Copenhagen
Copenhagen, Denmark
mfrm@itu.dk

Søren Debois
IT University of Copenhagen
Copenhagen, Denmark
debois@itu.dk

## ABSTRACT

We study *non-equivocation* in synchronous agreement protocols: the restriction on faulty processes that they cannot act differently towards distinct non-faulty processes. Guarantees of non-equivocation have been used to provide improved fault tolerance in agreement protocols, and various mechanisms for achieving it, have been proposed. However, the exact meaning of non-equivocation varies subtly in the literature. In this paper, we propose two different formal notions of non-equivocation: *strong* and *weak*. We define both as fault models for synchronous agreement protocols with reliable channels, and we show how the two models yield distinct bounds for the minimal number of communication rounds required and the maximum number of faulty processes tolerable to achieve agreement: 1 round, $n > t$ for strong non-equivocation; and $t + 1$ rounds, $n > 2t$ for weak non-equivocation. This makes weak non-equivocation the only fault model with a lower bound on fault tolerance of $n > 2t$ for broadcast agreement and interactive consistency, confirming the folklore knowledge that equivocation is, in a sense, the most critical of the Byzantine faults. Finally, we show how the weak and strong non-equivocation fault models relate to well-known agreement problems: strong non-equivocation corresponds to Byzantine broadcast and weak non-equivocation to crusader agreement.

## CCS CONCEPTS

• **Theory of computation → Distributed algorithms**.

## KEYWORDS

Agreement, non-equivocation, fault tolerance, synchronous systems

**ACM Reference Format:**
Mads Frederik Madsen and Søren Debois. 2020. On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems. In *Proceedings of The 39th ACM Symposium on Principles of Distributed Computing (PODC'20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*PODC'20, August 3–7 2020, Salerno, Italy,*
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Equivocation is the act of a faulty process *acting differently* when correct behaviour is to be consistent. In this paper, we study *non-equivocation* in synchronous agreement protocols: the guarantee that otherwise Byzantine processes will never equivocate.

The notion of equivocation was introduced informally and without the name by Lamport et al. [25]. Subsequently, Chun et al. [11] named it and showed that assuming non-equivocation (in one form) allows for an improvement in fault tolerance. Mechanisms providing non-equivocation can be constructed in different ways, with the most well-known being perhaps Attested Append-Only Memory (A2M) [11] and Trusted Incrementers (TrInc) [26].

Studies of non-equivocation have used subtly different definitions of the concept, leading to subtly different results. E.g.,[27] proposes the non-equivocation based "$f$-resilient" condition for agreement in synchronous systems, and conjecture informally that this property will be enough also in asynchronous ones. Meanwhile, [12] proves that transferable authentication is a necessary addition to non-equivocation for a fault tolerance improvement in asynchronous systems. The result of [12] does not contradict the conjecture in [27] as the two papers are referring to different notions of non-equivocation.

Thus, we propose two different concepts of non-equivocation: *strong* and *weak*. Both require faulty processes to not "lie differently", but weak non-equivocation allows faulty processes to selectively omit messages to some participants, where strong non-equivocation does not. In the above example [27] refers to a notion like that of strong non-equivocation, while [12] refers to one like that of weak non-equivocation.

Both A2M and TrInc can provide at best weak non-equivocation since a malicious or faulty process using a trusted module can still choose to simply not send messages. On the other hand, partial broadcast channels [13, 21, 22, 33], and local broadcast channels [23, 24] provide strong non-equivocation, as they guarantee broadcast messages are reliably and identically received by all neighbours.

Technically, we work in the original framework of Pease et al. [30], where we define both weak and strong non-equivocation as fault models for synchronous systems with reliable channels. We show for both how to solve the agreement problem of *interactive consistency* [30]. For strong non-equivocation, this is trivial; we obtain a 1-round, $n > t$ fault-tolerant algorithm simply by broadcasting a value, where $n$ is the number of processes, and $t$ is the number of faulty processes. For weak non-equivocation, we show how the problem is solvable using the algorithm of Pease et al. [30] with only minor adjustments. In this case, $t+1$ rounds are necessary, and we obtain exactly a fault-tolerance of $n > 2t$.

We relate the present results to the existing body of work on agreement protocols in Table 1 on page 3. To the best of our knowledge, weak non-equivocation is the only fault model with a lower bound of $n > 2t$ on the fault tolerance of broadcast agreement and interactive consistency in a synchronous model with reliable channels. This result demonstrates formally how equivocation is at the core of impossibility of Byzantine fault tolerance.

The notions of weak and strong non-equivocation as fault models fit neatly with our existing understanding of agreement problems in the following sense. Under guarantees of strong non-equivocation, it is trivial to implement Byzantine broadcast; conversely, given a Byzantine broadcast oracle, it is trivial to obtain strong non-equivocation. Similarly, under guarantees of weak non-equivocation, it is trivial to solve crusader agreement [15]; and conversely, weak non-equivocation is straightforward to obtain given a crusader agreement oracle. Essentially, properties of non-equivocation in synchronous models come down to properties of an oracle for the corresponding agreement problem.

The weak non-equivocation fault model resembles the authenticated Byzantine fault model in that both can be achieved by the use of cryptographic mechanisms. However, we show that while they do share a subspace of runs not in the omission fault model, both also contain a subspace of runs that the other fault model does not. So as fault models, they overlap, but only partly.

*Overview.* We proceed as follows. After presenting related work, we give in Section 2 a system model, in particular recalling the framework of Pease et al. [30]. We introduce strong non-equivocation in Section 3, followed by weak non-equivocation in Section 4. In Section 5 we give bounds on rounds and fault tolerance, and in Section 6 we prove those bounds tight. Finally, we discuss relations to other agreement problem variants in Section 7.

*Related work.* In *agreement protocols* processes attempt to agree on one or more values. Processes are given an input value, a *private value*, and must agree on one or more of these values, the *decision value(s)*. Table 1 shows optimal fault tolerance for agreement problems under various fault models, including results of the present paper. The notion of (authenticated) Byzantine agreement problems was introduced by Pease, Shostak, and Lamport [30][25], formulated as interactive consistency and the Byzantine generals problem. In interactive consistency, the non-faulty processes must agree on a value for each process, while in the Byzantine generals problem (i.e. Byzantine broadcast agreement) the non-faulty processes must agree on the private value of a pre-specified process. Interactive consistency can be achieved by running an instance of broadcast agreement per process [19]. Consensus is the problem formulation where every process suggests a value, and non-faulty processes decide on a single common value. It notably differs from interactive consistency and broadcast agreement in the validity property [3, 19]: if all non-faulty processes suggest the same value, then all non-faulty processes must decide that value. This stronger validity property makes the problem impossible to solve, unless $n > 2t$, when processes are allowed to lie about their value. Consensus can be achieved, if $n > 2t$, by running a majority function an interactive consistency vector, and similarly broadcast agreement can be achieved by running a consensus algorithm on the received values of a broadcast [19]. Uniform consensus differs from the other agreement problems in that also *faulty* processes must agree on the decided value [32, 34]. Fault-tolerant agreement is generally impossible under asynchrony (FLP-impossibility), as shown by Fischer, Lynch, and Paterson [20] because non-faulty processes cannot distinguish a faulty process from delayed messages. This impossibility is commonly circumvented with either partial synchrony [18], relaxation of termination or determinism [5, 7, 31] or the use of failure-detectors [9, 10].

*Non-equivocation* was suggested by Chun et al. [11] as a means of improving fault-tolerance, exemplified via improvements to PBFT by Castro and Liskov [8] for asynchronous Byzantine consensus with relaxed termination. The non-equivocation mechanism used was an attested append-only log. Levin et al. [26] showed how to construct such a log more efficiently using increment-only counters. Non-equivocation is strongly linked to broadcast primitives, and such two primitives have been used to achieve non-equivocation; partial broadcast channels [13, 21, 22, 33] and local broadcast channels [23, 24]. Here, non-equivocation means that messages sent through special channels are received reliably and identically by all neighbours. This corresponds to our notion of *strong non-equivocation* in Section 3 but limited to subsets of all processes. Other non-equivocation notions, e.g. that of [1, 4, 11, 12, 14, 27], are weaker system-wide properties, where faulty processes may omit messages. This corresponds to our notion of *weak non-equivocation* in Section 4. Algorithms that use non-equivocation to achieve agreement with improved fault-tolerance include [1, 14, 27].

Fitzi and Maurer [21] showed that synchronous Byzantine broadcasts (i.e. Byzantine generals) can be achieved with $n > 2t$ using a partial broadcast mechanism, which provides a property akin to strong non-equivocation, between only subsets of 3 processes. Chun et al. [11] showed how non-equivocation can be used to implement an $n > 2t$ resilient PBFT solution. Clement et al. [12] showed that neither non-equivocation nor transferable authentication is enough to give an $n > 2t$ resilient agreement protocol under asynchrony with relaxed termination; both are needed. Lastly, Jaffe et al. [22] examined the trade-off between equivocation and redundancy by showing under which conditions a fault-tolerance of $n = 2t + h$, for any $h > 0$, can be achieved using partial broadcast channels providing (strong) non-equivocation.

## 2 SYSTEM MODEL AND NOTATION

A system comprises $n$ synchronous processes connected with synchronous and reliable point-to-point channels. Up to $t$ processes may be faulty, and can deviate arbitrarily from the protocol, but not break the guarantees awarded by the fault model. To simplify the presentation, we separate the message-exchange protocol from the rest of the algorithm. By slight abuse of language, we will refer to the message-exchange as *the protocol*, and the calculation made by a process on its received messages as *the algorithm*.

We define $P = \{p, q, r, \ldots\}$ to be the set of all processes, $N$ to be the subset of all non-faulty processes, and $F$ to be the subset of faulty processes: $N \cup F = P$, $N \cap F = \emptyset$, $|N| = n$, and $|F| = t$. As usual, $P^*$ denotes all finite strings over $P$, $P^+$ denotes all non-empty finite strings, and we refer to an element of $P^*$ as a *process-sequence*.

**Table 1: Optimal fault tolerance of different agreement problems in different fault models, assuming synchronous processes and reliable, synchronous channels. Rows titled in bold represent results of the present paper.**

| Fault model | Broadcast agreement | Interactive consistency | Consensus | Uniform consensus |
|---|---|---|---|---|
| Byzantine | $n > 3t$ [25]† | $n > 3t$ [30] | $n > 3t$ [25]‡ | N/A |
| **Weak non-equivocation** | $n > 2t$ | $n > 2t$ | $n > 2t$ | N/A |
| Authenticated Byzantine | $n > t$ [25] | $n > t$ [30] | $n > 2t$ [35] | N/A |
| Omission | $n > t$ [34] | $n > t$ [34] | $n > t$ [34] | $n > 2t$ [32] |
| Crash | $n > t$ [28] | $n > t$ [28] | $n > t$ [28] | $n > t$ [34] |
| **Strong non-equivocation** | $n > t$ | $n > t$ | $n > 2t$ | N/A |

† Aka Byzantine generals

‡ Aka Byzantine agreement

$n$ processes, $t$ faulty

Each message sent comprises of a value $v \in V$, and a member of $P^+$, representing the route the value has been sent along. Each process can correctly determine the immediate sender of a message. Note that lying about the process-sequence of a message is equivalent to lying about the value [30], so we consider only lies about values for the remainder of this paper.

We will define non-equivocation as a fault model for the protocol. A fault model limits which messages faulty processes may send. For this purpose, we use the framework of [30], where the message-exchange is represented as a map $\sigma : P^+ \to V$ from process-sequences to values. For instance, $\sigma(p|q) = v$ means that $q$ has sent $v$ to $p$, ("|" denotes string concatenation); and $\sigma(p|q|r) = v$ that $q$ has reported to $p$ that $r$ has reported the initial value $v$ to $q$. Processes do not send messages to themselves, so, abusing notation, we consider only process-sequences which do not contain consecutive duplicate characters, e.g., we disregard $p|p|q$.

A process $p$ obviously has access only to the messages it has received. Formally, we define $\sigma_p$ as $\forall w \in P^* \colon \sigma_p(w) = \sigma(p|w)$. For ease of notation, and since process-sequences cannot include consecutive duplicates, we let $\sigma_p(p|w) = \sigma_p(w)$. In cases of message omissions, we use $\bot$ to denote the absence of a value in $\sigma$. We use $\langle\bot\rangle$ as the value specifying the absence of a value, that is, if $\sigma(p|r) = \bot$ then a correct $p$ will report $\langle\bot\rangle$ to any $q$: $\sigma(q|p|r) = \langle\bot\rangle$.

## 2.1 Interactive consistency

Recall the *interactive consistency problem* [30]. Each process $p$ starts with a private value $v_p$ and must compute a vector, $vec_p$, of size $n$ such that:

**Agreement:** All non-faulty processes compute the same vector: $\forall p, q \in N \colon vec_p = vec_q$

**Validity:** If $q$ is non-faulty, all non-faulty processes sets the $q$'th element in the vector to $q$'s private value: $\forall p, q \in N \colon vec_p[q] = v_q$

**Termination:** All non-faulty processes must eventually compute a vector.

Interactive consistency is known to be reducible to Byzantine broadcast and vice versa [19], so to solve interactive consistency, it is sufficient to give an algorithm describing Byzantine broadcast. We exploit this equivalence to interchangeably solve one or the other, depending on which is more convenient: We generally find it more convenient to solve Byzantine broadcast, however, interactive consistency supports a more intuitive induction proof in Section 5.

For Sections 3 to 5, we use the following protocol, which is a specialized round-based message-passing protocol. All processes go through $t+1$ *information exchange broadcast rounds.* In the first broadcast round, the non-faulty processes broadcast their private values, and subsequently, in round $r$ they broadcast the values they received in round $r-1$, along with the route that value has taken so far. As we consider only a finite number of rounds and synchronous channels and processes, *termination* of the protocol is trivial, and we need only consider the termination of the algorithm.

This protocol gives us the following guarantee: a non-faulty process neither lies about its private value, nor its previously received messages.

Definition 2.1 (Non-faulty process guarantee). *A non-faulty process truthfully broadcast values, reporting $\langle\bot\rangle$ only when they have observed a message omission:*

$$\forall q \in N : \forall p \in P \setminus \{q\} : \forall w \in P^* :$$

$$\sigma(p|q|w) = \begin{cases} \langle\bot\rangle, & \text{if } \sigma(q|w) = \bot \\ \sigma(q|w), & \text{otherwise} \end{cases}$$

In the non-faulty process guarantee, $\langle\bot\rangle$ can be interpreted as the reporting process reporting that there exists a suffix of the process-sequence of that message, let's call it $w'$, for which $\sigma(w') = \bot$. Note that $\langle\bot\rangle$ is simply a value, and *can* be reported as any process' private value. Also, note that this interpretation is not a guarantee; a faulty process can report $\langle\bot\rangle$ having received neither $\bot$ nor $\langle\bot\rangle$ in the previous round. *Faulty* processes are allowed to deviate arbitrarily from the protocol and so from Definition 2.1; they are restricted only by the fault model.

# 3 STRONG NON-EQUIVOCATION

We define strong non-equivocation fault model as a weakening of the Byzantine fault model:

Definition 3.1 (Strong non-equivocation). *For any given process-sequence, all processes—faulty or not—report the same value to all other processes:*

$$\forall p, r, r' \in P \colon \forall w \in P^* \colon \sigma(r|p|w) = \sigma(r'|p|w)$$

*Since $w$ can be the empty string and is prefixed with the same $p$ on both sides of the equality, this is equivalent*[1] *to*

$$\forall r, r' \in P \colon \forall w \in P^+ \colon \sigma(r|w) = \sigma(r'|w)$$

Under strong non-equivocation, a faulty process may still lie: a faulty process $q$ may receive $v$ from $p$ in round 1, but proceed to claim to other processes that $p$ said $w$ in round 2. Similarly, a faulty process may omit messages under the strong non-equivocation fault model, but only if it omits all messages in that round.

The strong non-equivocation fault model overlaps with the crash fault model: a faulty process may fail by not sending any messages from a given round onwards, thereby acting as if crashed. However, the strong non-equivocation fault model does not contain the crash fault model: in the crash fault model a faulty process can fail during its broadcast, leading to a situation where some processes have received a message from the crashing process, while others have not; a violation of Definition 3.1. Conversely, a faulty process can lie in a given round in the strong non-equivocation fault model, which is not allowed in either the general omission- or the crash fault model. So the strong non-equivocation fault model must overlap with a strict subspace of the crash fault model, but also extend into a strict subspace of the Byzantine fault model, see Figure 4 on page 9 for an illustration of the fault model spaces.

Given a guarantee of strong non-equivocation, it is trivial to solve *Byzantine broadcast*, and so also interactive consistency. For source $q$, each correct $p$ simply decides on the value it was sent by $q$ in round 1: $vec_p[q] = \sigma_p(p|q)$. Clearly, if $q$ is correct, each correct process sets the same value. If $q$ is faulty, then by Definition 3.1, each other process has received the same message from $q$, and so all decide on the same value. (We give a formal proof of correctness in Appendix A.) Note that the strong non-equivocation fault model really is very strong: all is done in one round, and within that single round, the source process *cannot fail* in a way that makes it in any way distinguishable from a correct process, making interactive consistency and Byzantine broadcast solvable for $n > t$. Note that for the problem of consensus, the lower bound is $n > 2t$, as processes can lie about their private value (see Section 1).

We argue that Byzantine broadcast and strong non-equivocation are interchangeable in the sense that strong non-equivocation implies a trivial solution to Byzantine broadcast as shown above. Similarly, a Byzantine broadcast oracle can be used to make algorithms assuming the strong non-equivocation fault model work in the Byzantine fault model. Informally, the latter holds because, after Byzantine broadcast, all non-faulty processes agree on some value for the source process, which is exactly strong non-equivocation.

Strong non-equivocation is most useful in systems with partial trust, or given primitives providing the properties to parts of the system. For examples, the extensive work on partial broadcast channels, e.g. [13, 21, 22, 33], use a definition of non-equivocation exactly like strong non-equivocation; every process on the partial broadcast channel receive the same messages. Using this primitive, Fitzi and Maurer [21] consider the problem of achieving global broadcast, i.e. Byzantine broadcast, from such strongly non-equivocating partial broadcast channels. Given that strong non-equivocation is interchangeable with Byzantine broadcast, the problem of achieving global broadcast from partial broadcast channels can be rephrased as a problem of *extending* strong non-equivocation across systems; achieving system-wide strong non-equivocation from strongly non-equivocating partial broadcast channels.

[1] We are implicitly understanding this requirement to be only when $r'|w$ contains no consecutive duplicates, see Section 2.

