UNIVERSITÉ DU
LUXEMBOURG

PhD-FSTM-2020-17
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 20 May 2020 in Esch-sur-Alzette
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Ivana Vukotic
Born on 27 July 1985 in Podgorica (Montenegro)

Formal Framework for Verifying Implementations of Byzantine Fault-Tolerant Protocols Under Various Models

Dissertation defence committee

Dr. Peter Y.A. Ryan, Chairman
Professor, University of Luxembourg

Dr. Marcus Völp, Vice-chairman
Associate Professor, University of Luxembourg

Dr. Paulo Esteves-Veríssimo, Supervisor
Professor, University of Luxembourg

Dr. Stephan Merz, Member
National Institute for Research in Computer Science and Control

Dr. John Rushby, Member
Computer Science Laboratory SRI International

To Gigio,
my all

Acknowledgements

I would like to start by thanking my mentor Prof. Paulo Esteves-Veríssimo for his unconditional support and guidance. I feel honored to have been part of his team over the course of the last four years and proud of our joint achievements.

I would also like to thank my co-supervisor Dr. Vincent Rahli for his patience and for sharing his knowledge about formal verification. I very much appreciated our discussions, as well as his assistance with various coding challenges.

Similarly, I would like to thank my co-supervisor Dr. Ing. Marcus Völp for all the great discussions we had, and for all the feedback he provided on my thesis drafts. His “you can do it“ attitude made all the difference during this journey.

Over the years, many members of our research team become my dear friends: Douglas, Jérémie, Jiangshan, Natalie, Tong and Tulio. I also managed to find two “long lost brothers” in Christoph and David, and to “adopt” two amazing girls—Maria and Inês. Great thanks to all of them; I was always in a great company and I never felt alone.

I am deeply grateful to my parents, my brothers and their families, for their unconditional love and support. I am aware how difficult it was for them to see me go abroad. Their faith in me was instrumental in my drive to succeed and to complete this exploration in a timely manner.

I would like to extend my deepest gratitude to my husband for his love, patience and support. It was a great comfort not to have to go on this journey alone. He always had my back and helped me care for our family.

I am also truly grateful to my aunt Ljiljana, who was there for my family when it was truly necessary. Her commitment and strength assured me that I could purse my work without ever fearing they will lack support.

Many thanks to my friends from Montenegro, who always believed in me: Ana, Aleksandar, Marija and Milena.

Finally, and above all, I would like to thank my “my little ball of energy”. Without her, I would never even dare to start a PhD, and without her smiles and hugs I would never finish it.

Declaration

I, Ivana Vukotic, declare that this thesis, titled "Formal Framework for Verifying Implementations of Byzantine Fault-Tolerant Protocols Under Various Models" and the work presented therein are my own. I confirm that:

- this work was done wholly or mainly while in candidature for the degree DOCTEUR DE L'UNIVERSITÉ DU;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
- where I have consulted the published works of others, these are clearly attributed;
- where I have quoted from the works of others, the sources are always given;
- where the work presented in this thesis is based on work done by myself jointly with others, I have clearly outlined what was done by others and what I contributed;
- with the exception of such quotations, this is entirely my own work; and
- I have acknowledged all main sources of help.

Signed:

Date:

Abstract

The complexity of critical systems our life depends on (such as water supplies, power grids, blockchain systems, etc.) is constantly increasing. Although many different techniques can be used for proving correctness of these systems errors still exist, because these techniques are either not complete or can only be applied to some parts of these systems. This is why fault and intrusion tolerance (FIT) techniques, such as those along the well-known Byzantine Fault-Tolerance paradigm (BFT), should be used.

BFT is a general FIT technique of the active replication class, which enables seamless correct functioning of a system, even when some parts of that system are not working correctly or are compromised by successful attacks. Although powerful, since it systematically masks any errors, standard (i.e., "homogeneous") BFT protocols are expensive both in terms of the messages exchanged, the required number of replicas, and the additional burden of ensuring them to be diverse enough to enforce failure independence. For example, standard BFT protocols usually require $3f+1$ replicas to tolerate up to f faults.

In contrast to these standard protocols based on homogeneous system models, the so-called hybrid BFT protocols are based on architectural hybridization: well-defined and self-contained subsystems of the architecture (*hybrids*) follow system model and fault assumptions differentiated from the rest of the architecture (the *normal* part). This way, they can host one or more components trusted to provide, in a trustworthy way, stronger properties than would be possible in the normal part. For example, it is typical that whilst the normal part is asynchronous and suffers arbitrary faults, the hybrids are synchronous and fail-silent. Under these favorable conditions, they can reliably provide simple but effective services such as perfect failure detection, counters, ordering, signatures, voting, global timestamping, random numbers, etc. Thanks to the systematic assistance of these trusted-trustworthy components in protocol execution, hybrid BFT protocols dramatically reduce the cost of BFT. For example, hybrid BFT protocols require $2f+1$ replicas instead of $3f+1$ to tolerate up to f faults.

Although hybrid BFT protocols significantly decrease message/time/space complexity vs. homogeneous ones, they also increase structural complexity and as such the probability of finding errors in these protocols increases. One other fundamental correctness issue not formally addressed previously, is ensuring that safety and liveness properties of trusted-trustworthy component services, besides being valid inside the hybrid subsystems, are made available, or *lifted*, to user components at the normal asynchronous and arbitrary-on-failure distributed system level.

This thesis presents a theorem-prover based, general, reusable and extensible framework for implementing and proving correctness of synchronous and asynchronous homogeneous FIT protocols, as well as hybrid ones. Our framework comes with: (1) a logic to reason about homogeneous/hybrid fault-models; (2) a language to implement systems as collections of interacting homogeneous/hybrid components; and (3) a knowledge theory to reason about crash/Byzantine homogeneous and hybrid systems at a high-level of abstraction, thereby allowing reusing proofs, and *capturing the high-level logic* of distributed systems. In addition, our framework supports the *lifting* of properties of trusted-trustworthy components, first to the level of the local subsystem the trusted component belongs to, and then to the level of the distributed system. As case studies and proofs-of-concept of our findings, we verified seminal protocols from each of the relevant categories: the asynchronous PBFT protocol, two variants of the synchronous SM protocol, as well as two versions of hybrid MinBFT protocol.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our society strongly depends on critical information infrastructures such as electrical grids, autonomous vehicles, distributed public ledgers, etc. Unfortunately, ensuring that they operate correctly is very hard to achieve due to their complexity. Moreover, given the increasing number of sophisticated attacks on such systems (e.g., Stuxnet [Kus13], Cloudbleed [17b], Bitcoin Unlimited [17a], etc.), ensuring their correct behavior becomes even more necessary, especially because failures of these systems might have catastrophic consequences. For example, two crashes of Boeing 737 MAX 8 aircraft caused death of 346 people and loss of billions of dollars [19b].

Although testing is one of the most common techniques used for ensuring correctness of these systems, usually it is not feasible to test all possible runs of a system. In consequence, it is not rare that systems that have been deployed for a long period of time and tested by many users, still contain residual faults [Fon+17]. Ideally, we should ensure the correctness of critical systems, relying on a minimal trusted computing base, and to the highest standards possible, i.e., critical systems should be formally proven correct and these proofs should be machine checked. Machine checked proofs are most commonly produced using model checkers or interactive/automated theorem provers. Although model checking is an automatic verification technique which has been widely used for debugging programs and specifications, its application to systems of reasonable complexity—e.g., implementations of complex critical infrastructures—is all but straightforward, because of the state space explosion problem. To overcome the state space explosion problem, systems are usually modeled at a high level of abstraction, which sometimes conceals problems and details at the system implementation level. On the contrary, interactive theorem proving allows reasoning about implementations of systems, but it is very labor intensive to use a theorem prover to prove properties about critical systems.[1] For example, Klein at al. needed 22.2 person years to develop and formally prove correct a seL4 microkernel [Kle+09]. This is why critical systems are usually only partially verified when using theorem provers. Unfortunately, it turns out that errors can be found even in software that is checked by a theorem prover [Fon+17], in particular if the assumptions they are based on are over-simplifying.

One standard approach to mitigate the inevitability of faults, is to accept that systems or applications will indeed have faults or vulnerabilities, even if residual, and to tolerate them, through fault and intrusion tolerance (FIT) techniques. A quite powerful such class of techniques is the well-known Byzantine Fault-Tolerant State Machine Replication paradigm (BFT-SMR) [LSP82]; [CL99b]; [BSA14]. BFT-SMR is a general FIT technique that is used to turn any service into one that can tolerate *arbitrary* (a.k.a. Byzantine) faults, by extensively replicating the service to mask the behavior of a minority of possibly faulty replicas behind a majority of correct replicas, operating in consensus. A faulty replica is one that does not behave according to its specification. BFT-SMR can encompass a wide range faults in replicas, accidental or malicious. For example, it can be one that is controlled by an attacker, or simply one that contains a bug. The total number of replicas N is a parameter over the maximum number of faulty replicas f, which the system is configured to tolerate at any point in time. Typically, $N = 3f + 1$ for classical protocols such as PBFT [CL99b].

Unfortunately, classical BFT-SMR protocols such as [CL99b]; [CL99a]; [Cas01] are expensive both in terms of the messages exchanged, the required number of replicas, and the additional burden of ensuring them to be diverse enough to enforce failure independence. To address these issues, *hybrid architectures* [VCF00]; [VC02]; [CVN02]; [Ver03]; [CNV04]; [Cor+05] have been developed. These architectures assume existence of a subsystems of the architecture (*hybrids*), which rely on stronger guaranties from the rest of the architecture (the *normal* part). This way, these architectures allow the coexistence and interaction of components with largely diverse behavior, e.g., synchronous vs. asynchronous, or crash vs. Byzantine [Ver06]. In such models, "normal" components *trust* "special" components that provide a set of pre-defined properties with a high level of *trustworthiness*. In consequence, these *trusted-trustworthy* "special" components should undergo careful design and verification. Therefore, by relying on stronger assumptions (e.g., full synchrony or crash-failure), they can be unconditionally trusted to provide stronger properties to the entire hybrid distributed system, than what would be possible in an otherwise homogeneous system. For example, these hybrid sub-

---

[1] Klein et al. in [Kle+14] concluded that formal verification is cheaper than full high-assurance software certification, but it is about a factor of five more expensive than traditional good quality software engineering in embedded systems.

systems can be used for development of many reliable services [Fet03]; [ZSR02]; [VCF00]; [Ver+13]; [VRC97], such as: perfect failure detection, counters, ordering, signatures, voting, global timestamping, random numbers, etc.