# 4 WEAK NON-EQUIVOCATION

Key practical approaches to non-equivocation revolve around using cryptographic primitives in trusted modules, possibly hardware, making equivocation readily detectable by non-faulty processes. This is the approach taken by TrInc and A2M [11, 26]. In this case, strong non-equivocation is not an appropriate fault model, since the possibly faulty process employing the trusted module may still omit any message it chooses, including omitting messages to some but not all other processes within a single round. Thus, we define *weak non-equivocation*, where all processes still must report the same value to other processes for each process-sequence, but are allowed to omit messages to some subset of processes. Recall absence of a message is represented as $\perp$ in $\sigma$.

Definition 4.1 (Weak non-equivocation). *For any given process-sequence, all processes—faulty or not—report the same value to a subset of processes, and omit messages to the rest:*

$$\forall p, r, r' \in P \colon \forall w \in P^* \colon \sigma(r|p|w) \neq \perp \wedge \sigma(r'|p|w) \neq \perp \implies \sigma(r|p|w) = \sigma(r'|p|w)$$

*Since $w$ can be the empty string and is prefixed with the same $p$ on both sides of the equality, this is equivalent to:*

$$\forall r, r' \in P \colon \forall w \in P^+ \colon \sigma(r|w) \neq \perp \wedge \sigma(r'|w) \neq \perp \implies \sigma(r|w) = \sigma(r'|w)$$

Observe that the omission- and crash fault models are strict subspaces of the weak non-equivocation fault model as a faulty process can arbitrarily omit any message, and consequently also stop sending messages altogether from a given round onwards, thereby acting as if crashed See Figure 4 on page 9 for an illustration of the fault model spaces.

## 4.1 Algorithm for $t = 1$

To establish intuition about the general algorithm, we give in Algorithm 1 on page 5 a solution to Byzantine broadcast under weak non-equivocation when $t = 1$. (For the general case, see Section 5.) Note that for $n = 3$ this is the traditional Byzantine generals problem, which is *not* solvable in the Byzantine fault model [25].

The algorithm proceeds as follows over two rounds. If a non-$\perp$ value is received in the first round (line 3), we decide on that value (line 4). Otherwise, the source must be faulty and (as $t = 1$) all other processes non-faulty. If any other non-faulty process received a value from the source, they correctly report that in the second round (line 5), and we decide on that value (line 6). Otherwise, no non-faulty process can have received a value in the first round, and so we decide on $\langle \perp \rangle$ as the value. The non-faulty processes all set the same value because they consider only the value sent by the source in the first round, and by weak non-equivocation (Definition 2.1), all who received the a value in round 1 received the same value. Altogether, we have correctness (for full proof, refer to the generalisation in Theorem 5.1):

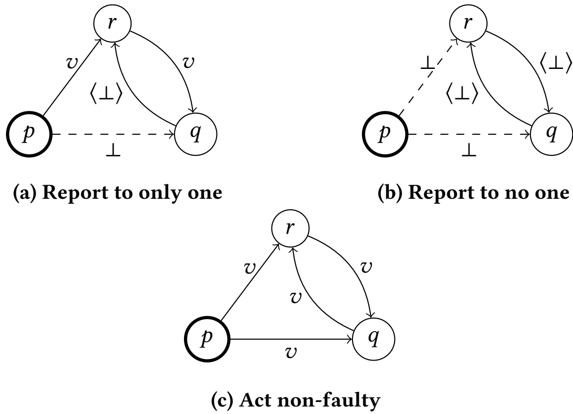



**Figure 1: Three examples of $n = 3, t = 1$ executions, showing all possible actions a faulty source, $p$, can take under weak non-equivocation, parametric on the value of $v$.**

Theorem 4.2. *In the weak non-equivocation fault model, interactive consistency is solvable for any $n > 1$ if $t \leq 1$.*

It is instructive to consider the different possibilities a faulty process has for lying under weak non-equivocation in the case $n = 3, t = 1$. Refer to Figures 1 and 2. In each case, it is immediately apparent to non-faulty nodes either who is faulty, or what the correct value should be. E.g., in Figure 1a, $r$ knows to decide on $v$ upon receiving it; $q$ knows that $p$ is faulty from the omission, thus deciding upon the value from $r$, which must be correct because $t = 1$. In Figure 2b, $r$ knows that $q$ is faulty as $p$ could not have reported $v'$ to $q$ without violating weak non-equivocation (Definition 4.1).

## 4.2 Properties of weak non-equivocation

To reason about the general case, we introduce the formal device of *consistent majorities*. Consider a process $p$ trying to decide on the value of a source process $q$. A consistent $q$-majority for a value $v$ is a set $Q$ of more than $t$ processes, including $q$, who have consistently reported the value $v$ to $p$, both for themselves and for each other.

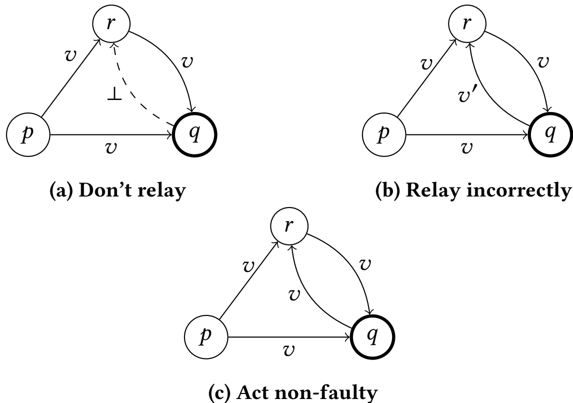



**Figure 2: Three examples of $n = 3, t = 1$ executions, showing all possible actions a faulty non-source, $q$, can take under weak non-equivocation, parametric on the values of $v$ and $v'$ and assuming that $v \neq v'$.**

**Algorithm 1** Algorithm for process p to set q'th value in vector

**function** $\text{WNE}_{t=1}(\sigma_p, p, q)$
1: **if** $p = q$ **then** ▷ $p$ is $q$
2: return $\sigma_p(p)$
3: **else if** $\sigma_p(p|q) \neq \bot$ **then** ▷ $q$ reported a value to $p$
4: return $\sigma_p(p|q)$
5: **else if** $\exists r. \sigma_p(p|r|q) \neq \langle\bot\rangle$ **then** ▷ Someone else told $p$ about $q$
6: return $\sigma_p(p|r|q)$
7: **else** ▷ $q$ have reported $\bot$ to everyone
8: return $\langle\bot\rangle$
9: **end if**
**end function**

Definition 4.3 (consistent majority). *We write $\mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P)$ and say that $p$ has a consistent $q$-majority, when*

$$\forall w \in Q^* : |w| \leq t : \sigma_p(p|w|q) = v \land |Q| \geq t + 1 \land Q \subseteq P$$

Note that $p$ having a consistent $q$-majority does not imply that any other process has a consistent $q$-majority, as $p$ may have received different messages.

A consistent majority is related to the notion of a *quorum* but differs by, (1) the source process must be part of a consistent majority, and (2) the processes in a consistent majority $Q$ also agree that everybody in $Q$ agree on the same value, at least as far as $p$ knows. The principle behind consistent majorities was described in [30].

Just from the message exchange $\sigma$, we can derive helpful relationships between the distribution of faulty processes and the existence of consistent majorities, *provided* we may assume $n > 2t$. Recall that $P = N \cup F$ is the set of $n$ processes, where $F$ is the set of at most $t$ faulty processes, and $N$ is the set of non-faulty processes. Let $p, q$ and $r$ be processes and let $\sigma$ be a map of messages after at least $t + 1$ information exchange rounds according to Definition 2.1 and in the weak non-equivocation fault model (Definition 4.1).

Lemma 4.4. *If $p$ is non-faulty, then all non-faulty processes will have a consistent majority for $p$'s private value $v_p$:*

$$\forall q \in N : \exists Q \subseteq P : p \in N \implies \mathsf{maj}^{\mathsf{p,v_p}}_{\mathsf{q,Q}}(t, \sigma_q, P)$$

Proof. By Definition 2.1, all non-faulty processes correctly forward values each round. So $p$ must have reported $v_p$ to the set of non-faulty processes $N$ in the first round, and all processes in $N$ must have reported $v_p$ to each other in all following rounds. Then $N$ must be a consistent majority for $v_p$, noting $|N| > t$. □

Lemma 4.5. *If a process $p$ does not have a consistent $q$-majority then $q$ must be faulty:*

$$\forall p, q \in P : (\forall v \in V : \forall Q \subseteq P : \neg\mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P)) \implies q \in F$$

Proof. Contraposition of Lemma 4.4. □

Lemma 4.6. *If $p$ has a consistent $q$-majority for $v$, then it cannot also have a consistent $q$-majority for a distinct value $v'$:*

$$\forall v, v' \in V : \forall p, q \in P : \exists Q, R \subseteq P :$$
$$\mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P) \wedge \mathsf{maj}^{\mathsf{q,v'}}_{\mathsf{p,R}}(t, \sigma_p, P) \implies v = v'$$

Proof. By contradiction. Assume that for some process $p$, $\mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P)$, $\mathsf{maj}^{\mathsf{q,v'}}_{\mathsf{p,R}}(t, \sigma_p, P)$ and $v \neq v'$ holds. By definition of consistent majority (Definition 4.3), any set $Q$ s.t. $\mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P)$ must include $q$, so it must be the case that $\sigma(p|q) = v$. But then by $\mathsf{maj}^{\mathsf{q,v'}}_{\mathsf{p,R}}(t, \sigma_p, P)$ it must be the case that $\sigma(p|q) = v'$, which contradicts $v \neq v'$. □

Lemma 4.7. *No consistent majority for $\perp$ can exist:*

$$\forall p, q \in P : \forall v \in V : (\exists Q \subseteq P : \mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P)) \implies v \neq \perp$$

Proof. By contradiction. Assume that there exists a $Q$ s.t. $\mathsf{maj}^{\mathsf{q,\perp}}_{\mathsf{p,Q}}(t, \sigma_p, P)$. By Definition 4.3, $|Q| > |F|$, so there must exist a non-faulty process in $Q$, i.e. a non-faulty process that has reported $\perp$. But, by Definition 2.1, non-faulty processes never report $\perp$. □

Lemma 4.8. *If non-faulty processes $p$ and $q$ each have a consistent $r$-majority, then those must be for the same value:*

$$\forall p, q \in N : \forall v, v' \in V :$$
$$(\exists Q, R \subseteq P : \mathsf{maj}^{\mathsf{r,v}}_{\mathsf{p,Q}}(t, \sigma_p, P) \wedge \mathsf{maj}^{\mathsf{r,v'}}_{\mathsf{q,R}}(t, \sigma_q, P)) \implies v = v'$$

Proof. *By contradiction.* Assume that $p$ and $q$ have consistent $r$-majorities for respectively different values $v$ and $v'$. By Definition 4.3, $p$ having an consistent $r$-majority for $v$ implies that $\sigma(p|r) = v$, and $q$ having an consistent $r$-majority for $v'$ implies $\sigma(q|r) = v'$. By weak non-equivocation (Definition 4.1), $\sigma(p|r) \neq \sigma(q|r)$ implies that either $v$ or $v'$ is $\perp$. But by Lemma 4.7, no consistent majority for $\perp$ can exist: a contradiction. □

## 5 SOLVING INTERACTIVE CONSISTENCY UNDER WEAK NON-EQUIVOCATION

To solve interactive consistency, it is sufficient to define an algorithm that achieves agreement, validity and termination for one source process, and then run that algorithm for each process in the system [19, 30]. Assuming $n > 2t$ and weak non-equivocation, we give such an algorithm in Algorithm 2, which is a generalisation of Algorithm 1 and an adaptation of the algorithm by Pease, Shostak, and Lamport [30], which assumes the Byzantine fault model. The key differences are the addition of $\perp$ and $\langle\perp\rangle$, the required size of consistent majorities in Definition 4.3, and the size of the set in line 20, which is adapted to fit with a fault tolerance of $n > 2t$.

Algorithm 2 relies on two insights. (1) If a process cannot find a consistent majority, then the source process must be faulty (Lemma 4.5). (2) After $t + 1$ rounds, if the source process is faulty, we can treat the other processes reports about the source process as their private values in a new instance of interactive consistency; write $\widehat{\sigma}$ for the restriction of $\sigma$ where we remove any mention of the source. This $\widehat{\sigma}$ will have $t$ message rounds regarding these values. Since the *faulty* source process is excluded, $\widehat{\sigma}$ can be used to recursively find the values reported to the rest of the processes,

**Algorithm 2** Algorithm for process $p$ to set source process $q$'s value in an interactive consistency vector

**function** PSL($\sigma_p, n, t, p, q, P$)
1: **if** $p = q$ **then** ▷ $p$'s own private value
2: **if** $\sigma_p(p) = \perp$ **then** ▷ $\perp \notin V$, so choose default $\langle\perp\rangle$
3: return $\langle\perp\rangle$
4: **else**
5: return $\sigma_p(p)$
6: **end if**
7: **end if**
8: **if** $\exists Q \subseteq P : \mathsf{maj}^{\mathsf{q,v}}_{\mathsf{p,Q}}(t, \sigma_p, P)$ **then** ▷ Consistent majority for $v$
9: return $v$
10: **end if** ▷ No consistent majority; $q$ is faulty (Lemma 4.5)
11: $\widehat{\sigma_p} \leftarrow \{\}$ ▷ Start constructing $\widehat{\sigma_p}$
12: $\widehat{P} \leftarrow P \setminus \{q\}$ ▷ Exclude $q$ from $\widehat{\sigma_p}$
13: **for all** $w \in \widehat{P}^*$, s.t. $p|w|q$ is in the domain of $\sigma_p$ **do**
14: $\widehat{\sigma_p}(p|w) \leftarrow \sigma_p(p|w|q)$
15: **end for** ▷ Done constructing $\widehat{\sigma_p}$
16: $\widehat{\mathrm{vec}} \leftarrow [\,]$ ▷ Constructing $\widehat{\mathrm{vec}}$: IC vector of $\widehat{\sigma_p}$
17: **for all** $r \in \widehat{P}$ **do**
18: $\widehat{\mathrm{vec}}[r] \leftarrow \mathrm{PSL}(\widehat{\sigma_p}, n-1, t-1, p, r, \widehat{P})$
19: **end for** ▷ Done constructing $\widehat{\mathrm{vec}}$
20: **if** $\exists v \neq \langle\perp\rangle : |\{r \mid \widehat{\mathrm{vec}}[r] = v\}| \geq t$ **then**
21: return $v$ ▷ Min. one non-faulty process received $v$
22: **else**
23: return $\langle\perp\rangle$
24: **end if**
**end function**

by either finding a consistent majority for them or recursively applying the same method. In the base case, there is only one faulty process left, in which case Algorithm 2 behaves like Algorithm 1.

We proceed to prove correctness of Algorithm 2.

Theorem 5.1. *Algorithm 2 is correct.*

Proof sketch (full proof in Appendix C). **Validity**. Assume a non-faulty source $p$; we must show that for any non-faulty process, Algorithm 2 returns the private value $v_p$. Because $p$ is non-faulty, all non-faulty processes have a consistent $p$-majority for $v_p$ (Lemma 4.4), and cannot have a consistent $p$-majority for another value (Lemma 4.6). Thus Algorithm 2 will return $v_p$ in line 9. **Agreement**. We must show that for all non-faulty processes, Algorithm 2 returns the same value. Consider a source $p$. If $p$ is non-faulty, agreement follows trivially from validity (line 9), so assume $p$ faulty. We proceed by induction on $t$. For the base case, $t = 1$, $p$ either reports a value to some processes, in which case all others know to trust a forwarded value (line 21), or not report at value at all, in which case everybody agrees that this is the case (line 23), and choose a default value. For the induction case, consider two arbitrary non-faulty processes, and proceed by cases on which have a consistent $p$-majority. If they both do, then it is for the same value by Lemma 4.8. If neither has a consistent $p$-majority, then both know that $p$ is faulty, and run Algorithm 2 recursively with $p$, obtaining a correct interactive consistency

vector by IH. They both run a majority function on this vector (line 20), arriving at the same result. If one has a consistent majority, and the other does not, then the latter must have a recursive vector with a majority of the same value as that of the consistent majority by Definition 4.3 and IH (see also Lemma C.1 in Appendix C). **Termination**. Immediate from the observation that message exchange is bounded at $t + 1$; the domain of $\sigma$ is then finite; the domain of $\sigma_p$ is then also finite; and it is now obvious that all loops in Algorithm 2 terminates. □

Theorem 5.1 shows that interactive consistency can be solved for $n > 2t$ under weak non-equivocation. However, the solution is not efficient, requiring $\Theta(n^{t+1})$ messages to construct the necessary $\sigma$. Algorithms that solve interactive consistency in a polynomial number of messages, e.g. [16] exists for the Byzantine fault model, and we leave a polynomial algorithm for the weak non-equivocation fault model as future work.

## 6 TIGHTNESS

Having proved interactive consistency possible for $n > 2t$ under weak non-equivocation, we now show that it is a tight lower bound.

Theorem 6.1. *Given any finite number of rounds, k, and $n \leq 2t$, no synchronous deterministic algorithm exists which, given a $\sigma$ in the weak non-equivocation fault model, always finds an interactive consistency vector with the properties of agreement, validity and termination.*

Proof. By contradiction. Assume a deterministic algorithm, $A$, where any process $p$ can run $A(p, \sigma_p, q)$ and obtain a value for $q$ fulfilling the requirements of agreement and validity. Without loss of generality assume that $n = 2t$ and that each run continues for $k$ rounds, where $k \geq t + 1$. Partition all processes into sets $O, Q, R$ and $S$, s.t. $|O| = \lfloor \frac{t}{2} \rfloor$, $|Q| = \lceil \frac{t}{2} \rceil$, $|R| = \lceil \frac{t}{2} \rceil$ and $|S| = \lfloor \frac{t}{2} \rfloor$. Note that all processes in any two partitions can be simultaneously faulty, except for $Q$ and $R$ as their combined size may be greater than $t$.

Let the source process be $p \in O$. We construct four runs with different faulty partitions, such that some non-faulty process will be unable to distinguish two specific runs from each other, leading to a contradiction if the algorithm achieves both validity and agreement. Let $v_1$ and $v_2$ be *distinct* values not equal to $\perp$ or $\langle\perp\rangle$. For ease of notation let $A(Q, \sigma_{Run1}, p)$ denote the value each process in $Q$ will decide for $p$ in run 1. We construct runs such that $A(Q, \sigma_{Run1}, p) = v_1$ by validity, and $A(Q, \sigma_{Run1}, p) = A(Q, \sigma_{Run3}, p)$ by determinism of $A$. Similarly we will construct the runs such that $A(R, \sigma_{Run2}, p) = v_2$ by validity, and $A(R, \sigma_{Run2}, p) = A(R, \sigma_{Run4}, p)$ by determinism of $A$. Lastly we will construct the runs such that $A(S, \sigma_{Run3}, p) = A(S, \sigma_{Run4}, p)$ by determinism of $A$, ultimately showing that $S$ must decide both $v_1$ and $v_2$ which is impossible as $v_1 \neq v_2$.