Actually, this generic *architectural hybridization* paradigm has been showing great promise for BFT-SMR. Many so-called *hybrid* solutions have been designed to reduce the message/time/space complexity of BFT protocols [CNV04]; [Chu+07]; [Lev+09]; [Ver+13]; [Ver+10]; [Woo+11]; [Kap+12]; [CNV13]; [DCK16]; [BDK17] by relying on trusted-trustworthy components (e.g., message counters in MinBFT [Ver+13]) that cannot be tampered with and are trusted to only fail by crashing, or otherwise always deliver correct results. For example, when applied to BFT-SMR, hybrid solutions only require $2f+1$ replicas instead of $3f+1$, to tolerate f faults, and normally have lower message complexity. An increasing number of off-the-shelf hardware systems are now providing trusted environments [19i]; [19m]; [19h]; [ERT17], thereby enabling the further development and large-scale use of hybrid protocols, by hosting trusted-trustworthy components in those environments.

Although hybrid BFT-SMR protocols significantly cut replication and execution costs vs. homogeneous ones, they have increased structural complexity, which typically induces a higher propension for errors. Also, to the best of our knowledge, there are no formal proofs showing that the safety and liveness properties provided by trusted-trustworthy component services inside the hybrid subsystems, are indeed available in a deterministic way, or *lifted*, to user components at the "normal" fault-prone distributed system level.

As explained above, the complete or thorough verification of complex systems and applications not being feasible, it is mitigated by making them resort to FIT libraries and runtime subsystems, masking whatever residual faults may exist behind a majority of correct replicas. These FIT subsystems become now a single point of failure (SPoF) and as such, we should guarantee their correctness to the highest standards known. That is, the proof of their correctness should be checked by a machine and their model refined down to machine code. Unfortunately, as pointed out in [DHZ15], most distributed algorithms, including BFT protocols, are published in pseudo-code or, in the best case, a formal but not executable specification, leaving their safety and liveness questionable. Moreover, Lamport, Shostak, and Pease wrote about such programs [LSP82]: "We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm.", which turned out to be true—many production systems, although tested and used by a significant number of users, still contain errors [Fon+17]. If we add on top of that the fact that many blockchain protocols either use or are inspired by BFT-SMR protocols [SBV18]; [DSW16]; [Kok+16]; [Abr+16]; [Abr+18]; [PS17]; [Luu+16]; [19c]; [19k]; [19j], it is clear that proving correctness of these protocols is extremely important.

1.1 Contributions

Anticipating the impact and widespread use of FIT systems, and to support the development of correct FIT systems, we present here: *A generic and extensible theorem-prover based framework for systematically supporting the mechanical verification of homogeneous and hybrid FIT protocols and their implementations, where processes communicate via message passing. We equipped our framework with a knowledge-level theory that enables reasoning about homogeneous and hybrid systems at a high-level of abstraction, thereby allowing reusing proofs, and capturing the high-level logic of distributed systems. In addition, our framework supports compositional reasoning, e.g., through mechanisms to lift properties about trusted-trustworthy components, to the level of the distributed systems they are integrated in.* We provide more details about our contributions in the next sections.

Although the implementation we present here is Coq specific, our model and reasoning patterns can be implemented using another theorem prover (e.g., Isabelle/HOL, PVS...). However, we are not aware of other frameworks that provide equivalent functionalities and reasoning patterns. We chose to rely on Coq because it relies on a small trusted core which was partially proven correct [Bar10] and which has been put under scrutiny for more than 30 years, as well as because it enables reasoning about implementations of protocols.

The work presented in this thesis was independently validated through peer-reviewed publications in international conferences of this area. The complete list of publications related to this research is:

- **Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq**
  Vincent Rahli, Ivana Vukotic, Marcus Völp, Paulo Esteves-Veríssimo
  European Symposium on Programming, 619-650 (April 2018).

- **An Ecosystem for Verifying Implementations of BFT protocols**
  Ivana Vukotic, Vincent Rahli, Marcus Völp, Paulo Esteves-Veríssimo
  EuroSys Doctoral Workshop (April 2018).

- **Asphalion: Trustworthy Shielding against Byzantine Faults**
  Ivana Vukotic, Vincent Rahli, Paulo Esteves-Veríssimo
  Proc. ACM Program. Lang. 3, OOPSLA, Article 138 (October 2019).

1.1.1 Reasoning about Homogeneous BFT-SMR protocols

Because most BFT-SMR protocols known today are actually homogeneous, we developed a generic and extensible theorem-prover based framework for proving the correctness of implementations of these protocols.

As part of our framework, we provide a model, called Byzantine Logic of Events (ByLoE), that captures the idea of arbitrary/Byzantine faults. This model is based on Lamport's *happened before* relation [Lam78], i.e., we model a distributed system as a collection of events happening at various locations, such that events are connected using the *happened before* relation. This relation defines a partial order between events.

One of the most fundamental concepts to reason about distributed systems is the concept of an event, which can been seen as a point in space/time at which something happened. In ByLoE, an event is an abstract entity that corresponds either: (1) to some correct behavior, i.e., to the handling of a protocol message, or (2) to some arbitrary activity about which no information is provided. We use those arbitrary events to model arbitrary/Byzantine faults.

To prove properties of distributed systems, one has to prove that these properties are true for all possible executions of a system. This includes proving both local invariants, as well as properties that hold about the collection of processes that form the distributed systems. We use induction on causal time, to prove both distributed and local properties. To simplify proofs of local properties, we have developed an automated proof technique, and to enable distributed reasoning we are relying on the standard quorum reasoning, which enables tracing back correct information between nodes.

We have successfully used this framework to prove a crucial safety property of an implementation of a complex BFT-SMR protocol called PBFT [CL99b]; [CL99a]; [Cas01]. We handle all the functionalities of the base protocol, including garbage collection and view-change, which are essential in practical protocols. Our performance evaluation shows that our version of PBFT is around one order of magnitude slower than the state-of-the-art BFT library, called BFT-SMaRT [BSA14]. Moreover, to show that our framework can be used for verification of protocols that rely on different system models—asynchronous as well as synchronous ones—we also proved safety of two implementations of a seminal synchronous protocol called SM [LSP82]. An advantage of SM over PBFT is its simplicity, which makes it a convenient protocol for learning how to use our framework.

1.1.2 Reasoning about Hybrid BFT-SMR protocols

Even though hybrid techniques significantly reduce message/time/space complexity, they increase the structural complexity of a protocol, thereby limiting the assurance we obtain in their correctness. This was the reason why we chose to also add support for development of correct hybrid BFT-SMR systems.

ByLoE does not support reasoning about systems that are composed of components nor about different failure assumptions (e.g., some components can fail arbitrarily, while others can only crash on failure). This motivated us to develop our Hybrid logic of Events (HyLoE), which enables reasoning about systems composed of multiple components that can have different failure assumptions. HyLoE supports reasoning about three types of events: (1) events where nodes follow their specification; (2) events that correspond to some arbitrary behavior of nodes; and (3) events that correspond to some arbitrary behavior of nodes, involving the invocation of a trusted component. In case an event corresponds to some arbitrary behavior of a node, which also involved the invocation of a trusted component, we can only rely on the information produced by the trusted component and we can not rely on information about the rest of the local subsystem.

Additionally, to provide support for implementing distributed systems as a collection of local subsystems (e.g., replicas), such that each local subsystem is composed of any number of components, we developed our Monadic Component language (MoC). To enable interactions between different components, as well as to hide complexity of a local subsystem, we used monads [Mog89]. Moreover, to enable implementing hybrid protocols, MoC allows tagging components as *trusted* in case components can only crash on failure, as well as *non-trusted* when components can fail arbitrary. Additionally, MoC allows modular reasoning by lifting properties proved about hybrid components residing in a local subsystem, to the level of that local subsystem.

We have successfully used our framework to prove, among other things, critical safety properties (e.g., agreement) of two versions of the seminal MinBFT hybrid protocol [Ver+13]: one based on USIGs (as in the original version) and one based on TrIncs [Lev+09]. We also managed to simplify some of the original proofs of those properties [Ver10]. Thanks to Coq's extraction mechanism, as well as our runtime, which supports running trusted components inside Intel SGX enclaves, we were able to compare performance of our verified version of MinBFT with our verified version of PBFT. Our performance evaluation shows that our verified version of MinBFT is faster than our verified version of PBFT.

1.1.3 Knowledge Reasoning

It turns out that formal verification of BFT-SMR protocols using theorem prover is very expensive. Namely, to verify the implementation of PBFT, which contains around 2 KLOC, we wrote more than 50 KLOC of specs and proofs, although we built some automation in form of Coq tactics. Unfortunately, in case system developers decide to change some protocol details, in most cases proofs have to be fixed, and sometimes even redone from scratch. To prevent this from happening, formal verification of BFT-SMR protocols should be done at a high-level of abstraction. This way, in case some low-level details change, the proof will not have to be redone. Also, these abstractions should enable re-usability of proofs by covering reasoning patterns that are common for many BFT-SMR protocols. For example, correctness of majority of BFT-SMR protocols comes from the fact that any two correct replicas will always be able to agree on the existence of one correct participant from which they have received a piece of information. These were the principles followed in our work, and we strongly believe that our formal framework will provide precious assistance to engineers attempting to prove correctness of BFT-SMR protocols.