We construct the runs as follows. Let the processes in $O$ broadcast to the other partitions in round 2 exactly as $p$ broadcast in round 1.

**Run 1** (The processes of $R$ and $S$ are faulty):
- $p$ broadcasts the value $v_1$ in the first round.
- In the second round, the processes in $R$ broadcast the value $v_2$ and the processes in $S$ broadcast $\langle\perp\rangle$.
- In the third round the processes in $R$ broadcast the value $v_2$ and the processes in $S$ broadcast $\langle\perp\rangle$, as the respective values received from $O$.
- All processes correctly report all other messages.

By validity, all processes in $Q$ decide $v_1$.

**Run 2** (The processes of $Q$ and $S$ are faulty):
- $p$ broadcasts the value $v_2$ in the first round.
- In the second round, the processes in $Q$ broadcast the value $v_1$ and the processes in $S$ broadcast $\langle\perp\rangle$.
- In the third round the processes in $Q$ broadcast the value $v_1$ and the processes in $S$ broadcast $\langle\perp\rangle$, as the respective values received from $O$.
- All processes correctly report all other messages.

By validity, all processes in $R$ decide $v_2$.

**Run 3** (The processes of $O$ and $R$ are faulty):
- In the first round $p$ broadcasts the value $v_1$ to all except for the processes in partition $S$, resulting in $S$ recording $\perp$.
- In the second round the processes in $R$ broadcast $v_2$, and $O$ broadcasts $v_1$ to all but $S$, resulting in $S$ recording $\perp$.
- In the third round the processes in $R$ broadcast the value $v_2$ as the values received from $O$.
- All processes correctly report all other messages.

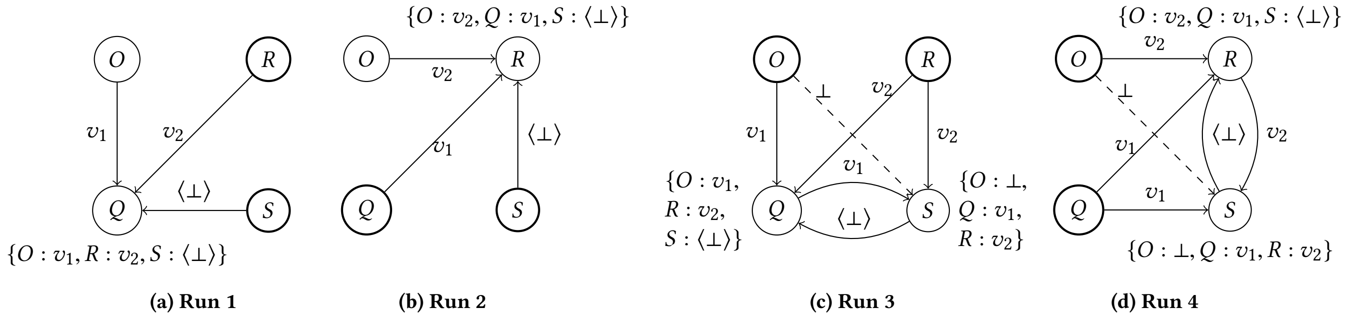
For the processes in $Q$, this run is indistinguishable from run 1: in both runs, they have received $v_1$ from $O$, $v_2$ from $R$, and $\langle\perp\rangle$ from $S$ by the end of round 2, and all partitions will act the same for the remaining rounds. By determinism, $Q$ should decide $v_1$ to be the value for $p$. By agreement, $S$ must then decide $v_1$ to be the value for $p$.

**Run 4** (The processes of $O$ and $Q$ are faulty):
- In the first round $p$ broadcasts the value $v_2$ to all except for the processes in partition $S$, resulting in $S$ recording $\perp$.
- In the second round processes in $Q$ broadcast $v_1$, and $O$ broadcasts $v_2$ to all but $S$, resulting in $S$ recording $\perp$.
- In the third round the processes in $Q$ broadcast the value $v_1$ as the values received from $O$.
- All processes correctly report all other messages.

For the processes in $R$, this run is indistinguishable from run 2: in both runs, they have received $v_2$ from $O$, $v_1$ from $Q$, and $\langle\perp\rangle$ from $S$ by the end of round 2, and all partitions will act the same for the remaining rounds. By determinism, $R$ should decide $v_2$ to be the value for $p$. By agreement, $S$ must then decide $v_2$ to be the value for $p$. But for the processes in $S$, this run is indistinguishable from run 3: in both runs, they have received nothing ($\perp$) from any process in $O$, $v_1$ from all processes in $Q$, and $v_2$ from all processes in $R$ by the end of round 2, and all partitions will act the same in the two runs for the remaining rounds. So, by determinism all processes in $S$ should decide $v_1$ to be the value for $p$, leading to the contradiction that the processes in $S$ should decide both $v_1$ and $v_2$ for distinct $v_1$ and $v_2$. □

Note that this proof uses determinism only to restrict that non-faulty processes decides the same value under indistinguishable runs. This restriction may be implied by the validity and agreement properties, making the proof have no requirement of determinism. We leave a more thorough investigation of this as future work.

The weak non-equivocation fault model is, to the best of our knowledge, the only fault model with this lower bound fault-tolerance for interactive consistency and broadcast in the synchronous model; the Byzantine fault model has $n > 3t$ and all




**(a) Run 1** **(b) Run 2** **(c) Run 3** **(d) Run 4**

**Figure 3: Runs described in the proof of Theorem 6.1. Faulty partitions are bold. Arrows are selected messages sent in round 2. The source process $p$ is in $O$, so the arrows from $O$ are messages in both round 1 and 2. For selected partitions, the messages received by end of round 2 are annotated. Note how partition $Q$ receives the same messages in run 1 and 3, partition $R$ receives the same messages in run 2 and 4, and partition $S$ receives the same messages in run 3 and 4.**

others has $n > t$ (see Table 1). This uniqueness confirms formally the folklore knowledge that equivocation is the most critical of the Byzantine fault and what necessitates the triple replication of processes in Byzantine agreement problems.

Considering the fault tolerance of general omission algorithms ($n > t$), Theorem 6.1 also suggests that the type of faults that separates general omission from weak non-equivocation, i.e. lying without equivocating, is what necessitates anything but the least amount of replication. This hypothesis is also backed up by the fault tolerance of the authenticated Byzantine fault model ($n > t$), where faulty processes are prevented from lying *at all* about (but not omitting) the messages of non-faulty processes.

A proof sketch of the tightness of the bound of $t + 1$ rounds is simply showing that weak non-equivocation encompasses the crash fault model. As it is known that the broadcast agreement cannot be solved for less than $t + 1$ rounds in the crash fault model [2], it follows that the problem requires at least as many rounds in weak non-equivocation. A formal proof can be found in Appendix B.

## 7 OTHER MODELS

In this section, we relate weak non-equivocation to (1) the problem of crusader agreement as defined by Dolev [15], and (2) the authenticated Byzantine fault model as defined by Pease et al. [30].

### 7.1 Weak non-equivocation and Crusader Agreement

*From weak non-equivocation to crusader agreement.* Weak non-equivocation (Definition 4.1) implies a trivial solution to the problem of *crusader agreement* [15]. Crusader agreement is an agreement problem weaker than Byzantine broadcast: non-faulty receiving processes must decide a value in agreement with all other non-faulty processes *unless* the receiving process *knows* that the sending process is faulty. Formally, processes in the crusader agreement problem decides a value for the process $p$'s private value $v_p$, with these properties:

**Crusader agreement:** All non-faulty processes decide the same value unless they know that $p$ is faulty.[2]

**Validity:** If process $p$ is non-faulty, then all non-faulty processes decide $v_p$.

**Termination:** All non-faulty processes must eventually decide on a value, or know that $p$ is faulty.

These properties are *exactly true* for any message in the weak non-equivocation fault model: for some message, a non-faulty process either receives a value from the sending process or it receives nothing; if the receiving process receives nothing, then it knows that the sending process is faulty; if the process receives a value, then it is guaranteed that every other non-faulty process that also received a value, have received the same value. Thus, a procedure where the non-faulty processes (1) decides on the received value if they receive one, and (2) otherwise sets the source process a faulty, solves crusader agreement under weak non-equivocation.

*From Crusader Agreement to Weak non-equivocation.* Suppose there exists an oracle producing crusader agreement upon request, and that we use this oracle to produce crusader agreement for the value of some source process $p$. After finishing deciding such a value, all processes will (knowingly) be either faulty; non-faulty and having decided a common value; or non-faulty and knowing that $p$ is faulty. This is exactly the same three states a process will find itself in after $p$ has broadcast a value under weak non-equivocation: for some $v$, all correct processes $q$ will have either $\sigma_q(p) = v$ or $\sigma_q(p) = \perp$; in the latter case, $q$ knows for a fact that $p$ is faulty.

Dolev [15] showed that crusader agreement has a fault tolerance of $n > 3t$ in the pure Byzantine fault model, but that it only requires a constant of 2 information exchange rounds. Dolev's crusader agreement algorithm works by using a purifying function much like Definition 4.3[3] to agree on a value or know that the source is faulty. Assuming that $n > 3t$, we can use Dolev's crusader agreement

[2]"know that $p$ is faulty" is a slightly imprecise definition. For more precision, the phrase "can prove that if the receiving process itself is non-faulty, then $p$ must be faulty" can be used.

[3]The most notable difference between Definition 4.3 and Dolevs purifying function is that the purifying function does not require the source process to be part of the consistent majority, and, of course, that the consistent majority must be of size at least $2t + 1$, due to the lower fault tolerance.

algorithm as a primitive to map any $\sigma$ with $t+2$ rounds in the Byzantine fault model into a $\sigma$ with $t+1$ rounds in the weak non-equivocation fault model. The mapping proceeds by using the extra (last) round as the second round in Dolev's algorithm, thus achieving crusader agreement for the values transmitted in round $t+1$. These values can then be used to achieve crusader agreement for the values transmitted in round $t$, and so on.

Altogether, it seems any algorithm that operates correctly given a crusader agreement primitive, can replace that primitive with an assumption of weak non-equivocation. Examples of algorithms using such a primitive can be found in [6, 17, 29].

## 7.2 Comparison to the authenticated Byzantine fault model

In this section, we examine the weak non-equivocation fault model's relationship with the authenticated Byzantine fault model. We refer to the authenticated Byzantine fault model as defined in Pease et al. [30], which says that a faulty process cannot lie (but can *omit* messages) about the values of non-faulty processes:

$$\forall p \in N \colon \forall w', w \in P^* \colon \sigma(w'|p|w) = \sigma(p|w) \vee \bot \ .$$

Given that interactive consistency problem is solvable in the authenticated Byzantine fault model for $n > t$, and that the same problem is solvable for weak non-equivocation for a tight $n > 2t$, one would expect the authenticated fault model is the stronger fault model. However, we shall see that whereas the authenticated fault model and weak non-equivocation obviously have an intersection, they also have a non-empty set difference in either direction. As Clement et al. [12] showed in the asynchronous setting both authentication and non-equivocation is needed to achieve a fault tolerance of $n > 2t$ for agreement problems, this relationship between the fault models is not entirely surprising. See Figure 4 for a Venn diagram of spaces of $\sigma$'s that exist in different fault models.

We demonstrate that (1) there exists $\sigma$'s where algorithms for weak non-equivocation can solve agreement, but algorithms for the authenticated Byzantine fault model cannot; (2) there exists $\sigma$'s where algorithms for the authenticated Byzantine fault model can solve agreement, but algorithms for weak non-equivocation cannot; and (3) there exists $\sigma$'s where algorithms for either fault model can solve agreement, but algorithms for general omission cannot. Examples of each follow here:

(1) For any such $\sigma$, no process can equivocate, but some process must lie about the value of a non-faulty process. Let $n = 3, t = 1, p, r \in N$ and $q \in F$. Let $r$ be the source process, and broadcast $v$ in the first round. Then let $q$ proceed to report that they received $u$ in the next round. Any algorithm for authenticated Byzantine faults will wrongly conclude that $r$ is faulty, as only fault processes can have their value spoofed, while any algorithm for weak non-equivocation will correctly conclude $q$ is faulty, as $r$ has not equivocated.
(2) For any such $\sigma$, no process can lie about the value of a non-faulty process but some process must equivocate. Let $n = 3, t = 1, p, q \in N$ and $r \in F$. Let $r$ be the source process, and send a $v$ to $p$, and a $u$ to $q$ in the first round, and let $p$ and $q$ proceed to report their received values. Any algorithm for authenticated Byzantine faults will correctly conclude that $r$ is faulty, as different values can only exists for faulty processes, while any algorithm for weak non-equivocation will wrongly conclude the other non-faulty process is, in fact, faulty, as it "believes" that $r$ is unable to equivocate.
(3) For any such $\sigma$, the following three properties must hold: (I) Some process must lie, so that general omission algorithms cannot solve agreement. (II) No process can equivocate, so that weak non-equivocation algorithms can solve agreement. (III) No faulty process can lie about the values of non-faulty processes so that authenticated Byzantine algorithms can solve agreement. From (I), (II) and (III), we can deduce that $t \geq 2$, and, by the lower bound fault tolerance of weak non-equivocation, that $n \geq 5$: Let $n = 5, t = 2, p, q, r \in N$ and $s, x \in F$. Let $s$ be the source process, and send $v$ to $p$ in the first round and nothing, $\bot$, to $q$ and $r$. Let $x$ act as if it received $u$, but otherwise, let every process report correctly for the proceeding rounds[4]. Both weak non-equivocation and authenticated Byzantine algorithms will deduce that $s$ is faulty, and that there is no majority for any specific value, and so decide some default value. Meanwhile, for a general omission algorithm, $q$ and $r$ know that $s$ is faulty, but have received the contradicting values $v$ and $u$, and will thus behave in an unspecified manner.

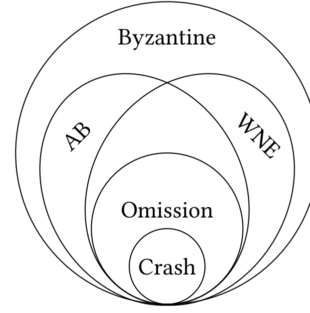



**Figure 4: The relationship between spaces of $\sigma$s described by various fault models. Abbreviations: AB – authenticated Byzantine, WNE – weak non-equivocation. Not pictured is strong non-equivocation, which overlaps with but is not contained in the crash fault model, yet contained in the weak non-equivocation fault model.**

## 8 FUTURE WORK

Our results on fault-tolerance together with that of [12] suggests that the property of *transferable authentication* alone can bridge the gap between an agreement algorithm with non-equivocation in a synchronous system and an asynchronous system with weakened termination. An examination of this result, and in which other system models this translation is valid, could lead to more efficient translations between synchronous and asynchronous algorithms.

Given that the strong non-equivocation fault model is overlapping with the crash fault model, and that it implies a trivial

[4]For the sake of argument, let $r$ and $q$ receive $u$ from $x$ before they receive $v$ from $p$, to prevent early stopping general omission algorithms from deciding $v$ before receiving the contradicting $u$ value

solution to Byzantine agreement as seen in Section 3, it seems that FLP-impossibility may not apply in systems using a primitive providing strong non-equivocation. This opens up questions of whether any weakening of termination, determinism or asynchrony is strictly needed in asynchronous agreement systems that use strong non-equivocation primitives such as partial broadcast channels, e.g. [13, 21, 22, 33].

## 9 CONCLUSION

In this paper, we addressed and formalised the notion of *non-equivocation* in synchronous agreement protocols. We have proposed two different notions of non-equivocation: *strong* and *weak*. Both require faulty processes to not "lie differently", however, weak non-equivocation allows faulty processes to selectively omit messages to some participants, where strong non-equivocation does not. We defined both formally as fault models for synchronous agreement protocols with reliable channels, and we showed how the two models yield distinct round- and fault tolerance-bounds for agreement: 1 round, $n > t$ for strong non-equivocation; and $t + 1$ rounds, $n > 2t$ for weak non-equivocation. This makes weak non-equivocation the only fault model with a lower bound on fault tolerance of $n > 2t$ for broadcast agreement and interactive consistency, thus confirming formally the folklore knowledge that equivocation is somehow the most critical of the Byzantine faults. Finally, we have shown how the weak and strong non-equivocation fault models relate to well-known agreement problems: strong non-equivocation corresponds to Byzantine broadcast and weak non-equivocation to crusader agreement.

## REFERENCES

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. 2017. Efficient Synchronous Byzantine Consensus. *arXiv:1704.02397 [cs]* (Sept. 2017). arXiv:cs/1704.02397 http://arxiv.org/abs/1704.02397

[2] Marcos Kawazoe Aguilera and Sam Toueg. 1999. A Simple Bivalency Proof That T-Resilient Consensus Requires T+1 Rounds. *Inform. Process. Lett.* 71, 3 (1999), 155–158.

[3] Chagit Attiya, Danny Dolev, and Joseph Gil. 1984. Asynchronous Byzantine Consensus. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (PODC '84)*. Association for Computing Machinery, New York, NY, USA.

[4] Michael Backes, Fabian Bendun, Ashish Choudhury, and Aniket Kate. 2014. Asynchronous MPC with a Strict Honest Majority Using Non-Equivocation. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. Association for Computing Machinery, Paris, France, 10–19.

[5] Michael Ben-Or. 1983. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC '83)*. Association for Computing Machinery, Montreal, Quebec, Canada, 27–30.

[6] Malte Borcherding. 1997. Byzantine Agreement with Limited Authentication. In *Safe Comp 96*, Erwin Schoitsch (Ed.). Springer, London, 404–413.

[7] Gabriel Bracha and Sam Toueg. 1985. Asynchronous Consensus and Broadcast Protocols. *J. ACM* 32, 4 (Oct. 1985), 824–840.

[8] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, New Orleans, Louisiana, USA, 173–186.

[9] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The Weakest Failure Detector for Solving Consensus. *J. ACM* 43, 4 (July 1996), 685–722.

[10] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267.

[11] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. Association for Computing Machinery, Stevenson, Washington, USA, 189–204.

[12] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. 2012. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC '12)*. Association for Computing Machinery, Madeira, Portugal, 301–308.

[13] Jeffrey Considine, Matthias Fitzi, Matthew Franklin, Leonid A. Levin, Ueli Maurer, and David Metcalf. 2005. Byzantine Agreement Given Partial Broadcast. *J Cryptology* 18, 3 (July 2005), 191–217.

[14] Miguel Correia, Giuliana S. Veronese, and Lau Cheuk Lung. 2010. Asynchronous Byzantine Consensus with 2f+1 Processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. Association for Computing Machinery, Sierre, Switzerland, 475–480.

[15] Danny Dolev. 1981. *The Byzantine Generals Strike Again.* Technical Report. Stanford University, Stanford, CA, USA.

[16] Danny Dolev, Michael J. Fischer, Rob Fowler, Nancy A. Lynch, and H. Raymond Strong. 1982. An Efficient Algorithm for Byzantine Agreement without Authentication. *Information and Control* 52, 3 (March 1982), 257–274.

[17] Danny Dolev and Avi Wigderson. 1983. On the Security of Multi-Party Protocols in Distributed Systems. In *Advances in Cryptology*. Springer US, Boston, MA.

[18] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323.

[19] Michael J. Fischer. 1983. The Consensus Problem in Unreliable Distributed Systems (a Brief Survey). In *Foundations of Computation Theory (Lecture Notes in Computer Science)*, Marek Karpinski (Ed.). Springer Berlin Heidelberg, 127–140.

[20] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

[21] Mattias Fitzi and Ueli Maurer. 2000. From Partial Consistency to Global Broadcast. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC '00)*. Association for Computing Machinery, Portland, Oregon, USA, 494–503.

[22] Alexander Jaffe, Thomas Moscibroda, and Siddhartha Sen. 2012. On the Price of Equivocation in Byzantine Agreement. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC '12)*. Association for Computing Machinery, Madeira, Portugal, 309–318.

[23] Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. 2019. Exact Byzantine Consensus on Undirected Graphs under Local Broadcast Model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. Association for Computing Machinery, Toronto ON, Canada, 327–336.

[24] Muhammad Samir Khan and Nitin H. Vaidya. 2019. Byzantine Consensus under Local Broadcast Model: Tight Sufficient Condition. *arXiv:1901.03804 [cs]* (Jan. 2019). arXiv:cs/1901.03804 http://arxiv.org/abs/1901.03804

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401.

[26] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Boston, Massachusetts, 1–14.

[27] Chuanyou Li, Michel Hurfin, Yun Wang, and Lei Yu. 2016. Towards a Restrained Use of Non-Equivocation for Achieving Iterative Approximate Byzantine Consensus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, 710–719.

[28] Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[29] Stephen R. Mahaney and Fred B. Schneider. 1985. Inexact Agreement: Accuracy, Precision, and Graceful Degradation. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing (PODC '85)*. Association for Computing Machinery, Minaki, Ontario, Canada, 237–249.

[30] M. Pease, R. Shostak, and L. Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (April 1980), 228–234.

[31] Michael O. Rabin. 1983. Randomized Byzantine Generals. In *24th Annual Symposium on Foundations of Computer Science (SFCS'83)*. 403–409.

[32] Philippe Raipin Parvedy and Michel Raynal. 2003. Uniform Agreement Despite Process Omission Failures. In *2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*. IEEE.

[33] D. V. S. Ravikant, V. Muthuramakrishnan, V. Srikanth, K. Srinathan, and C. Pandu Rangan. 2004. On Byzantine Agreement over (2,3)-Uniform Hypergraphs. In *18th International Symposium on Distributed Computing (DISC'18)*, Rachid Guerraoui (Ed.). Springer, Berlin, Heidelberg, 450–464.

[34] Michel Raynal. 2002. Consensus in Synchronous Systems: A Concise Guided Tour. In *2002 Pacific Rim International Symposium on Dependable Computing, 2002 (PRDC'02)*. 221–228.

[35] Michel Raynal. 2010. *Fault-Tolerant Agreement in Synchronous Message-Passing Systems.* Morgan & Claypool.

# Appendices

## A STRONG NON-EQUIVOCATION CORRECTNESS PROOF

**Algorithm 3** Algorithm for process $p$ to set $q$'th value in vector, under the strong non-equivocation fault model

```
    function SNE(σ_p, p, q)
1:      if p = q then
2:          return σ_p(p)
3:      else
4:          return σ_p(p|q)
5:      end if
    end function
```

Lemma A.1. *Algorithm 3 guarantees validity for $n > t$, in the strong non-equivocation fault model.*

Proof. We must prove that for any non-faulty process, $p$, running Algorithm 3 for some other non-faulty process $q$, Algorithm 3 will produce $q$'s private value $v_q$.
Assume $n > t + 1$, since validity is trivially ensured for $n = t + 1$.

Let $p$ and $q$ be non-faulty processes. By the Definition 2.1 we know that $\sigma(p|q) = \sigma(q)$, for all non-faulty $q$.

□

Lemma A.2. *Algorithm 3 guarantees agreement for $n > t$, in the strong non-equivocation fault model.*

Proof. We must prove that for any two non-faulty processes, $p$ and $r$, running Algorithm 3 for some third process $q$, Algorithm 3 will produce the same result for $p$ and $r$.

Proof by contradiction. Let $p$ and $r$ be non-faulty processes. Assume that $p$ and $r$ get different results from Algorithm 3, for some other process $q$. Then $\sigma(p|q) \neq \sigma(r|q)$, but this contradicts Definition 3.1. □

Lemma A.3. *Algorithm 3 guarantees termination for any terminating protocol.*

Proof. The proof follows trivially from the fact that the protocol terminates, which implies that a lookup in $\sigma$ terminates. □

## B TIGHTNESS OF ROUNDS FOR WEAK NON-EQUIVOCATION

We will now show that $t + 1$ rounds is the least amount of rounds needed to solve interactive consistency in the weak non-equivocation fault model. We do this by arguing that the weak non-equivocation fault model completely encompasses the crash fault model (i.e. any $\sigma$ under the crash fault model can exist under the weak non-equivocation fault model). Since it is well-known, e.g. from [2], that agreement problems in a crash fault model cannot be solved with less than $t + 1$ information exchange rounds, it must follow that agreement problems in weak non-equivocation must use at least as many. And with the proof of Theorem 5.1, we know that interactive consistency is solvable using $t + 1$ information exchange rounds, showing that $t + 1$ rounds is a tight lower bound.

That weak non-equivocation encompass crash faults follow trivially from the fact that faulty processes can arbitrarily not report anything and thus can act exactly as a crashed process by not reporting anything from some time and until the protocol has terminated. A formal proof follows:

Theorem B.1. *Any $\sigma$ that can exist under the crash-failure model must also be possible under weak non-equivocation.*

Proof. By contradiction. Assume that there exists a $\sigma$ in the crash-failure model that cannot exist under weak non-equivocation, and denote such a $\sigma$ as $\sigma_{crash}$. In $\sigma_{crash}$ there must exist two processes $p, q$, a (possibly empty) string $w$ and value (or $\perp$) $v$ such that $\sigma_{crash}(q|p|w) = v$, and no $\sigma$ restricted under weak non-equivocation can produce the same $v$ given the input $q|p|w$:

$$\forall \sigma_{WNE} \in WNE \colon \sigma_{WNE}(q|p|w) \neq v$$

where $WNE$ is the space of possible $\sigma$'s restricted under weak non-equivocation.

Now we construct a contradiction by cases on the faultiness of $p$:

**Case 1:** ($p$ is non-faulty): as $p$ is non-faulty, $v$ must be a correct and valid value determined by $w$: $\sigma_{crash}(p|w) = v$, which must be possible in all $\sigma$'s restricted under weak non-equivocation according to the non-faulty process guarantee. As such $p$ cannot be non-faulty.
**Case 2:** ($p$ is faulty): as $p$ is faulty, it can act non-faulty which is identical to Case 1, which we have shown cannot be the case. Alternatively, $p$ can act faulty, which in the crash-failure model means crashed. If $p$ is crashed then $v = \perp$. But according to weak non-equivocation, is always allowed to report $\perp$, meaning any such $\sigma$ must be possible under the restriction of weak non-equivocation. As such $p$ cannot be faulty.

Leading to the contradiction that $p$ is neither faulty nor non-faulty. □

So we can conclude that the number of information exchange rounds of Algorithm 2 is optimal. This should not be mistaken as the algorithm being efficient or optimal in the number of messages; we conjecture that a message-efficient (polynomial) algorithm such as the one presented by Dolev et al. [16] could be modified to allow for optimal fault tolerance in a weak non-equivocation fault model, but leave this as future work.

## C PROOFS OF CORRECTNESS FOR THE GENERAL WEAK NON-EQUIVOCATION ALGORITHM

The proof relies on the following technical Lemma.

Lemma C.1. *If a process $p$ has a consistent $q$-majority for the value $v$, then $p$'s $\widehat{\mathsf{vec}}_p$ must contain at least $t$ $v$-values.*

$$\forall p, q \in P \colon \forall v \in V \colon (\exists Q \subseteq P \colon \mathsf{maj}^{q,v}_{p,Q}(t, \sigma_p, P)) \implies |\widehat{\mathsf{vec}}_p[\cdot] = v| \geq t$$

Proof. By Definition 4.3, each process in $Q$ have consistently reported $v$ wrt. all other processes in $Q$ including $q$, to $p$:

$$\forall w \in Q^* \colon |w| \leq t \colon \sigma_p(p|w|q) = v$$

So $\widehat{\sigma_p}$, the map used to create $\widehat{\text{vec}}_p$, must have the property that each process in $Q \setminus \{q\}$ have consistently reported $v$ about itself and all other processes in $Q \setminus \{q\}$, to $p$:

$$\forall w' \in (Q \setminus \{q\})^* : |w'| \leq t-1 : \widehat{\sigma_p}(p|w') = v$$

So, for each process in $Q \setminus \{q\}$, $p$ must be able to find a consistent majority for $v$ in the recursive call that makes up $\widehat{\text{vec}}$:

$$\forall r \in Q \setminus \{q\} : \mathsf{maj}^{\mathsf{r,v}}_{\mathsf{p,(Q\setminus\{q\})}}(t-1, \widehat{\sigma_p}, P \setminus \{q\})$$

And since $|Q| \geq t+1$ by Definition 4.3, then $(Q \setminus \{q\}) \geq t$, giving us the result that $\widehat{\text{vec}}_p$ must contain at least $t$ $v$-values. ☐

We can now proceed to prove Algorithm 2 correct, by proving that it satisfies validity, agreement and termination:

Lemma C.2. *Algorithm 2 satisfies validity when $n > 2t$.*

Proof. Assume a non-faulty source $p$; we must show that for any non-faulty process, Algorithm 2 returns the private value $v_p$. Because $p$ is non-faulty, all non-faulty processes have a consistent $p$-majority for $v_p$ (Lemma 4.4), and cannot have a consistent $p$-majority for another value (Lemma 4.6). Thus Algorithm 2 will return $v_p$ in line 9. ☐

Lemma C.3. *Algorithm 2 satisfies agreement when $n > 2t$.*

Proof. We must prove that every non-faulty process will get the same result by running Algorithm 2 for some source-process $q$. By cases on the faultiness of $q$:

**Case 1:** ($q$ is non-faulty) Agreement follows from validity if $q$ is non-faulty.

**Case 2:** ($q$ is faulty) By induction on $t$:

**Base step**: $t = 1$. By cases on the $q$'s first messages:

**Case 2.i:** First consider the case $\forall p \in P \setminus \{q\} : \sigma(p|q) = \bot$, i.e. where the source process does not send a single message in the first round. By Lemma 4.7, none of the non-faulty processes will have a consistent majority. Thus all processes will calculate $\widehat{\text{vec}}$, excluding the source process and make a recursive call with $t = 0$. By Definition 2.1 and since $t = 0$ in the recursive call, all other processes must have sent $\langle\bot\rangle$ in round 2. Thus all values in $\widehat{\text{vec}}$ must be $\langle\bot\rangle$, making Algorithm 2 return $\langle\bot\rangle$ for all non-faulty processes on line 23.

**Case 2.ii:** Next consider the case $\exists p \in P \setminus \{q\} : \sigma(p|q) \neq \bot$ i.e. where the source process sends a value $v$ ($\neq \bot$) to one or more processes in the first round. For any process, $p$, receiving $v$ in the first round, Algorithm 2 will return $v$ on line 9 as $\{p, q\}$ is a consistent $q$-majority. All other processes must have received $\bot$ in the first round by weak non-equivocation (Definition 4.1), and thus, by Lemma 4.7, calculate $\widehat{\text{vec}}$, excluding $q$ and with $t = 0$. By the non-faulty process guarantee (Definition 2.1), and Lemma 4.4, $\widehat{\text{vec}}$ consist only of $\langle\bot\rangle$ and a $v$ value for each process that received $v$ in the first round, which is at least 1. As $\widehat{\text{vec}}$ contains only $\langle\bot\rangle$ and $v$ Algorithm 2 will, for all non-faulty processes that did not receive $v$ in the first round, return $v$ in line 21 since $1 \geq t = 1$.

**Induction step**: Induction hypothesis: Algorithm 2 satisfies agreement for $t-1$ for any $n > 2(t-1)$. We must prove that, given the induction hypothesis, Algorithm 2 satisfies agreement for $t$. Recall that by assumption $n > 2t$. Consider any two non-faulty processes, $q$ and $r$, and a faulty source process $p$. The proof is on cases of existence of consistent $p$-majorities for $q$ and $r$.

**Case 2.iii:** Consider the case $\exists v, v' \in V : \exists (Q, R) \subseteq P : \mathsf{maj}^{\mathsf{p,v}}_{\mathsf{q,Q}}(t, \sigma_q, P) \wedge \mathsf{maj}^{\mathsf{p,v'}}_{\mathsf{r,R}}(t, \sigma_r, P)$ i.e. where $q$ has a consistent $p$-majority for some value $v$, and $r$ has a consistent $p$-majority for some value $v'$. By Lemma 4.8, we know that $v = v'$.

**Case 2.iv:** Consider the case $\forall v, v' \in V : \forall Q \subseteq P : \neg\mathsf{maj}^{\mathsf{p,v}}_{\mathsf{q,Q}}(t, \sigma_q, P) \wedge \neg\mathsf{maj}^{\mathsf{p,v'}}_{\mathsf{r,Q}}(t, \sigma_r, P)$ i.e. where neither $q$ nor $r$ have a consistent $p$-majority. Both $q$ and $r$ will calculate $\widehat{\text{vec}}$ by calling Algorithm 2 with $n-1$ and $t-1$ and each process other than $p$ as the source process. By assumption $n > 2t$, so $n-1 > 2(t-1)$. As such the induction hypothesis applies, and so $q$'s $\widehat{\text{vec}}$ and $r$'s $\widehat{\text{vec}}$ must be equal.

**Case 2.v:** Consider the case $(\exists v \in V : \exists Q \subseteq P : \mathsf{maj}^{\mathsf{p,v}}_{\mathsf{q,Q}}(t, \sigma_Q, P)) \wedge (\forall v' \in V : \forall R \subseteq P : \neg\mathsf{maj}^{\mathsf{p,w}}_{\mathsf{r,R}}(t, \sigma_R, P))$ i.e. where $q$ has a consistent $p$-majority for some value $v$, and $r$ does not have a consistent $p$-majority for any value. Let $q$ calculate $\widehat{\text{vec}}$ using $\sigma_q$, *despite* having a consistent $p$-majority, and denote the result $\widehat{\text{vec}}_q$. By the induction hypothesis $r$ will calculate the same $\widehat{\text{vec}}$ as $q$: $\widehat{\text{vec}}_r = \widehat{\text{vec}}_q$. By Lemma C.1 we know that there is at least $t$ $v$-values in $\widehat{\text{vec}}_q$ and therefore also in $\widehat{\text{vec}}_r$. As there is $t$ $v$-values in $\widehat{\text{vec}}_t$ Algorithm 2 returns $v$ on line 21 for $r$, while returning $v$ for $q$ in line 9, as $q$ has a consistent $p$-majority for $v$.

Note that in the case where $n = 2t + 1$, $\widehat{\text{vec}}_r$ will contain exactly $2t$ values, leaving room in $\widehat{\text{vec}}_r$ for a value different from $v$ with $t$ instances. However, such a value can only be $\langle\bot\rangle$ by the following argument. If there are two different values with $t$ instances in $\widehat{\text{vec}}_r$, then $q$'s consistent $p$-majority for value $v$ must be of size exactly $t+1$ by consequence of Lemma C.1. As $r$ does not have a consistent $p$-majority, $q$'s consistent majority must contain at least 1 faulty process by Lemma 4.5. This implies that there is at least one non-faulty process, $s$, that is not part of $q$'s consistent majority. Note that $s$ and $r$ may be the same process. According to weak non-equivocation (Definition 4.1) $s$ must have received either $v$ or $\bot$ in the first round. And by validity (Lemma C.2), $\widehat{\text{vec}}_r[s]$ must then be either $v$ or $\langle\bot\rangle$, ensuring that if $\widehat{\text{vec}}_r$ $t$ instances of a value other than $v$, that value must be $\langle\bot\rangle$.

☐

Lemma C.4. *Algorithm 2 satisfies termination for any terminating protocol.*

Proof. Termination follows from the following facts: (1) when $t = 0$ there is always a consistent majority (Lemma 4.4); (2) all loops are finite, including the loop over all $w \in \widehat{P^*}$ s.t. $p|w|q$ is a key in $\sigma$ (line 13), as there is a finite amount of keys in $\sigma$; and (3) $n$ and $t$ are finite, and so the number of recursion-calls to PSL($\widehat{\sigma_p}, n-1, t-1, p, r, \widehat{P}$) (line 18) must be finite. ☐

Proof of Theorem 5.1. Immediate from Lemmas C.2 (validity), C.3 (agreement), and C.4 (termination). ☐

Chapter 4

# Impalpable Differences: Secret Actions in Processes and Concurrent Workflows

# Impalpable Differences: Secret Actions in Processes and Concurrent Workflows

Mads Frederik Madsen and Søren Debois

IT University of Copenhagen, Copenhagen, Denmark
{mfrm,debois}@itu.dk

**Abstract.** We study *secrecy of actions* in concurrent systems where multiple actors collaborate on executing a process, and not all actors are privy to all parts of that process. This problem is particularly relevant in the field of business process modelling, where the situation that organisations engage in 3-way collaborations, but two parties wish the details of their sub-collaboration to be hidden from the third, is quite common. To this end, we present a novel theory of *indistinguishability of actions*, generally applicable to LTS-based or run-based process semantics, and use it to derive concrete results about indistinguishability of workflow sub-models in the declarative DCR process notation.

**Keywords:** Action secrecy, concurrent workflows, indistinguishability

## 1 Introduction

We consider secrecy of observable actions in systems where multiple agents work in concert to execute a process. We are especially interested in this concept in the setting of formal models of collaborating business processes, where the problem becomes harder because agents (businesses and organisations) must necessarily observe some of each other's actions. If an agent wants to keep some action secret from some other agent, it must ask itself: Can I be sure that my adversary cannot, through observation and inference, determine whether I took the action?

While this work is in principle situated in information flow theory [14,33,11], we draw heavily on work on secrecy in process models [10,18,26] and in formal models of business processes [24,2]. We are motived by applications within the latter, in particular collaborative business process execution. Experience from attempts to apply tenets of Secure Multiparty Computation (MPC) (see e.g. [13]) led to the realisation that even under ideal secrecy conditions, some actions are impossible to keep secret due to the process model itself, and the necessary information leakage through required communication between parties.