To address the issues listed above, we equipped our framework with an *epistemic knowledge* theory. We chose to rely on epistemic knowledge for several reasons. First, reasoning about distributed protocols at the level of knowledge is close to the way system experts informally reason about distributed systems. For example, when designing a new distributed protocol system designers reason as follows: to achieve a task, or to simply evolve, participants need to make new discoveries and to exchange their knowledge so that others can know about it. Second, knowledge allows us to capture the high-level logic of distributed systems. For example, we already abstracted away crucial reasoning patterns for formal verification of BFT-SMR protocols, such as finding overlapping quorums, going back in time to find the correct replica that disseminated the information, etc. Moreover, reasoning at the level of knowledge allows reusing results proved once and for all at the abstract knowledge level, to derive properties of multiple concrete implementations. For example, we already used our high-level lifting property presented in Section 6.4 to prove safety of two variants of MinBFT protocol (USIG-based and TrInc-based). Finally, in case we have a lemma that is already part of our knowledge theory, that lemma can help developers of new protocols to decide upon a set of assumptions they have to make in order to prove a distributed property.

We provide knowledge theories in two flavors:

- A *library* to reason about distributed knowledge of homogeneous systems, called ByK (which stands for Byzantine Knowledge).
- A *sound knowledge calculus* to reason about both homogeneous and hybrid systems at a high-level of abstraction, called LoCK (which stands for Logic Of events-based Calculus of Knowledge). In addition, LoCK supports lifting properties proved about (trusted) components to the level of a distributed system.

Although our knowledge calculus is more powerful that our knowledge library, especially because it can be used for reasoning about both trusted and non-trusted components, we decided to keep both variants because both methods have their advantages and disadvantages.

Because our knowledge library is shallowly embedded in Coq, it is as expressive as Coq and one can use Coq's tactics to find parts of the proofs that are already proven. Also, to prove new properties, proof engineers do not have to be familiar with all rules that are already proven.

Thanks to the fact that our knowledge calculus is deeply embedded in Coq, there exists an abstraction barrier between high-level and low-level reasoning. Moreover, developing this calculus forced us to identify the primitive constructs (as constructors of the language) and principles (as derivation rules) necessary to reason about knowledge. Additionally, we believe that our knowledge calculus opens a door for further automation.

1.2 Thesis Outline

The remainder of this thesis is organized as follows:

- Chapter 2 discusses related work.
- Chapter 3 presents a model, called ByLoE (which stands for Byzantine Logic of Events) that captures the concept of arbitrary/Byzantine faults. This model comes with a collection of assumptions that capture standard fault and system models, allowing one to reason about systems with faulty participants, as well as proof tactics that capture common reasoning patterns.
- Chapter 4 presents a library, called ByK (which stands for Byzantine Knowledge) to reason about *distributed epistemic knowledge.* This library allows reasoning about homogeneous distributed protocols at a high level of abstraction (without having to worry about low level details), thereby allowing reusing proofs.
- In chapter 5 we demonstrate that our framework can be used for formal verification of homogeneous BFT-SMR protocols. We proved critical safety properties of a version of the landmark asynchronous protocol called PBFT, as well as of two versions of the landmark synchronous protocol called SM. Our performance evaluation shows that our version of PBFT is an order of magnitude slower than state-of-the-art BFT-SMR library, called BFT-SMaRt.
- Chapter 6 presents a model called HyLoE (which stands for Hybrid Logic of Events) to reason about programs composed of multiple components that can have different failure assumptions, as well as a language called MoC (which stands for Monadic Component language) to implement systems as collections of interacting hybrid components. Additionally, in this chapter we present methods to *lift* properties of (trusted) components of a hybrid subsystem to the level of the local subsystem it resides in(Section 6.4).

- Chapter 7 presents a sound knowledge calculus, called LoCK (which stands for Logic Of events-based Calculus of Knowledge), to reason about hybrid systems at a high-level of abstraction. As opposed to ByK, LoCK supports reasoning about both trusted and non-trusted data. In this chapter, we present several reasoning patterns, which we verified within LoCK, that are used to prove standard properties about both homogeneous and hybrid systems. Additionally, in this chapter we present methods to *lift* properties of (trusted) components of a local subsystem to the level of a distributed system (Section 7.7).

- In chapter 8 we demonstrate that our framework can be used for formal verification of hybrid BFT-SMR protocols. We proved critical safety properties of two versions of the seminal hybrid protocol called MinBFT. Thanks to the fact that our runtime environment supports execution of a trusted component inside Intel SGX, we were able to compare our verified version of MinBFT with our verified version of PBFT. In all our experiments, MinBFT was faster than PBFT.

- Chapter 9 concludes the thesis and discusses some future work.

Chapter 2

Related Work

As explained above, the main goal of this thesis is developing a general, reusable and extensible framework that can be used for formal verification of homogeneous and hybrid fault-tolerant protocols, which rely on synchronous or asynchronous model. To show that our framework can indeed be used for implementing and formally proving properties of these protocols, we proved agreement of several well-known fault-tolerant protocols: SM [LSP82], PBFT [CL99b] and MinBFT[Ver+13]. We provide a brief description of these protocols in Section 2.1. Next, because we built our framework using interactive theorem prover called Coq [19d]; [BC04], in Section 2.2 we explain some basic Coq constructs we used in rest of this thesis. Finally, because we are not the first ones to reason formally about fault-tolerant distributed systems, in Section 2.3 we provide more details about related work as well as how others approaches model Byzantine behavior.