As an example, consider a collaboration between a supplier of goods and a shipping agent. The shipping agent may pick up a parcel from a warehouse speculatively but must wait for the supplier to confirm receipt of payment before undertaking actual delivery. Thus, in this process, the supplier must observe a

"confirm payment" action at the supplier to decide availability of the "delivery" action. Moreover, if the recipient of the parcel (who may not be the buyer) knows of the underlying process, he can deduce from receiving the parcel that "confirm payment" happened, even though he cannot observe it directly.

Formally, we model this situation in labelled transition systems (LTSes), where actions become labels, and each agent is responsible for some subset of actions and decides if and when to take those actions. The actions of agents are not independent: the availability of actions for one agent may depend on actions previously taken by others. Thus agents must be informed of such previous executions, at least enough to decide which actions are currently available.

We model this communication via *observations* on runs. An adversary has access to certain observations for a run, and the question is whether the adversary can, from those, determine if the secret actions have happened or not. Formally, our theory is parametric in a notion of observability ($\mathcal{N}$), akin to the ones of [31,32]. It is in this setup we define runs of the system as being indistinguishable to an adversarial agent, from which we derive notions of secrecy. It is presumed here that the adversary knows the structure of expected interactions, modelled by the adversary knowing the set of all possible runs.

Altogether, our contributions are: (1) we give a general definition of indistinguishability in LTSes of concurrent systems, parametric in a notion of *observations* available to the adversary. (2) We give a sufficient condition for actions to be indistinguishable in the general case of LTS models. (3) We apply this theory to the setting of Dynamic Condition Response Graphs, showing how (a) deciding indistinguishability is generally coNP-hard; and (b) the practically useful result that secrecy is guaranteed under certain graph isomorphisms.

*Overview* After related work in the next subsection, Section 2 defines indistinguishability on runs, serving as the foundation for definitions and characterisation of secrecy in Section 3. We then apply the theory to Dynamic Condition Response (DCR) graphs in Section 4, in particular obtaining both the abovementioned infeasibility and isomorphism results.

For want of space, some proofs have been relegated to App. A.2.

### 1.1 Related work

The present work draws from and contributes to work in the overlapping spaces of both information flow security and process modelling, in particular business process modelling. In information flow security, it is helpful to distinguish between *probabilistic* and *possibilistic* secrecy.

*Probabilistic secrecy* is concerned with defining secrecy of information over probabilities; can an adversary make *a good guess* on secret information, even without knowing with certainty if their guess is correct. Examples include Perfect Secrecy [28], flow models [22] and probabilistic non-interference [12]. However, this work concerns *possibilistic* secrecy.

*Possibilistic secrecy* is concerned with defining secrecy of information over certainty; can an adversary ever know some secret information *with certainty*. Pos-

sibilistic secrecy definitions include non-interference [11], non-deducibility [30], restrictiveness [21] and more (e.g. [33]). We, to some extent, follow Sutherland's non-deducibility methodology , which defines *information functions* – our *observation functions* are essentially parametric versions of these – and define that secrecy is upheld when any deducible information flowing from one information function to another is not secret. In that sense, our work can be considered a specialisation of non-deducibility of actions but translated to process models, parametric in notions of observability and descriptive rather than prescriptive. The observation function gives rise to an indistinguishability relation upon which we define secrecy. This approach is similar to that of [14], which defines secrecy in epistemic modal logic by way of an indistinguishability relation on *possible worlds* [9], i.e. scenarios that are considered possible by the observing agent. Epistemic logic is very powerful and general tool. Our goal in this paper has been to create a theory for dealing with secrecy solely in process models. As such we have specialised notions from epistemic logic where possible, and created new where appropriate. Since the temporal logic HyperCTL$^*$ for Hyperproperties is known to overlap with epistemic logic [3], some the of the results in this paper could, in all likelihood, be rediscovered using HyperCTL$^*$.

*Secrecy in process models* Secrecy is also a well-studied concept in process modelling. There has been headway in language based-approaches, see [27] where static analysis is made with a security focus, e.g. [18]. In the area of business process modelling [24] uses an extension of BPMN to capture privacy requirements, which [2] extends with model checking techniques to detected unintended leakages. In [10] the authors (1) translate possibilistic security properties from system models to LTSes, and (2) lift the bases of security properties from trace-equivalence to weak bisimulation. This work similarly concerns LTSes, and lift secrecy from trace-equivalence, but differs by focusing on a single notion of secrecy: non-deducibility, and only of actions, while lifting this property, not only to weak bisimulation, but to any system equivalence through generalisation. Related to (2) is [26], which defines non-interference properties in the process algebra CSP showing how different information flow security definitions reduce to different system equivalences. Inspired by this idea, our observation function is parametric in *notions of observability* – what *kind* of information an adverssary can observe during a run. This term is inspired by the work [31,32], where different notions of observability of systems define different system equivalences. By being parametric in notions of observability, the secrecy notion of actions remains independent of an underlying system equivalence.

## 2 Indistinguishability

In this section we define indistinguishability of Labelled Transitions System (LTS) runs, parametric in the set of observed events, $\pi$. We call the relation $\pi$-indistinguishability, which is the foundation of our later secrecy definitions.

We note that while the LTS model is to some degree an obvious choice of model, pervasive as it is in both concurrency theory and semantics of business

process notations, the present development is phrased in terms of *runs* of LTSes , so conceivably the theory would apply also in other settings that provide notions of runs, e.g., asynchronous transition systems [20].

**Definition 1 (Labelled transition system[23]).** *A labelled transition system is a quadruple $T = (S, s_0, E, \delta)$, where $S$ is the set of states, $s_0 \in S$ is the initial state, $E$ is a set of actions, and $\delta$ is a set of transition triplets $(s, e, s')$ where $s, s' \in S$ and $e \in E$. We write $s \xrightarrow{e} s'$ for $(s, e, s') \in \delta$, and $s \xrightarrow{e} s' \xrightarrow{e'} s'' \ldots$ for $(s, e, s'), (s', e', s''), \ldots \in \delta$.*

**Definition 2 (Run fragments[1]).** *Let $T = (S, s_0, E, \delta)$ be an LTS. A* run fragment *in $T$ is a sequence of transitions s.t. $s \xrightarrow{e} s' \xrightarrow{e'} s'' \xrightarrow{e''} \ldots$ If $s_0 = s$, we call that run fragment a run. We denote the set of all runs in $T$ as $R^T$.*

Note that a run fragment may be infinite. If a run fragment is finite we refer to the last state as the *final* state. If there exists a transition $s \xrightarrow{a} s'$ in a run fragment $r$, we say that action $a$ is in $r$, or that $r$ contains $a$, denoted $a \in r$.

During a run, the agents in the system are provided with information relevant to the correct execution of the specific system. This is encapsulated in the *observation function*, which is parametric in what *kind* of information this is ($\mathcal{N}$):

**Definition 3 (Observation function).** *Let $\mathcal{N}$ be a universe of observations. An* observation function *of an LTS $T$ is a function $O_{\mathcal{N}} : r^{\infty} \to \mathcal{N}^{\infty}$, where $r^{\infty}$ is the set of run fragments in $T$, and $\mathcal{N}^{\infty}$ is the set of sequences of observations.*

Intuitively, $O_{\mathcal{N}}$ gives the observation, as defined by the universe $\mathcal{N}$, of a run fragment in $T$. A basic example is an LTS that produces strings of 0's and 1's. The universe of observations ($\mathcal{N}$) could then be $\{1\}$, and $O_{\mathcal{N}}$ could be the sequence of 1's traversed by a given run. We will see more interesting examples of $O_{\mathcal{N}}$ in the example in Sec. 4.2. Our observation functions are essentially parametric versions of [30]'s information functions.

The intuition behind the observation function is that each agent participates in the execution of the system, and because of that participation, observe the actions of other agents. Given their prior knowledge, i.e. runs of the process in which they are participating, this observation may give rise to information leaks. As a very basic example, suppose that the system has actions $a, b, c, d$, and the adversarial agent can observe actions $b$ and $c$. If this agent *knows* that the system admits only the two runs $\langle a\underline{b}da \rangle$ and $\langle ad\underline{c}a \rangle$, observing just a single $b$ or $c$ tells the agent exactly what the underlying run must have looked like.

In general, observations can be both more and less complex than simply restricting to particular actions. E.g., for some systems, an agent may observe all possible future traces in an observation, so that they may choose their actions based on the possible outcomes. In other systems, an agent may observe only information about the final state of the run, thus allowing them only to choose their actions on the current state of the system while ignoring the history of the run. This work gives only one formal example of $\mathcal{N}$ (see Section 4), but all

definitions not restricted to a specific choice of $\mathcal{N}$ are general under the central assumption that no further information is acquired than that provided by $O_{\mathcal{N}}$.

As mentioned, each agent is privy to only part of the system's actions. We assume that an agent has some responsibility over the actions in their partial system. To model this responsibility, we introduce the notion of an ownership:

**Definition 4 (Ownership).** *Let $T = (S, s_0, E, \delta)$ be an LTS. An* ownership*, denoted $\pi$ for* projection*, is any subset of $E$.*

Def. 4 allows us to define what an agent with ownership $\pi$ can see during a run. Intuitively an agent is *only* responsible for the actions in their ownerships, and so an agent's observation of a run should be limited to the information relevant to that agent's ownership. Ownerships allows us to augment the observation function, denoting that the observation is limited by $\pi$:

**Definition 5 ($\pi$-Observation function).** *Let $T = (S, s_0, E, \delta)$ be an LTS, $O_{\mathcal{N}}$ be an observation function, and $\pi$ be an ownership. A* $\pi$-observation function *of $T$, denoted $O^{\pi}_{\mathcal{N}}$, is $O_{\mathcal{N}}$ projected onto $\pi$. We require only that such a projection is idempotent and that $\pi = E \implies O^{\pi}_{\mathcal{N}} = O_{\mathcal{N}}$. Otherwise a projection is dependent only on the choice of $\mathcal{N}$.*

An adversarial agent may attempt to determine which actions outside their ownership have been previously executed, both by passively observing a run of the process, and by actively forcing the execution of other actions by choosing which of their own actions are (not) executed during a run.

We consider only the case where an agent has a single ownership, i.e. a single subset of actions. For most observation functions, it will be the case that observing multiple ownerships is equivalent to observing the union of those ownerships, but not always. An agent with multiple ownerships, i.e. an agent observing both $\pi$ and $\tau$ separately, is analogous to collaborating agents sharing observations, an interesting topic which we will leave as future work.

Using Def. 5 we define $\pi$-equivalence leading us to indistinguishability.

### 2.1 $\pi$-indistinguishability

We proceed as follows. First, we define an equivalence of runs using Def. 5. Next, we lift this idea to sets of runs. If the runs in both sets produce the same observations, we say that the two sets are $\pi$-indistinguishable (Lemma 8). We then only need to define when a set of runs characterises an action, and the indistinguishability relation on actions becomes obvious.

We define the relation of runs equivalent under observation of $\pi$:

**Definition 6 ($\pi$-equivalence).** *Given an LTS $T$, a $\pi$-observation function $O^{\pi}_{\mathcal{N}}$ and two run fragments $r_1, r_2$, we say that $r_1$ and $r_2$ are $\pi$-equivalent, denoted $r_1 \equiv^{\pi}_{\mathcal{N}} r_2$, iff*

$$O^{\pi}_{\mathcal{N}}(r_1) = O^{\pi}_{\mathcal{N}}(r_2)$$

*$\equiv^{\pi}_{\mathcal{N}}$ is an equivalence-relation as per equality. By $[\![r_1]\!]^{\pi}_{\mathcal{N}}$ we denote the class of runs equivalent to $r_1$. We lift the notation $[\![-]\!]^{\pi}_{\mathcal{N}}$ pointwise from run fragment to sets of run fragments: $[\![R]\!]^{\pi}_{\mathcal{N}} = \{[\![r]\!]^{\pi}_{\mathcal{N}} \mid r \in R\}$.*

We may omit to specify the implicit $O^{\pi}_{\mathcal{N}}$ assumed by the $[\![-]\!]^{\pi}_{\mathcal{N}}$ notation.

Essentially $[\![r]\!]^{\pi}_{\mathcal{N}}$ are the runs that the observer considers possible after run $r$. This heavily inspired by Halpern & O'Neill's $\mathcal{K}_i$ relation [14], and *possible worlds* in epistemic modal logic [9], however, the theory diverges from here.

The definition of $\pi$-equivalence immediately suggests that some run fragments are not $\pi$-equivalent, and can be *distinguished* by an agent observing $\pi$. We call this $\pi$-distinguishability and define it on sets of runs:

**Definition 7 ($\pi$-indistinguishability).** *Given an LTS $T$ and two sets of run fragments of $T$, $R_1$ and $R_2$, we say that $R_1$ is distinguishable by $\pi$, or $\pi$-distinguishable, from $R_2$ iff*

$$\exists r \in R_1.\ [\![r]\!]^{\pi}_{\mathcal{N}} \notin [\![R_2]\!]^{\pi}_{\mathcal{N}}$$

Indistinguishability of runs are easily derived from $\pi$-distinguishability:

**Lemma 8.** *Given an LTS $T$ and two sets of run fragments $R_1, R_2$, we say that $R_1$ is not $\pi$-distinguishable from $R_2$ and vice versa iff*

$$[\![R_1]\!]^{\pi}_{\mathcal{N}} = [\![R_2]\!]^{\pi}_{\mathcal{N}}$$

*If so, we say that $R_1$ and $R_2$ are indistinguishable by $\pi$, or $\pi$-indistinguishable.*

Intuitively, Lemma 8 says that given two $\pi$-indistinguishable sets of runs and an observation of a run from either set, an agent observing $\pi$ cannot know from which set it was chosen. This is an excellent basis for a possibilistic secrecy definition. We will now narrow the scope to actions and full runs of the system.

Going forward, we will refer to runs containing an action $a$ as $R^T_a$, s.t. $R^T_a = \{r \in R^T \mid a \in r\}$. Note that $R^T_a$ contains only runs, not run fragments. Lemma 8 easily lends itself to define when one action in the system cannot be distinguished by an agent with ownership $\pi$ from some other action:

**Definition 9 ($\pi$-indistinguishability of actions).** *Given an LTS $T = (S, s_0, E, \delta)$ and two actions $a, b \in E$. We say that $a$ is indistinguishable by $\pi$, or $\pi$-indistinguishable, from $b$ iff*

$$[\![R^T_a]\!]^{\pi}_{\mathcal{N}} \subseteq [\![R^T_b]\!]^{\pi}_{\mathcal{N}}$$

*This relation is trivially a preorder on actions. If $a$ is $\pi$-indistinguishable from $b$ and $b$ is $\pi$-indistinguishable from $a$, we say that $a$ and $b$ are $\pi$-indistinguishable, which is an equivalence-relation by the addition of symmetry to a preorder.*

On the surface, $\pi$-indistinguishability looks like a good definition for secrecy of actions: if an agent cannot distinguish the runs containing $a$ from the runs containing $b$, then surely $a$ is secret? Counter-intuitively, no: if $a$ is $\pi$-indistinguishable from $b$, an agent with ownership $\pi$ may *still* deduce that a run contains an $a$. E.g., suppose $R^T_a = R^T_b = \{r\}$, i.e., just a single run $r$ which contains both $a$s and $b$s. Since $R^T_a = R^T_b$, $a$ is trivially $\pi$-indistinguishable from $b$. But since $O^{\pi}_{\mathcal{N}}(r)$ is unique, an agent observing $\pi$ can deduce that $a$ is in the observed run, and so $a$ cannot be said to be secret.

Consequently, $\pi$-indistinguishability of actions does not necessarily say anything about what the observing agent can infer about the inclusion and cardinality of the actions for some given observation. $\pi$-indistinguishability is a step in the right direction, though, and serves as a good foundation for defining secrecy.

## 3 Observational $\pi$-secrecy

From the notion of $\pi$-distinguishability we derive a notion of an action being secret to an agent: that any observation of run containing that action is also an observation of a run without it.

**Definition 10 ($\pi$-secrecy of actions).** *Given an LTS $T$ and an action $a$, we say that $a$ is secret to $\pi$, or $\pi$-secret, iff*

$$\exists H \subseteq (R^T \setminus R^T_a).\ [\![H]\!]^\pi_\mathcal{N} = [\![R^T_a]\!]^\pi_\mathcal{N}$$

*We call $H$ a* $\pi$-hiding set *for $a$, and say that it* hides $a$ from $\pi$, *or just* hides $a$.

It is instructive to consider the case where $R^T_a = \emptyset$, i.e. when there is no run in the system containing the action $a$. In this case, the empty set is a hiding set, and so $a$ is $\pi$-secret. Intuitively, $a$ is secret since no agent will be able to observe a run and identify that $a$ is part of the run. An agent may still dedcuce that a run does *not* contain $a$ – we leave this strengthening of secrecy as future work.

If action $a$ is $\pi$-secret, an agent with ownership $\pi$ cannot distinguish a run that contains $a$ from a run that does not, as captured by the following theorem:

**Theorem 11.** *Let $a$ be $\pi$-secret, then for any run $r$ containing $a$, there must exist some run $r'$ not containing $a$, s.t. $r \equiv^\pi_\mathcal{N} r'$*

*Proof.* By contradiction. Assume that such an $r'$ does not exist. Then $[\![r]\!]^\pi_\mathcal{N} \notin [\![(R^T \setminus R^T_a)]\!]^\pi_\mathcal{N}$.
But then $\forall H \subseteq (R^T \setminus R^T_a).\ [\![H]\!]^\pi_\mathcal{N} \neq [\![R^T_a]\!]^\pi_\mathcal{N}$, which contradicts $\pi$-secrecy. □

Intuitively, Thm. 11 shows that Def. 10 captures exactly the essence of a possibilistic secrecy definition. When observing the system during a run with a $\pi$-secret action $a$, no agent observing $\pi$ can know that the run did, in fact, contain $a$ since another run without $a$ could have caused the same observation.

Having defined secrecy of an action using sets of runs, we now want to lift this notion to actions making other actions secret. A further examination of hiding sets will allow us to define this secrecy property. In Section 4 we shall also see how hiding sets are a building block in designing processes with $\pi$-secret actions.

### 3.1 Hiding sets

An important special case is if the hiding set essentially replaces the secret action $b$ with some other action $a$:

**Definition 12** (**$\pi$-hiding actions**). *Given an LTS $T$ and two actions $a, b$ in $T$, we say that $a$ is a $\pi$-hiding action for $b$, or that $a$ hides $b$ (from $\pi$) iff*

$$\exists H \subseteq (R_a^T \setminus R_b^T).\ [\![H]\!]_{\mathcal{N}}^{\pi} = [\![R_b^T]\!]_{\mathcal{N}}^{\pi}$$

*We say that $H$ is a hiding set* in $R_a^T$.