2.1 Protocols

Many BFT-SMR protocols have been developed over the years, such as [LSP82]; [CL99b]; [Kap+12]; [Ver+13]; [BSA14]; [DCK16]; [Ver+09]; [Gue+10]; [Chu+07]; [Lev+09]; [BDK15]; [BDK17]; [Ami+11], to cite only a few. In this section we provide a brief descriptions of the protocols we used as a case study: **SM** [LSP82] (Section 2.1.1), **PBFT** [CL99b]; [CL99a]; [Cas01] (Section 2.1.2) and **MinBFT** [Ver+13] (Section 2.1.3). The notations we used in the sections below correspond to the notations used in original papers.

2.1.1 SM

To show that, our framework supports reasoning about synchronous protocols, we chose to verify the agreement property of two implementations of SM (short for *Signed Messages*), which is a synchronous Byzantine fault tolerant protocol. Lamport, Shostak and Pease introduced SM in [LSP82], and provided a pen-and-paper proof of its correctness. An advantage of SM over other protocols we verified (PBFT and MinBFT) is its simplicity, which makes it a convenient protocol for learning how to use our framework.

2.1.1.1 SM Recap

As it turns out, SM does not require any constraint on the number of tolerated faults to be correct. However, as the authors pointed out in [LSP82], the problem is vacuous when there are strictly less than $f + 2$ replicas, because in that case there would be strictly less than 2 correct replicas. There, replicas are called metaphorically *generals*. One of those generals is the *commander*, while all the others are *lieutenants*. The goal of the protocol is for the correct generals to reach agreement on the order of the commander. The algorithm starts with the *commander*, e.g., replica 0, signing and sending its value v (its order) to every lieutenant. We write $\langle v : 0 \rangle$ for that message, where 0 stands for replica 0's signature of the value v. Lieutenants receive orders, storing them locally in a vector V, and finally make a decision depending on the values they have received. The rest of the algorithm can be divided in one of the following steps:

1. Commander message: When lieutenant i receives a message of the form $\langle v : 0 \rangle$ from the commander, assuming that it has not received any value yet: (i) it saves value v in its vector V; (ii) appends its signature to the end of that message; (iii) and broadcasts the new message, i.e. $\langle v : 0 : i \rangle$, to all other lieutenants.

2. Lieutenant message: When lieutenant i receives a message of the form $\langle v : 0 : j_1 : ... : j_k \rangle$, containing a value v, which it has not received before, lieutenant i saves that value in its vector V. In case the number of lieutenants that signed the received message excluding the commander (i.e. k here) is less than f, lieutenant i appends its signature to the end of that message and broadcasts the new message, i.e. $\langle v : 0 : j_1 : ... : j_k : i \rangle$, to all other lieutenants who have not already signed this message.

3. Execution: When a lieutenant does not receive any more messages (see assumption **A3.** below), it calculates $\mathsf{choice}(V_i)$ in order to make a decision. Here choice is a deterministic function, which takes a vector of values and produces a single value.

Authenticity of generals as well as message integrity are ensured through signatures, and a replica accepts a message only if the signature is correct. We actually implemented two variants of the SM protocol, which we call SM_{seq} (SM with Sequential signatures) and SM_{mul} (SM with Multi signatures). As in SM [LSP82], in SM_{seq}, each lieutenant signs the whole message (i.e. the value with all the appended signatures), appends its signature to that message, and then disseminates it to other lieutenants (see `code/SM` for more details). As opposed to SM [LSP82], in $\mathsf{SM_{mul}}$, each lieutenant signs only the value (as opposed to signing the message along with the received signatures), adds its signature to the collection of received signatures, and then disseminates this new message to other lieutenants (see `code/SM2` for more details). We chose to implement and verify multiple variants of SM to illustrate the fact that the high-level knowledge reasoning principles we developed apply to protocols which assume that each node signs the whole message (i.e. the value with all the appended signatures) before it disseminates that message to other nodes, as well as to protocols that assume that each node signs only the value and disseminates its message to other nodes only after it appends its signature to the collection of received signatures. Additionally, our formal verification of both versions of SM shows that the safety of SM does not depend on the way messages are signed—the safety of SM comes from the fact that all correct nodes received the same commands by the end of $f + 1$ synchronous rounds (see Section 5.1 for more details).

SM is a synchronous protocol, i.e., it relies on bounded message generation and transmission delays (up to μ), as well as clock drifts (up to τ). On top of that, SM assumes the following: **A1.** Every message sent by a non-faulty process is delivered correctly; **A2.** The receiver of a message knows who sent it; **A3.** The absence of a message can be detected; **A4.a** A loyal general's signature cannot be forged, and any alternation of the contents of his signed message can be detected; **A4.b** Anyone can verify the authenticity of a general's signature. See [LSP82, Section 6] for examples of how these assumptions can be substantiated. We present in Section 3.1.7 the assumptions we used to verify our SM variants, as well as how they relate to the original SM assumptions mentioned above.