Def. 12 says that if a hiding set for $b$ is in the runs that contain $a$'s and no $b$'s, then $a$ hides $b$. Note that the $\pi$-hiding action relation, unlike $\pi$-indistinguishability, is not transitive: if $a$ hides $b$, and $b$ hides $c$, the hiding set in $R_a^T$ for $b$ may contain $c$'s, and so is no valid hiding set of $c$. Also note that an action cannot hide itself, as the hiding set would be empty, so the hiding action relation must be irreflexive.

It follows from Def. 12 that if $a$ hides $b$, then $b$ is $\pi$-secret:

**Corollary 13.** *If $a$ hides $b$, then $b$ is $\pi$-secret.*

From Def. 12 also follows that if $a$ hides $b$, $b$ is $\pi$-indistinguishable from $a$:

**Lemma 14.** *If $a$ hides $b$, then $b$ is $\pi$-indistinguishable from $a$.*

Considering Cor. 13, and Lemma 14, then Def. 12 characterises exactly when $\pi$-indistinguishability implies secrecy. Def. 12 thus circumvents the problem with $\pi$-indistinguishability, where a run containing both $a$'s and $b$'s might produce a unique observation allowing for deduction of $b$ in the run.

Considering Cor. 13, and Lemma 14, then Def. 12 the central difference between $b$ being $\pi$-indistinguishable from $a$ and $a$ hiding $b$ lies in the runs containing both $a$'s and $b$'s ($R_a^T \cap R_b^T$). Intuitively, this is due to the example at the end of Section 2: two actions may be $\pi$-indistinguishable while still allowing for the existence of a run with both $a$'s and $b$'s that have a unique observation. As it happens, such ones turn out to be key to obtaining $\pi$-secrecy, so we investigate them in more detail.

As the following theorem states, if we design a system with action $a$ being $\pi$-indistinguishable from action $b$, while ensuring that any run with both an $a$ and a $b$ is $\pi$-indistinguishable from a run with a $b$ and no $a$, then $a$ will be secret:

**Theorem 15.** *If $a$ is $\pi$-indistinguishable from $b$, and $R_a^T \cap R_b^T$ is $\pi$-indistinguishable from $R_b^T \setminus R_a^T$, then $b$ hides $a$.*

*Proof.* We show that $b$ is a $\pi$-hiding action for $a$ by showing that $\forall r \in R_a^T.\ \exists \hat{r} \in (R_b^T \setminus R_a^T).\ r \equiv_{\mathcal{N}}^{\pi} \hat{r}$.

Let $r$ be any run in $R_a^T$. Since $a$ is $\pi$-indistinguishable from $b$ there must exist an $r' \in R_b^T$ s.t. $r \equiv_{\mathcal{N}}^{\pi} r'$. Then either $r' \in R_a^T$, or $r' \notin R_a^T$

If $r' \notin R_a^T$ then $r' \in (R_b^T \setminus R_a^T)$, and we are done.

If $r' \in R_a^T$ then $r' \in R_a^T \cap R_b^T$. And since $[\![R_a^T \cap R_b^T]\!]_{\mathcal{N}}^{\pi} \subseteq [\![R_b^T \setminus R_a^T]\!]_{\mathcal{N}}^{\pi}$, there must exist a run $r'' \in (R_b^T \setminus R_a^T)$ s.t. $r' \equiv_{\mathcal{N}}^{\pi} r''$. By transitivity, then $r \equiv_{\mathcal{N}}^{\pi} r''$. □

It follows that if no run contains both $a$'s and $b$'s, then $a$ being $\pi$-indistinguishable from $b$ is enough to ensure secrecy of $a$ from $\pi$:

**Corollary 16.** *If $a$ is $\pi$-indistinguishable from $b$ and $R_a^T \cap R_b^T = \emptyset$, then $b$ hides $a$.*

## 4 Applications to DCR Graphs

With our theory of secrecy in place, we turn to applications in business process modelling, specifically the declarative notation DCR [25,8,4,29]. To understand how secrecy may be important in this setting, consider the following Example.

*Example 17.* Imagine a manufacturing company. Occasionally, products are discovered to be defective and must be recalled. The finance or engineering department can independently initiate a recall, as can the board of directors. For the former two, the legal department can veto the recall, but not when the board of directors issues the recall. The recall itself is executed by the sales department.

This process requires secrecy since recalls are expensive, and costly to employee bonuses, and therefore hugely unpopular in the sales department. To avoid that the powerful sales departments exerts undue pressure on finance or engineering, the sales department must not discover which of these departments initiated a recall. The sales department can exert no such power over the board of directors, so a recall from the board of directors need not be kept secret.

### 4.1 DCR Graphs: Syntax & Semantics

We present here, rather tersely, core DCR syntax and semantics. We refer the reader to [5, Sec. 2] for a more comprehensive and gentler introduction.

The DCR formalism is a declarative process model with primary use in business process modelling (see e.g. [16,6,7]). The actions in the model have state, by virtue of the *marking* ($\mathtt{M}$) of the graph.

**Definition 18 (Dynamic Condition Response Graphs [7]).** *A DCR Graph is a tuple $G = (\mathtt{E}, \mathtt{M}, \mathtt{R}, \mathtt{L}, l)$, where*

- $\mathtt{E}$ *is a set of* events
- $\mathtt{M} = (\mathtt{Ex}, \mathtt{Pe}, \mathtt{In}) \in (\mathcal{P}(\mathtt{E})\times\mathcal{P}(\mathtt{E})\times\mathcal{P}(\mathtt{E}))$ *is the* marking *of the graph (mnemonics: Executed, Pending, and Included)*
- $\mathtt{R} = (\rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\div)$ *is the (binary)* relations *between events of the graph consisting, respectively, of* conditions*,* responses*,* includes *and* excludes*.*
- $\mathtt{L}$ *is the* labels *of the graph*
- $l$ *is the* labelling function *of the graph, mapping events to labels:* $l : \mathtt{E} \to \mathtt{L}$

For all intents and purposes, $\mathtt{E}$ is the set of actions in the system, and we shall conflate events and actions. A nontrivial labelling function may provide opportunities for additional secrecy as it allows processes different ways to take the same action. However, we leave this as future work, and so we let $l(e) = e$ for all events in $\mathtt{E}$. For that reason, we ignore $\mathtt{L}$ and $l$ in specifying a DCR Graph. When $e \in \mathtt{Ex}$, we say that $e$ *is executed*, and when $e \notin \mathtt{Ex}$, we say that $e$ is *not* executed. Similarly, for events' inclusion in the sets $\mathtt{Pe}$ and $\mathtt{In}$, which we refer to as *pending* and *included*, respectively. As shorthand we refer to $\{e' \mid (e, e') \in\rightarrow\}$ as $e \rightarrow$, and $\{e' \mid (e', e) \in\rightarrow\}$ as $\rightarrow e$, for some relation $\rightarrow$.

The semantics of a DCR Graph is an LTS defined as follows:

**Definition 19 (Semantics of a DCR Graph as an LTS [16]).** *Let $G = (\mathtt{E}, \mathtt{M}, \mathtt{R})$ be a DCR Graph ($\mathtt{L}$ and $l$ omitted). The corresponding LTS is the tuple $T(G) = (\mathsf{M}(G), \mathtt{M}, \mathtt{E}, \rightarrow \subseteq \mathsf{M}(G) \times \mathtt{E} \times \mathsf{M}(G))$, where $\mathsf{M}(G) \subseteq (\mathcal{P}(\mathtt{E}) \times \mathcal{P}(\mathtt{E}) \times \mathcal{P}(\mathtt{E}))$, and $\rightarrow$ is the transition relation given by $\mathtt{M}' \xrightarrow{e} \mathtt{M}''$ s.t.*

- $\mathtt{M}' = (\mathtt{Ex}', \mathtt{Pe}', \mathtt{In}')$ *is the marking before the transition*
- $\mathtt{M}'' = (\mathtt{Ex}' \cup \{e\}, \mathtt{Pe}'', \mathtt{In}'')$ *is the marking after the transition*
- $e \in \mathtt{In}'$
- $(\mathtt{In}' \cap \rightarrow\!\bullet e) \subseteq \mathtt{Ex}'$
- $\mathtt{In}'' = (\mathtt{In}' \cup e\!\rightarrow\!+) \setminus e\!\rightarrow\!\div$
- $\mathtt{Pe}'' = (\mathtt{Pe}' \setminus \{e\}) \cup e\!\bullet\!\!\rightarrow$

*By $\mathsf{M}(G)$, we denote the set of markings* reachable *by $\rightarrow$ from $\mathtt{M}$, including $\mathtt{M}$. By $R^{T(G)}$ we denote all (possibly infinite) runs of $G$, consistent with Def. 2.*

The works [7,16] also includes a notion of acceptance, however, that is inconsequential to this work and so we have omitted any such consideration.

In Section 2 and 3, we modelled the agents' prior knowledge as all possible runs in the system. By Def. 19, the initial marking of a DCR Graph captures this knowledge. So from here on in, we assume that each agent knows the relations and initial marking of the DCR Graph they are participating in.

DCR Graphs include a notion of *enabledness*. If there exists a run $r = r' \cdot e$ in $R^{T(G)}$, then the event $e$ is *enabled* in the marking reached by $r'$. Formally:

**Definition 20 (Enabled event [7]).** *For a DCR Graph $G$ with relations $\mathtt{R} = (\rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!+, \rightarrow\!\div)$, for some marking $\mathtt{M} = (\mathtt{Ex}, \mathtt{Pe}, \mathtt{In})$ we say that an event $e \in \mathtt{E}$ is* enabled*, denoted $\mathtt{M} \vdash e$, iff:*

$$e \in \mathtt{In} \wedge (\mathtt{In} \cap \rightarrow\!\bullet e) \subseteq \mathtt{Ex}$$

*if $e$ is not enabled in $\mathtt{M}$, we say that it is* disabled *in $\mathtt{M}$, denoted $\mathtt{M} \nvdash e$.*

Note that enabledness is included as bullet 3 and 4 in Def. 19, and we define it separately here only for convenience of reasoning later.

Example 17 can be straightforwardly modelled as a DCR process, by choosing the actions in Table 1, and embedding them into a graph as follows. Let $G = (\mathtt{E}, \mathtt{M}_0, \mathtt{R})$ be a DCR Graph, s.t. $\mathtt{E} = \{A, B, C, D, E\}$, $\mathtt{M}_0 = (\mathtt{Ex}_0, \mathtt{Pe}_0, \mathtt{In}_0)$, $\mathtt{R} = (\rightarrow\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!+, \rightarrow\!\div)$ and

$$\begin{aligned}
&\mathtt{Ex}_0 = \mathtt{Pe}_0 = \emptyset, \mathtt{In}_0 = \{A, B, C, D\},\\
&\rightarrow\!\bullet = \{(A, C), (B, C)\}, \bullet\!\!\rightarrow = \emptyset, \rightarrow\!+ = \{(B, E), (A, E), (D, E)\},\\
&\rightarrow\!\div = \{(A, A), (B, B), (C, C), (D, D), (E, E), (A, B), (B, A), (C, E)\}
\end{aligned}$$

Fig. 1 is a visualisation of the DCR Graph $G$.

*Example 21.* We sketch the semantics of $G$. Relations (arrows) define (a) how events prevent each other from executing, and (b) update the marking of $G$.

In the initial marking, finance, engineering or the board can initiate a recall (events $A$, $B$ and $D$, respectively). These events include the recall event $E$, which

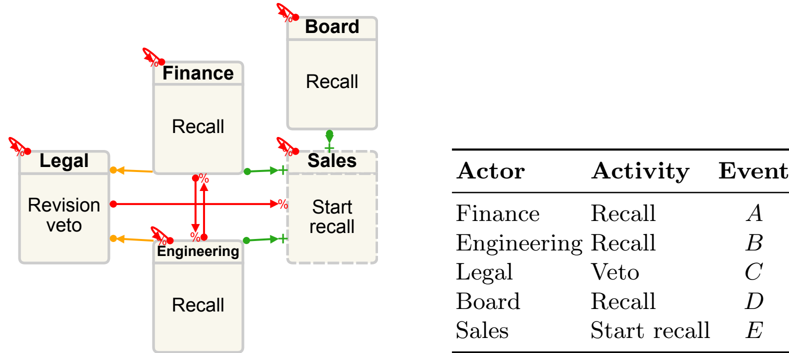



**Fig. 1.** Visualisation of the recall example DCR Graph.

| Actor | Activity | Event |
|---|---|---|
| Finance | Recall | $A$ |
| Engineering | Recall | $B$ |
| Legal | Veto | $C$ |
| Board | Recall | $D$ |
| Sales | Start recall | $E$ |

**Table 1.** Actions of the recall process

becomes eligible for execution. If finance or engineering initiates a recall, then the other is excluded from initiating one (exclusion relations). Legal can execute the revision veto (event $C$), but only if either $A$ or $B$ has happened (condition relations, in combination with mutual exclusion of $A$ and $B$). If legal executes a veto, sales lose the option of starting the recall (exclude relation $C, E$), unless the board subsequently re-instates the option (include relation $D, E$).

### 4.2 Ownership and observability

We now consider appropriate notions of ownerships and observations for DCR Graphs. The work [15] includes a notion of distribution of DCR Graphs, by a *projection parameter*: a subset of events and labels, which induces a subgraph. For a partitioning of the events in a DCR Graph, the collection of such subgraphs composes into the original graph. Simplifying this concept slightly, we simply consider a distribution as a subset of events – exactly an ownership as per Def. 4.

Similarly, for notions of observations. In [15] each event is considered a process in a distributed message-passing system. As an event is executed, a message identifying the executing event is sent to all the events that change marking or enabledness as a result. This, intuitively, translates into a notion of observability where the marking of all events that can affect marking or enabledness of an ownership is included in the observation of a run.

We consider a similar scenario where agents observe changes to the marking and enabledness of their ownership, but, crucially, cannot directly observe what events produced those changes. For an ownership $\pi$ and marking $\mathtt{M} = (\mathtt{Ex}, \mathtt{Pe}, \mathtt{In})$ we refer to $(\mathtt{Ex} \cap \pi, \mathtt{Pe} \cap \pi, \mathtt{In} \cap \pi)$ as the *marking of* $\pi$, denoted $\mathtt{M}|\pi$.

**Definition 22 (Observation function for distributed DCR Graphs).** *Let* $G = (\mathtt{E}, \mathtt{M}_0, \mathtt{R})$ *be a DCR Graph, where* $\mathtt{M}_0 = (\mathtt{Ex}_0, \mathtt{Pe}_0, \mathtt{In}_0)$. *Let* $d$ *(for DCR) denote a notion of observability, observing of the marking and enabledness of*

*events. Then* $o_d^\pi$*, i.e. the observation of a single transition in* $T(G)$*, is defined as*

$$o_d^\pi((\mathtt{Ex},\mathtt{Pe},\mathtt{In}) \xrightarrow{e} (\mathtt{Ex}',\mathtt{Pe}',\mathtt{In}')) = (\mathtt{Ex}' \cap \pi, \mathtt{Pe}' \cap \pi, \mathtt{In}' \cap \pi, \mathtt{En}' \cap \pi)$$

*where* $\mathtt{En}' = \{e' \in \mathtt{E} \mid (\mathtt{Ex}',\mathtt{Pe}',\mathtt{In}') \vdash e\}$ *– the enabled events, post-transition.*

*We let* $O_d^\pi$ *denote the pointwise lifting of* $o_d^\pi$ *to runs, with* consecutive duplicates removed.

$\mathtt{Ex}' \cap \pi$ contains the events in $\pi$ marked executed after the transition. Since an agent's prior knowledge is the initial marking, this provides exactly the information of how the executed-marking has changed for $\pi$, and similarly for the $\mathtt{Pe}' \cap \pi$ and $\mathtt{In}' \cap \pi$. Since $\mathtt{En}' \cap \pi$ contains the events that are enabled after the transition, the observation describes exactly any change to the marking and enabledness of $\pi$. Any consecutive duplicate elements are removed from $O_d^\pi$, since activity not affecting $\pi$, should be hidden from the agent; they should not be able to distinguish no action taken from actions taken that does not concern $\pi$.

We show in App. A.1 that deciding $\pi$-distinguishability in a DCR Graph with the observation function from Def. 22 is NP-hard. We do so by reduction of the known–to–be–NP-hard problem of event-reachability [8] to a decision-problem of $\pi$-distinguishability. It follows that determining $\pi$-indistinguishability of events using the observation function from Def. 22 is coNP-hard:

**Corollary 23.** *Determining* $\pi$*-indistinguishability of two events in a DCR Graph, using the observation function from Def. 22, is coNP-hard.*

Since the event-reachability problem is considered generally infeasible in DCR Graph [8,17], it follows that $\pi$-indistinguishability is generally infeasible, and, by extension, that $\pi$-secrecy is generally infeasible. We will therefore look for sufficient conditions for creating DCR-graphs with $\pi$-secret actions by design.

### 4.3 $\pi$-indistinguishability of automorphic events in DCR Graphs

In this section, we will show that (nontrivially) automorphic DCR events are $\pi$-indistinguishable. For ease of reasoning, we consider only DCR Graphs with finite runs. We conjecture the results to be general to all DCR Graphs. Infinite runs in DCR graphs are $\omega$-regular, and finite runs are regular [8], so the two structures are similar.

**Definition 24 (DCR Graph isomorphism).** *Given two DCR Graphs* $G = (\mathtt{E}_G, \mathtt{M}_G, \mathtt{R}_G)$ *and* $H = (\mathtt{E}_H, \mathtt{M}_H, \mathtt{R}_H)$*, where* $\mathtt{M}_G = (\mathtt{Ex}_G, \mathtt{Pe}_G, \mathtt{In}_G)$*,* $\mathtt{M}_H = (\mathtt{Ex}_H, \mathtt{Pe}_H, \mathtt{In}_H)$*,* $\mathtt{R}_G = (\rightarrow\!\bullet_G, \bullet\!\!\rightarrow_G, \rightarrow\!+_G, \rightarrow\!\div_G)$*, and* $\mathtt{R}_H = (\rightarrow\!\bullet_H, \bullet\!\!\rightarrow_H, \rightarrow\!+_H, \rightarrow\!\div_H)$*, an isomorphism of G and H is a bijection between the events of two DCR Graphs* $f : E_G \rightarrow E_H$ *s.t. f is marking-preserving:*

$$e \in \mathtt{In}_G \textit{ iff } f(e) \in \mathtt{In}_H, e \in \mathtt{Ex}_G \textit{ iff } f(e) \in \mathtt{Ex}_H, \textit{ and } e \in \mathtt{Pe}_G \textit{ iff } f(e) \in \mathtt{Pe}_H$$

*and f is relation-preserving:*

$$(e,e') \in \rightarrow\!+_G \textit{ iff } (f(e),f(e')) \in \rightarrow\!+_H, (e,e') \in \rightarrow\!\div_G \textit{ iff } (f(e),f(e')) \in \rightarrow\!\div_H,$$
$$(e,e') \in \bullet\!\!\rightarrow_G \textit{ iff } (f(e),f(e')) \in \bullet\!\!\rightarrow_H, \textit{ and } (e,e') \in \rightarrow\!\bullet_G \textit{ iff } (f(e),f(e')) \in \rightarrow\!\bullet_H$$

*We say that* $G$ *and* $H$ *are* isomorphic *if there exists an isomorphism* $E_G \to E_H$*.*

We lift a DCR-isomorphism $f$ as follows: to sets of events by pointwise application. Then to markings by pointwise application to a marking's sets. Then to transitions by pointwise application to the start- and end-marking and the executing event. Then to runs by pointwise application to the transitions. And, finally, to observations of $O^{\pi}_{d}$ (Def. 22), by applying $f$ pointwise to each element in the sequence (recall that $O^{\pi}_{d}$ produces sequences of quadruples of sets of events).