2.1.1.2 SM Properties

SM is both safe and live. Proofs of its safety and liveness are presented in [LSP82]. SM's correctness is stated in terms of two properties called *IC1* and *IC2*. *IC1* says that all loyal lieutenants obey the same order; while *IC2* says that if the commanding general is loyal, then every loyal lieutenant obeys the order he sends. These properties combine both safety and liveness properties. We have proved the agreement property of both our variants of SM (i.e., the safety part of *IC1*). The safety part of *IC1* says that the orders decided upon by any two loyal lieutenants must be the same, while its liveness part says that loyal lieutenants eventually decide upon orders.

2.1.2 PBFT

To demonstrate that the framework presented in this thesis is also capable of proving properties of fault-tolerant protocols that a way more complex than SM, as well as to show that our framework supports reasoning about asynchronous protocols, we chose to verify the agreement property of the seminal practical asynchronous BFT protocol (called *PBFT*). We have chosen PBFT because several BFT-SMR protocols designed since its publication, either use (part of) PBFT as one of their main building blocks, or are inspired by it, such as [Kap+12]; [Ver+13]; [BSA14]; [DCK16]; [Ver+09]; [Gue+10], to cite only a few. Therefore, a bug in PBFT could imply bugs in those protocols too. If we add on top of that rising number of the blockchain technologies and different cryptocurrencies which are either inspired by or adapt PBFT [SBV18]; [DSW16]; [Kok+16]; [Abr+16]; [Abr+18]; [PS17]; [Luu+16]; [19c]; [19k]; [19j], the importance of formal verification of this protocol becomes even higher. Castro provided a thorough study of PBFT: he described the protocol in [CL99b], studied how to proactively rejuvenate replicas in [Cas01], and provided a pen-and-paper proof of PBFT's safety in [CL99a]; [CL02]. Even though we use a different model—Castro used I/O automata (see Section 2.3.1), while we use a logic-of-events model (see Section 3.1)—our mechanical proof builds on top of his pen-and-paper proof. One major difference is that here we verify actual running code, which we obtain thanks to Coq's extraction mechanism.

2.1.2.1 PBFT Recap

We describe here the public-key version of PBFT, for which Castro provides a formal pen-and-paper proof of its safety. As mentioned above, PBFT is considered the first practical BFT-SMR protocol. Compared to its predecessors, it is more efficient and does not rely on unrealistic assumptions. It works with asynchronous, unreliable networks (i.e., messages can be dropped, altered, delayed, duplicated, or delivered out of order), and it tolerates independent network failures. To achieve this, PBFT assumes strong cryptography in the form of collision-resistant digests, and an existentially unforgeable signature scheme. It supports any deterministic state machine. Each state machine replica maintains the service state and implements the service operations. Clients send requests to all replicas and await for $f + 1$ matching replies from different replicas. PBFT ensures that correct replicas execute the same operations in the same order.

Assuming that set of replicas is denoted by $|R|$, PBFT requires $|R| = 3f + 1$ replicas to tolerate up to f faults. Replicas move through a succession of configurations called *views*. In each view v, one replica ($p = v\ mod\ |R|$) assumes the role of *primary* and the others become *backups*. The primary coordinates the votes, i.e., it picks the order in which client requests are executed. When a backup suspects

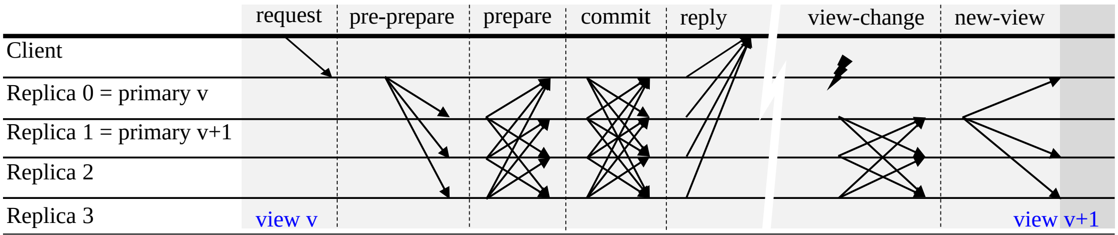


Figure 2.1: PBFT normal-case (left) and view-change (right) operations

the primary to be faulty, it requests a view-change (see below) to select another replica as new primary.

Normal-case. During normal-case operation, i.e., when the primary is not suspected to be faulty by a majority of replicas, clients send requests to be executed, which trigger agreement among the replicas. Various kinds of messages have to be sent among clients and replicas before a client knows its request has been executed. Figure 2.1 shows the resulting message patterns for PBFT's normal-case operation and view-change protocol. Let us discuss here the normal-case operation:

1. Request: To initiate agreement, a client c sends a request of the form $\langle \texttt{REQUEST}, o, t, c\rangle_{\sigma_c}$ to the primary, but is also prepared to broadcast it to all replicas if replies are late or primaries change. $\langle \texttt{REQUEST}, o, t, c\rangle_{\sigma_c}$ specifies the operation to execute o and a timestamp t that orders requests from the same client. Replicas will not re-execute requests with a lower timestamp than the last one processed for this client, but are prepared to resend recent replies.

2. Pre-prepare: The primary of view v puts the pending requests in a total order and initiates agreement by sending $\langle \texttt{PRE-PREPARE}, v, n, m\rangle_{\sigma_p}$ to all the backups, where m should be the n^{th} executed request.[1] The strictly monotonically increasing and contiguous sequence number n ensures preservation of this order despite message reordering.