First, we show a foundational lemma, namely that isomorphisms on DCR Graphs are run-preserving – a run exists in a graph iff its isomorphic image exists in the isomorphic graph:

**Lemma 25.** *Given two DCR Graphs* $G$ *and* $H$ *isomorphic under* $f : G \to H$*, then* $r \in R^{T(G)}$ *iff* $f(r) \in R^{T(H)}$

*Proof.* ( $\Longrightarrow$ )
By induction over the length $n$ of $r$.

*Base step* $n = 0$. The empty run is trivially in all DCR Graphs (since we ignore the acceptance criteria of DCR Graphs).

*Induction step* $n > 0$. Let $r = r' \cdot t$, where $r'$ is a run with final state $\mathtt{M}$ and $t = (\mathtt{M}, a, \mathtt{M}')$ is a transition in $T(G)$. By IH, we know that $f(r') \in R^{T(H)}$, so we need only show that $f(t) = (f(\mathtt{M}), f(a), f(\mathtt{M}'))$ is a transition in $T(H)$.
Seeking a contradiction, assume not. Since, by IH, $f(\mathtt{M})$ is a state in $T(H)$, then $f(t)$ can only be missing from the transitions of $T(H)$ if (1.) $f(a)$ is not enabled in $f(\mathtt{M})$, or (2.) executing $f(a)$ from $f(\mathtt{M})$ does lead to the state $f(\mathtt{M}')$:

1. If $f(a)$ is not enabled in $f(\mathtt{M})$, then $f$ is not relation-preserving, since enabledness of $f(a)$ is a function only on the relations to $f(a)$ and the marking $f(\mathtt{M})$, see (Def. 20).
2. If executing $f(a)$ from $f(\mathtt{M})$ does lead to the state $f(\mathtt{M}')$, then $f$ is not relation-preserving, as the resulting marking is a function depends only on the relations of $f(a)$ and the marking $f(\mathtt{M})$ (see Def 19).

Leading to that contradiction that $f$ both is and is not relation-preserving.
The ( $\Longleftarrow$ ) direction is identical with the ( $\Longrightarrow$ ) direction, but for $f^-1$ rather than $f$. ☐

From Lemma 25 follows another foundational result, namely that isomorphisms of DCR Graphs *preserves the space of reachable markings*:

**Corollary 26.** *Given two DCR Graphs* $G$ *and* $H$ *isomorphic under* $f : G \to H$*, then*

$$\mathtt{M} \in \mathsf{M}(G) \textit{ iff } f(\mathtt{M}) \in \mathsf{M}(H)$$ ☐

Lemma 25 and Cor. 26 will be used as shortcuts for reasoning about $\pi$-indistinguishability in the following.

Lifting an isomorphism $f$ to observations of $O^{\pi}_{d}$ allows us to reason about isomorphic observations. Specifically, we can show that the observation of a run's isomorphic image is the same as the observation's isomorphic image of the run:

**Lemma 27.** *Consider two DCR Graphs $G$ and $H$ that are isomorphic under $f : G \rightarrow H$. In that case:*

$$\forall r \in R^{T(G)}.\ O_d^{f(\pi)}(f(r)) = f(O_d^{\pi}(r))$$

*Proof.* By induction on the length of $r$. First, note that $f(r)$ must be in $R^{T(H)}$ by Lemma 25. Let $n$ denote the length of $r$.

*Base step* If $n = 0$, then $r = f(r)$ is the empty run. The rest follows trivially.

*Induction step* Let $r = r' \cdot t$, for some transition $t$, where $O_d^{f(\pi)}(f(r')) = f(O_d^{\pi}(r'))$ by IH. Seeking a contradiction, assume that

$$O_d^{f(\pi)}(f(r)) \neq f(O_d^{\pi}(r))$$

Then, by IH and definition of $O_d^{\pi}$ (Def. 22),

$$o_d^{f(\pi)}(f(t)) \neq f(o_d^{\pi}(t))$$

Then either

$$f(\mathtt{Ex}_G{\downharpoonleft_\pi}) \neq \mathtt{Ex}_H{\downharpoonleft_{f(\pi)}} \quad \text{or} \quad f(\mathtt{Pe}_G{\downharpoonleft_\pi}) \neq \mathtt{Pe}_H{\downharpoonleft_{f(\pi)}} \quad \text{or}$$
$$f(\mathtt{In}_G{\downharpoonleft_\pi}) \neq \mathtt{In}_H{\downharpoonleft_{f(\pi)}} \quad \text{or} \quad f(\mathtt{En}_G{\downharpoonleft_\pi}) \neq \mathtt{En}_H{\downharpoonleft_{f(\pi)}}$$

all of which violate Cor. 26. □

Lemma 27 intuitively says that being given the observation of the isomorphic image of a run is equal to being given the isomorphic image of the observation of a run. This, in turn, implies that an isomorphism is what we might call observation-preserving; we can apply the isomorphism to an observation without losing or gaining information about the run that produces it.

Recall that an isomorphism $f$ from a structure to itself is called an *automorphism*, and $e$ s.t. $f(e) = e$ is called a fixed point of $f$. We say that $e$ and $f(e)$ are *automorphic* and, if $e$ is no fixed point in $f$, they are *nontrivially automorphic*. If $f$ has $e \neq f(e)$, we say that $f$ is a *nontrivial automorphism*.

With these foundations in mind, we are ready to show that, given certain conditions of an ownership, a run and its automorphic image *produces the same observation*:

**Lemma 28.** *Given a graph $G$, an ownership $\pi$, and an automorphism $f$ on $G$, if the events of $\pi$ are fixed points in $f$, then, for any run $r \in R^{T(G)}$, $O_d^{\pi}(r) = O_d^{\pi}(f(r))$*

*Proof.* The proof follows from Lemma 27:

$$\begin{aligned}
& O_d^{\pi}(r) && \overset{\pi \text{ are fixed points in } f}{=} \\
& f(O_d^{\pi}(r)) && \overset{\text{Lemma 27}}{=} \\
& O_d^{f(\pi)}(f(r)) && \overset{\pi \text{ are fixed points in } f}{=} \\
& O_d^{\pi}(f(r))
\end{aligned}$$

□

Lemma 28 has immediate uses for $\pi$-indistinguishability; given a set of runs $R$, then $f(R)$ is a $\pi$-indistinguishable set. We will show how this applies to $\pi$-indistinguishability of events, after a Corollary, an immediately consequence of Lemma 25, needed for reasoning.

**Corollary 29.** *Given a graph $G$ and an automorphism $f$ on $G$. Then, for any automorphic events $e$ and $f(e)$, $f(R_e^{T(G)}) = R_{f(e)}^{T(G)}$* □

**Theorem 30.** *Given a graph $G$, an ownership $\pi$, and a nontrivial automorphism $f$ on $G$. If the events of $\pi$ are fixed points in $f$ then any nontrivial automorphic events in $G$ are $\pi$-indistinguishable.*

*Proof.* Fix the automorphic events $e$ and $f(e)$. Since $\pi$ are fixed points in $f$, then neither $e$ nor $f(e)$ is in $\pi$. We must now show that $[\![R_e^{T(G)}]\!]_d^\pi = [\![R_{f(e)}^{T(G)}]\!]_d^\pi$:

$$
\begin{aligned}
&[\![R_e^{T(G)}]\!]_d^\pi && \overset{\text{Def. 6}}{=} \\
&\{\{r' \mid O_d^\pi(r') = O_d^\pi(r)\} \mid r \in R_e^{T(G)}\} && \overset{\text{Lem. 28 \& Cor. 29}}{=} \\
&\{\{r' \mid O_d^\pi(r')) = O_d^\pi(r)\} \mid r \in R_{f(e)}^{T(G)}\} && \overset{\text{Def. 6}}{=} \\
&[\![R_{f(e)}^{T(G)}]\!]_d^\pi
\end{aligned}
$$
□

Thm. 30 is the main takeaway from this section. It says that, as long as there exists an automorphism in which two events are nontrivially automorphic, then they are $\pi$-indistinguishable. This serves as a sufficient condition for creating $\pi$-indistinguishable events by design. Together with Cor. 16, we can then create $\pi$-secret events by design, as stated by the following Corollary:

**Corollary 31.** *Given a graph $G$, an ownership $\pi$, and a nontrivial automorphism $f$ on $G$. If the events of $\pi$ are fixed points in $f$, then any* mutually-exclusive *nontrivial automorphic events hide each other from $\pi$.* □

Note that nontriviality is a strict requirement here; the $\pi$-hiding action relation is *irreflexive*, making Cor. 31 inapplicable to *trivially* automorphic events.

It should be noted that the complexity of finding a nontrivial automorphism is unknown, but believed to be in NP, while neither being in P nor NP-complete [19]. Regardless, the real power of of Thm. 30 comes from the fact that it makes simple the process of constructing DCR Graphs with $\pi$-indistinguishable actions *by design*: Consider an existing process with an action $a$, that should eventually become secret. Adding another event $a'$ with relations equal to $a$'s, ensures $\pi$-indistinguishability, as there now exist a nontrivial automorphism, where $e$ and $e'$ are nontrivial automorphic events, and all other events are fixed points. Now making $e$ and $e'$ mutually exclusive ensures $\pi$-secrecy, by Cor. 31. This can, for example, be ensured by adding exclude-relations between $e$ and $e'$, and ensuring there exists no include-relations to them.

The DCR Graph ($G$) of the recall workflow (Fig. 1) has been constructed using the above procedure, with the actions $A$ and $B$ designed to be $\pi$-secret.

**Proposition 32.** *In $G$, for an agent with ownership $\pi = \{E\}$, $A$ and $B$ will be $\pi$-secret by isomorphism and mutual exclusivity of $A$ and $B$.*

*Proof.* Consider $f : \mathtt{E} \to \mathtt{E}$, s.t. $f(A) = B$, $f(B) = A$, and otherwise $f(e) = e$.

Clearly, $f$ is both relation- and marking preserving, so $f$ is a nontrivial automorphism on $G$. Likewise, $E$ is a fixed point in $f$, and $A$ and $B$ are nontrivially automorphic events. So, by Thm. 30, $A$ and $B$ are $\pi$-indistinguishable. Since $(A, B) \in \to\div$ and $(B, A) \in \to\div$, and $\to\!+A = \to\!+B = \emptyset$, at most *one* of the actions $A$ or $B$ can be in a run, by the semantics of DCR Graphs (Def. 19), so they are *mutually exclusive*. And so, by Cor. 31, $A$ and $B$ are $\pi$-secret. □

## 5 Future work

Possibilistic secrecy has its uses, especially in the business process modelling world, where a dual to non-repudiation is sometimes a requirement for multi-participant workflows. For security-critical tasks where an action must be completely obscured, however, possibilistic secrecy is not always enough and can be vulnerable to statistical analysis (see [33]). A natural next step is, therefore, to examine probabilistic secrecy properties of actions.

We have not considered collaborating adversarial agents in this work, except in Section 2, where we mentioned that collaborating agents can be considered a single agent with the union ownership, only for some notions of observability. Examining which notions of observability has this property might give rise to a deeper understanding of the relationship between system equivalence and secrecy.

Several weakenings of action secrecy also present themselves. E.g. keeping secrecy before/after a state, or allowing agents to deduce the first $n$ secret actions, but nothing after. Such weakening could allow leakage of secret actions as the process reaches an end-state, which could have applications in game theory.

Lastly, we have shown that computing $\pi$-indistinguishability for some notions of observability is infeasible in the general case. An approximation routine could serve as the foundation for more process models with secret actions.

## 6 Conclusion

In this paper, we have defined possibilistic secrecy of actions in concurrent systems where multiple actors collaborate on executing a process, but where not all actors are privy to all parts of that process. This secrecy of actions is parametric in notions of observability, which makes the definition usable regardless of what an agent can observe in specific system implementation. We have shown the practical use of action secrecy in the declarative process model of DCR Graphs, and through that, have shown how computing secrecy of actions for a chosen notion of observability is infeasible. We have also shown that, under that notion of observability, automorphic actions are indistinguishable to an agent, which serves as a foundation for creating processes with secret actions by design.

## References


1. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge, Mass (2008)
2. Belluccini, S., De Nicola, R., Dumas, M., Pullonen, P., Re, B., Tiezzi, F.: Verification of Privacy-Enhanced Collaborations. In: Proceedings of the 8th International Conference on Formal Methods in Software Engineering. pp. 141–152. FormaliSE '20, Association for Computing Machinery, New York, NY, USA (Oct 2020). https://doi.org/10.1145/3372020.3391553
3. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal Logics for Hyperproperties. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Abadi, M., Kremer, S. (eds.) Principles of Security and Trust, vol. 8414, pp. 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
4. Debois, S., Hildebrandt, T.: The DCR workbench: Declarative choreographies for collaborative processes. Behavioural Types: from Theory to Tools pp. 99–124 (2017)
5. Debois, S., Hildebrandt, T., Slaats, T.: Hierarchical Declarative Modelling with Refinement and Sub-processes. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) Business Process Management, vol. 8659, pp. 18–33. Springer International Publishing, Cham (2014)
6. Debois, S., Hildebrandt, T., Slaats, T.: Safety, Liveness and Run-Time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes. In: Bjørner, N., de Boer, F. (eds.) FM 2015: Formal Methods. pp. 143–160. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015)
7. Debois, S., Hildebrandt, T., Slaats, T.: Concurrency and asynchrony in declarative workflows. In: International Conference on Business Process Management. pp. 72–89. Springer (2016)
8. Debois, S., Hildebrandt, T.T., Slaats, T.: Replication, refinement & reachability: Complexity in dynamic condition-response graphs. Acta Informatica **55**(6), 489–520 (2018)
9. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT press (2004)
10. Focardi, R., Gorrieri, R.: Classification of security properties. In: International School on Foundations of Security Analysis and Design. pp. 331–396. Springer (2000)
11. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: 1982 IEEE Symposium on Security and Privacy. pp. 11–11 (Apr 1982). https://doi.org/10.1109/SP.1982.10014
12. Gray III, J.: Probabilistic interference. In: Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy. pp. 170–179 (May 1990). https://doi.org/10.1109/RISP.1990.63848
13. Guanciale, R., Gurov, D., Laud, P.: Business Process Engineering and Secure Multiparty Computation. In: Applications of Secure Multiparty Computation, p. 21. No. 13 in Cryptology and Information Security Series (2015)
14. Halpern, J., O'Neill, K.: Secrecy in multiagent systems. In: Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15. pp. 32–46. IEEE Comput. Soc, Cape Breton, NS, Canada (2002). https://doi.org/10.1109/CSFW.2002.1021805

15. Hildebrandt, T., Mukkamala, R., Slaats, T.: Safe distribution of declarative processes. In: Proceedings of the 9th International Conference on Software Engineering and Formal Methods. pp. 237–252 (2011)
16. Hildebrandt, T.T., Mukkamala, R.R.: Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. Electronic Proceedings in Theoretical Computer Science **69**, 59–73 (Oct 2011). https://doi.org/10.4204/EPTCS.69.5
17. Høgnason, T., Debois, S.: DCR Event-Reachability via Genetic Algorithms. In: Daniel, F., Sheng, Q.Z., Motahari, H. (eds.) Business Process Management Workshops. pp. 301–312. Lecture Notes in Business Information Processing, Springer International Publishing, Cham (2019)
18. Honda, K., Vasconcelos, V., Yoshida, N.: Secure Information Flow as Typed Process Behaviour. In: Smolka, G. (ed.) Programming Languages and Systems. pp. 180–199. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2000)
19. Lubiw, A.: Some NP-Complete Problems Similar to Graph Isomorphism. SIAM Journal on Computing **10**(1), 11–21 (Feb 1981). https://doi.org/10.1137/0210002
20. Mazurkiewicz, A.: Trace theory. In: Advanced Course on Petri Nets. pp. 278–324. Springer (1986)
21. McCullough, D.: Noninterference and the composability of security properties. In: Proceedings. 1988 IEEE Symposium on Security and Privacy. pp. 177–186. IEEE Comput. Soc. Press, Oakland, CA, USA (1988). https://doi.org/10.1109/SECPRI.1988.8110
22. McLean, J.: Security models and information flow. Tech. rep., Naval Research Lab Washington DC Center for High Assurance Computing Systems (1990)
23. Mukund, M., Nielsen, M.: CCS, locations and asynchronous transition systems. In: International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 328–341. Springer (1992)
24. Pullonen, P., Tom, J., Matulevičius, R., Toots, A.: Privacy-enhanced BPMN: Enabling data privacy analysis in business processes models. Software and Systems Modeling **18**(6), 3235–3264 (Dec 2019). https://doi.org/10.1007/s10270-019-00718-z
25. R. R. Mukkamala, T.: Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs,. In: Post-Proceedings of PLACES. vol. 69, pp. 59–73 (2010)
26. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. Journal of Computer Security **9**(1-2), 75–103 (Jan 2001). https://doi.org/10.3233/JCS-2001-91-204
27. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (Jan 2003). https://doi.org/10.1109/JSAC.2002.806121
28. Shannon, C.E.: Communication Theory of Secrecy Systems*. Bell System Technical Journal **28**(4), 656–715 (Oct 1949)
29. Slaats, T., Mukkamala, R.R., Hildebrandt, T., Marquard, M.: Exformatics Declarative Case Management Workflows as DCR Graphs. In: Proceedings of International Conference on Business Process Management (BPM2013) (2013)
30. Sutherland, D.: A model of information. In: Proceedings of the 9th National Computer Security Conference. vol. 247, pp. 175–183. Washington, DC (1986)
31. van Glabbeek, R.: The linear time-branching time spectrum I. In: International Conference on Concurrency Theory. pp. 278–297. Springer (1990)
32. van Glabbeek, R.: The linear time—branching time spectrum II. In: International Conference on Concurrency Theory (1993)

33. Wittbold, J.T., Johnson, D.M.: Information Flow in Nondeterministic Systems. In: IEEE Symposium on Security and Privacy. vol. 161 (1990)

# A Appendix

## A.1 Showing NP-hardness of distinguishability

In this section, we show how $\pi$-distinguishability is NP-hard. We use this fact to argue for the infeasibility of computing $\pi$-indistinguishability in the general case.

To show that $\pi$-distinguishability is NP-hard, we show a reduction from DCR event-reachability [8] to $\pi$-distinguishability. First, we recall the former problem:

**Definition 33 (The event-reachability problem [8]).** *For $(G, e)$, where $G = (\mathtt{E}, \mathtt{M}, \mathtt{R})$ is a DCR Graph, and $e \in \mathtt{E}$, the event-reachability problem of $(G, e)$ is deciding if there exists a run $\mathtt{M} \xrightarrow{r} \mathtt{M}'$ s.t. $\mathtt{M}' \vdash e$. We call such an $r$ a* witness *of the reachability of $e$. If such a witness exists, we say that $e$ is reachable.*

**Definition 34 (The $\pi$-distinguishability problem).** *For $(G, e, e', \pi)$, where $G = (\mathtt{E}, \mathtt{M}, \mathtt{R})$ is DCR Graph, $\pi$ is an ownership, and $e, e' \in \mathtt{E}$, the $\pi$-distinguishability problem is deciding if $R_e^{T(G)}$ is $\pi$-distinguishable from $R_{e'}^{T(G)}$ in $T(G)$ (Def. 7), under the notion of observability in Def. 22. If $R_e^{T(G)}$ is $\pi$-distinguishable from $R_{e'}^{T(G)}$ in $T(G)$ then there exists a run $r \in R_e^{T(G)}$ s.t. $[\![r]\!]_d^\pi \notin [\![R_{e'}^{T(G)}]\!]_d^\pi$. We call such an $r$ a* witness of distinguishability. *If such a witness exists, we say that $e$ is distinguishable from $e'$.*

**Lemma 35.** *For any DCR Graph, there exists a reduction $(G, e) \to (G', e, e', \pi)$ from event-reachability to $\pi$-distinguishability.*

*Proof sketch.* Construct a new DCR Graph, $G'$, with the new events $e'$ and $e''$. Let $e'$ have a condition to itself, making it unreachable. Add exclude relations from $e$ and $e'$ to $e''$, ensuring that they have the same effect on $e''$. Since $e'$ is unreachable, then $e$ and $e'$ will be $\pi$-distinguishable for $\pi = \{e''\}$ iff $e$ is reachable in $G$. □

From Lemma 35 follows the NP-hardness of $\pi$-distinguishability:

**Theorem 36.** *The $\pi$-distinguishability problem is NP-hard*

*Proof.* From [8] we know that event-reachability is NP-hard. From Lemma 35 we know that event-reachability is reducible to $\pi$-distinguishability. □

### A.2 Proofs

*Proof of Lemma 8*

*Proof.*

$$\begin{array}{rc} & \neg(\exists r \in R_1.\ [\![r]\!]^\pi_\mathcal{N} \notin [\![R_2]\!]^\pi_\mathcal{N}) \wedge \neg(\exists r' \in R_2.\ [\![r']\!]^\pi_\mathcal{N} \notin [\![R_1]\!]^\pi_\mathcal{N}) \\ \Leftrightarrow & (\forall r \in R_1.\ [\![r]\!]^\pi_\mathcal{N} \in [\![R_2]\!]^\pi_\mathcal{N}) \wedge (\forall r' \in R_2.\ [\![r']\!]^\pi_\mathcal{N} \in [\![R_1]\!]^\pi_\mathcal{N}) \\ \Leftrightarrow & \{[\![r]\!]^\pi_\mathcal{N} \mid r \in R_1\} \subseteq [\![R_2]\!]^\pi_\mathcal{N} \wedge \{[\![r']\!]^\pi_\mathcal{N} \mid r' \in R_2\} \subseteq [\![R_1]\!]^\pi_\mathcal{N} \\ \Leftrightarrow & [\![R_1]\!]^\pi_\mathcal{N} \subseteq [\![R_2]\!]^\pi_\mathcal{N} \wedge [\![R_2]\!]^\pi_\mathcal{N} \subseteq [\![R_1]\!]^\pi_\mathcal{N} \\ \Leftrightarrow & [\![R_1]\!]^\pi_\mathcal{N} = [\![R_2]\!]^\pi_\mathcal{N} \end{array}$$

□

*Proof of Cor. 13*

*Proof.* Since $R^T_a \subseteq R^T$ it follows that

$$\exists H \subseteq (R^T_a \setminus R^T_b).\ [\![H]\!]^\pi_\mathcal{N} = [\![R^T_b]\!]^\pi_\mathcal{N} \implies \exists H \subseteq (R^T \setminus R^T_b).\ [\![H]\!]^\pi_\mathcal{N} = [\![R^T_b]\!]^\pi_\mathcal{N}$$

□

*Proof of Lemma 14*

*Proof.*

$$\begin{array}{rc} & \exists H \subseteq (R^T_a \setminus R^T_b).\ [\![H]\!]^\pi_\mathcal{N} = [\![R^T_b]\!]^\pi_\mathcal{N} \\ \implies & \exists H \subseteq R^T_a.\ [\![H]\!]^\pi_\mathcal{N} = [\![R^T_b]\!]^\pi_\mathcal{N} \\ \iff & [\![R^T_b]\!]^\pi_\mathcal{N} \subseteq [\![R^T_a]\!]^\pi_\mathcal{N} \end{array}$$

□

*Proof of Cor. 16*

*Proof.* Special case of Thm. 15, where $R^T_a \cap R^T_b = \emptyset$, so $[\![R^T_a \cap R^T_b]\!]^\pi_\mathcal{N} = \emptyset$. As such $R^T_a \cap R^T_b$ is vacuously $\pi$-indistinguishable from $R^T_b \setminus R^T_a$. □

*Proof of Cor. 23*

*Proof.* Since the problem of determining $\pi$-indistinguishability of two events in a DCR-graph is the complement of the $\pi$-distinguishability problem (Def. 34), coNp-hardness follows trivially from Thm. 36. □

# Bibliography


[1] Stephen A. White. “Introduction to BPMN”. In: *Ibm Cooperation* (2004).

[2] Thomas T. Hildebrandt and Raghava Rao Mukkamala. “Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs”. In: *Post-proceedings of PLACES 2010* (2010).

[3] Wil MP Van der Aalst. “The Application of Petri Nets to Workflow Management”. In: *Journal of circuits, systems, and computers* 8.01 (1998), pp. 21–66.

[4] Wil MP Van der Aalst. “Formalization and Verification of Event-Driven Process Chains”. In: *Information and Software technology* 41.10 (1999), pp. 639–650.

[5] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. “Service-Oriented Composition in BPEL4WS.” In: *WWW (Alternate Paper Tracks)*. 2003, pp. 27–28.

[6] Marlon Dumas and Arthur HM Ter Hofstede. “UML Activity Diagrams as a Workflow Specification Language”. In: *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings 4*. Springer, 2001, pp. 76–90.

[7] George Coulouris et al. *Distributed Systems, Concepts and Design*. 5th ed. 2012. ISBN: 978-0-13-214301-1.

[8] T. Hildebrandt, R.R. Mukkamala, and T. Slaats. “Safe Distribution of Declarative Processes”. In: *Proceedings of the 9th International Conference on Software Engineering and Formal Methods* (Montevideo, Uruguay). 2011, pp. 237–252. DOI: 10.1007/978-3-642-24690-6_17.

[9] Wil MP van der Aalst and Mathias Weske. “The P2P Approach to Interorganizational Workflows”. In: *Advanced Information Systems Engineering: 13th International Conference, CAiSE 2001 Interlaken, Switzerland, June 4–8, 2001 Proceedings 13*. Springer, 2001, pp. 140–156.

[10] M. Pease, R. Shostak, and L. Lamport. “Reaching Agreement in the Presence of Faults”. In: *J. ACM* 27.2 (Apr. 1, 1980), pp. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. URL: https://doi.org/10.1145/322186.322188 (visited on 05/15/2020).

[11] Mads Frederik Madsen et al. "Collaboration among Adversaries: Distributed Workflow Execution on a Blockchain". In: *2018 Symposium on Foundations and Applications of Blockchain (SFAB '18)*. 2018.

[12] Google Finance. *Ether (ETH) Price*. Mar. 1, 2023. URL: `https://www.google.com/finance/quote/ETH-USD` (visited on 03/06/2023).

[13] ethereumprice. *Ethereum Gas Price Charts & Historical Gas Fees*. Mar. 1, 2023. URL: `https://ethereumprice.org/gas/` (visited on 03/06/2023).

[14] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1, 1990), pp. 299–319. ISSN: 0360-0300. DOI: `10.1145/98163.98167`. URL: `https://doi.org/10.1145/98163.98167` (visited on 03/13/2023).

[15] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI*. Vol. 99. 1999, pp. 173–186.

[16] Farah Abdmeziem, Saida Boukhedouma, and Mourad Chabane Oussalah. "On the Security of Business Processes: Classification of Approaches, Comparison, and Research Directions". In: *2021 International Conference on Networking and Advanced Systems (ICNAS)*. 2021 International Conference on Networking and Advanced Systems (ICNAS). Oct. 2021, pp. 1–8. DOI: `10.1109/ICNAS53565.2021.9628908`.

[17] Elio Goettelmann et al. "Paving the Way towards Semi-Automatic Design-Time Business Process Model Obfuscation". In: *2015 IEEE International Conference on Web Services*. IEEE, 2015, pp. 559–566.

[18] Amina Ahmed Nacer et al. "Obfuscating a Business Process by Splitting Its Logic with Fake Fragments for Securing a Multi-Cloud Deployment". In: *2016 IEEE World Congress on Services (SERVICES)*. IEEE, 2016, pp. 18–25.

[19] Jan Mendling et al. "Blockchains for Business Process Management-Challenges and Opportunities". In: *ACM Transactions on Management Information Systems (TMIS)* 9.1 (2018), pp. 1–16.

[20] Tiphaine Henry. "Towards Trustworthy, Flexible, and Privacy-Preserving Peer-to-Peer Business Process Management Systems". PhD thesis. Institut Polytechnique de Paris, 2022.

[21] Orlenys López-Pintado et al. "Caterpillar: A Blockchain-Based Business Process Management System." In: *BPM (Demos)* 172 (2017).

[22] An Binh Tran, Qinghua Lu, and Ingo Weber. "Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management." In: *BPM (Dissertation/Demos/Industry)*. 2018, pp. 56–60.

[23] Marcel Müller et al. "Engineering Trust-Aware Decentralized Applications with Distributed Ledgers". In: *Trust Models for Next-Generation Blockchain Ecosystems* (2021), pp. 1–35.

[24] Marcel Müller et al. "Towards Trust-Aware Collaborative Business Processes: An Approach to Identify Uncertainty". In: *IEEE Internet Computing* 24.6 (2020), pp. 17–25.

[25] Michael Backes, Birgit Pfitzmann, and Michael Waidner. "Security in Business Process Engineering". In: *Business Process Management: International Conference, BPM 2003 Eindhoven, The Netherlands, June 26–27, 2003 Proceedings 1*. Springer, 2003, pp. 168–183.

[26] Alexander W. Rohm, Gaby Herrmann, and Günther Pernul. "A Language for Modelling Secure Business Transactions". In: *Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99)*. IEEE, 1999, pp. 22–31.

[27] Barbara Carminati, Christian Rondanini, and Elena Ferrari. "Confidential Business Process Execution on Blockchain". In: *2018 Ieee International Conference on Web Services (Icws)*. IEEE, 2018, pp. 58–65.

[28] Wil Van Der Aalst. "Process Mining: Overview and Opportunities". In: *ACM Transactions on Management Information Systems (TMIS)* 3.2 (2012), pp. 1–17.

[29] Marcel Müller et al. "Process Mining in Trusted Execution Environments: Towards Hardware Guarantees for Trust-Aware Inter-organizational Process Analysis". In: *Process Mining Workshops: ICPM 2021 International Workshops, Eindhoven, The Netherlands, October 31–November 4, 2021, Revised Selected Papers*. Springer International Publishing Cham, 2022, pp. 369–381.

[30] David Basin, Søren Debois, and Thomas Hildebrandt. "On Purpose and by Necessity: Compliance under the GDPR". In: *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*. Springer, 2018, pp. 20–37.

[31] Simone Agostinelli et al. "Achieving GDPR Compliance of BPMN Process Models". In: *Information Systems Engineering in Responsible Information Systems: CAiSE Forum 2019, Rome, Italy, June 3–7, 2019, Proceedings 31*. Springer, 2019, pp. 10–22.

[32] Tobias Heindel and Ingo Weber. "Incentive Alignment of Business Processes". In: *Business Process Management: 18th International Conference, BPM 2020, Seville, Spain, September 13–18, 2020, Proceedings 18*. Springer, 2020, pp. 93–110.

[33] Frederik Haagensen and Søren Debois. "Incentive Alignment Through Secure Computations". In: *Business Process Management: 20th International Conference, BPM 2022, Münster, Germany, September 11–16, 2022, Proceedings*. Springer, 2022, pp. 343–360.

[34] Mads Frederik Madsen et al. "Transforming Byzantine Faults Using a Trusted Execution Environment". In: *15th European Dependable Computing Conference (EDCC '19)*. 2019.

[35] Mikkel Gaub, Malthe Ettrup Kirkbro, and Mads Frederik Madsen. "Trusted DCR: Decentralised Workflow Management in a Byzantine Setting". [Unpublished Master Thesis]. IT University of Copenhagen, June 1, 2018.

[36] Simon Johnson et al. "Intel Software Guard Extensions: EPID Provisioning and Attestation Services". In: *White Paper* 1.1-10 (2016), p. 119.

[37] William Stallings et al. *Computer Security: Principles and Practice*. 4th ed. Pearson Upper Saddle River, 2018.

[38] Jo Van Bulck et al. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1741–1758.

[39] Dayeol Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[40] Job Noorman et al. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices". In: *ACM Transactions on Privacy and Security (TOPS)* 20.3 (2017), pp. 1–33.

[41] Fatima Khalid and Ammar Masood. "Hardware-Assisted Isolation Technologies: Security Architecture and Vulnerability Analysis". In: *2020 International Conference on Cyber Warfare and Security (ICCWS)*. IEEE, 2020, pp. 1–8.

[42] Antonio Muñoz et al. "A Survey on the (in) Security of Trusted Execution Environments". In: *Computers & Security* (2023), p. 103180.

[43] Kit Murdock et al. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1466–1482.

[44] Kit Murdock et al. "Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble". In: *IEEE Security & Privacy* 18.5 (2020), pp. 28–37.

[45] Guoxing Chen et al. "Sgxpectre: Stealing Intel Secrets from Sgx Enclaves via Speculative Execution". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[46] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *Communications of the ACM* 63.7 (2020), pp. 93–101.

[47] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Communications of the ACM* 63.6 (2020), pp. 46–56.

[48] Kevin Driscoll et al. "Byzantine Fault Tolerance, from Theory to Reality". In: *Computer Safety, Reliability, and Security*. Ed. by Stuart Anderson, Massimo Felici, and Bev Littlewood. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 235–248. ISBN: 978-3-540-39878-3.

[49] K. Driscoll et al. "The Real Byzantine Generals". In: *The 23rd Digital Avionics Systems Conference (IEEE Cat. No.04CH37576)*. The 23rd Digital Avionics Systems Conference. Salt Lake City, UT, USA: IEEE, 2004, pp. 6.D.4-61–11. ISBN: 978-0-7803-8539-9. DOI: 10.1109/DASC.2004.1390734. URL: http://ieeexplore.ieee.org/document/1390734/ (visited on 01/11/2023).

[50] Gabriel Bracha and Sam Toueg. "Asynchronous Consensus and Broadcast Protocols". In: *J. ACM* 32.4 (Oct. 1, 1985), pp. 824–840. ISSN: 00045411. DOI: 10.1145/4221.214134. URL: http://portal.acm.org/citation.cfm?doid=4221.214134 (visited on 10/21/2019).

[51] Mads Frederik Madsen and Søren Debois. "On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems". In: *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC '20)*. 2020, pp. 159–168.

[52] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1, 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176 (visited on 05/15/2020).

[53] Byung-Gon Chun et al. "Attested Append-Only Memory: Making Adversaries Stick to Their Word". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. New York, NY, USA: ACM, 2007, pp. 189–204. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294280.

[54] Alexander Jaffe, Thomas Moscibroda, and Siddhartha Sen. "On the Price of Equivocation in Byzantine Agreement". In: *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*. PODC '12. Madeira, Portugal: Association for Computing Machinery, July 16, 2012, pp. 309–318. ISBN: 978-1-4503-1450-3. DOI: 10.1145/2332432.2332491. URL: https://doi.org/10.1145/2332432.2332491 (visited on 05/15/2020).

[55] Danny Dolev. *The Byzantine Generals Strike Again*. 1981, p. 30.

[56] Eralp A. Akkoyunlu, Kattamuri Ekanadham, and Richard V. Huber. "Some Constraints and Tradeoffs in the Design of Network Communications". In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. 1975, pp. 67–74.

[57] J. Gray. "Notes on Data Base Operating Systems". In: *Operating Systems* (1978), pp. 393–481.

[58] Gadi Taubenfeld. "Distributed Computing Pearls". In: *Synthesis Lectures on Distributed Computing Theory* 7.1 (2018), pp. 1–123.

[59] Michel Raynal. "Consensus in Synchronous Systems: A Concise Guided Tour". In: *2002 Pacific Rim International Symposium on Dependable Computing, 2002*. PRDC'02. Dec. 2002, pp. 221–228. DOI: 10.1109/PRDC.2002.1185641.

[60] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. "Replication, Refinement & Reachability: Complexity in Dynamic Condition-Response Graphs". In: *Acta Informatica* 55.6 (2018), pp. 489–520.

[61] John Morris. "Can Computers Ever Lie?" In: *The Philosophy Forum*. Vol. 14. 4. Taylor & Francis, 1976, pp. 389–401.

[62] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2015.

[63] Shireesh Apte and Nikolai Petrovsky. "Will Blockchain Technology Revolutionize Excipient Supply Chain Management?" In: *JEFC* 7.3 (Sept. 25, 2016). URL: `https://jefc.scholasticahq.com/article/910-will-blockchain-technology-revolutionize-excipient-supply-chain-management` (visited on 01/18/2023).

[64] Darcy WE Allen et al. "International Policy Coordination for Blockchain Supply Chains". In: *Asia & the Pacific Policy Studies* 6.3 (2019), pp. 367–380.

[65] Pankaj Dutta et al. "Blockchain Technology in Supply Chain Operations: Applications, Challenges and Research Opportunities". In: *Transportation Research Part E: Logistics and Transportation Review* 142 (Oct. 2020), p. 102067. ISSN: 13665545. DOI: `10.1016/j.tre.2020.102067`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S1366554520307183` (visited on 01/17/2023).

[66] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd ed. Upper Saddle RIiver, NJ: Pearson Prentice Hall, 2007. 686 pp. ISBN: 978-0-13-239227-3.