3. Prepare: Backup i acknowledges the receipt of a pre-prepare message by sending the digest d of the client's request in $\langle \texttt{PREPARE}, v, n, d, i\rangle_{\sigma_i}$ to all replicas.

4. Commit: Replica i acknowledges the reception of $2f$ prepares matching a valid pre-prepare by broadcasting $\langle \texttt{COMMIT}, v, n, d, i\rangle_{\sigma_i}$. In this case, we say that the message with the sequence number n is *prepared* at i.

5. Execution & Reply: Replicas execute client operations after receiving $2f + 1$ matching commits, and after having executed all operations with lower sequence numbers. Once replica i has executed the operation o requested by client c, it sends $\langle \texttt{REPLY}, v, t, c, i, r\rangle_{\sigma_i}$ to c, where r is the result of applying o to the

---

[1] Pre-prepare messages introduced in [CL99b] and [Cas01] have different formats. Namely, pre-prepare messages introduced in [CL99b] contain additional field (original client's message).

service state. Client c accepts r if it receives $f+1$ matching replies from different replicas.

Client and replica authenticity, as well as message integrity are ensured through signatures of the form $\langle m \rangle_{\sigma_i}$. A replica accepts a message m only if: (1) m's signature is correct, (2) m's view number matches the current view, and (3) the sequence number of m is in the water mark interval (see below).

PBFT buffers pending client requests, processing them later in batches. Moreover, it makes use of checkpoints and water marks (which delimit sequence number intervals) to limit the size of all message logs and to prevent replicas from exhausting the sequence number space.

Why do we need commits? As explained in [Cas01, p.21–22], commits are necessary to ensure that decisions are consistent across views. As explained there, without the commit phase, a replica could collect a prepare certificate in some view v for a request r with sequence number n, and directly execute r. The primary of this view could be suspected to be faulty, leading to a view-change, before the other replicas collect prepare certificates for r. The new primary may be unaware of the prepared message, and might even have received pre-prepare/prepare messages with the same sequence number n for a request different than r. In that case, it is not possible for the new primary to know which request to choose (it should chose r to satisfy consistency). This problem is solved by an additional commit phase: after the prepare phase, we know that enough replicas have received a given request; and after the commit phase, we know that enough replicas have agreed to execute this request.

Garbage collection. Replicas store all correct messages that were created or received in a log. Checkpoints are used to limit the number of logged messages by removing the ones that the protocol no longer needs. A replica starts checkpointing after executing a request with a sequence number divisible by some predefined constant, by multicasting the message $\langle \texttt{CHECKPOINT}, v, n, d, i \rangle_{\sigma_i}$ to all other replicas. Here n is the sequence number of the last executed request and d is the digest of the state. Once a replica received $f+1$ different checkpoint messages[2] (possibly including its own) for the same n and d, it holds a proof of correctness of the log corresponding to d, which includes messages up to sequence number n. The checkpoint is then called *stable* and all messages lower than n (except view-change messages) are pruned from the log.

View-change. The view-change procedure ensures progress by allowing repli-

---

[2] Although $2f+1$ checkpoint messages were required in [CL99b], this requirement was relaxed in [Cas01].

cas to change the primary so as to not wait indefinitely for a faulty one. Each backup starts a timer when it receives a request and stops it after the request has been executed. Expired timers cause the backup to suspects the primary and requests a view-change. It stops receiving normal-case messages, and multicasts $\langle \texttt{VIEW-CHANGE}, v+1, n, s, \mathcal{C}, \mathcal{P}, i \rangle_{\sigma_i}$, reporting the sequence number n of the last stable checkpoint s, its proof of correctness $\mathcal{C}$, and the set of messages $\mathcal{P}$ with sequence numbers greater than n that backup i prepared since then.[3] When the new primary p receives $2f+1$ view-change messages, it multicasts $\langle \texttt{NEW-VIEW}, v+1, \mathcal{V}, \mathcal{O}, \mathcal{N} \rangle_{\sigma_p}$, where $\mathcal{V}$ is the set of $2f+1$ valid view-change messages that p received; $\mathcal{O}$ is the set of messages prepared since the latest checkpoint reported in $\mathcal{V}$; and $\mathcal{N}$ contains only the special *null* request for which the execution is a no-op. $\mathcal{N}$ is added to the $\mathcal{O}$ set to ensure that there are no gaps between the sequence numbers of prepared messages sent by the new primary. Upon receiving this new-view message, replicas enter view $v+1$ and re-execute the normal-case protocol for all messages in $\mathcal{O} \cup \mathcal{N}$.

We have proved a critical safety property of PBFT, including its garbage collection and view-change procedures, which are essential in practical protocols.

2.1.2.2 PBFT Properties

PBFT with $|R| = 3f+1$ replicas is safe and live. Its safety boils down to linearizability [HW87], i.e., the replicated service behaves like a centralized implementation that executes operations atomically one at a time. Castro used a modified version of linearizability in [Cas01] to deal with faulty clients. As presented in Section 5.2, we proved the crux of this property, namely the agreement property. Agreement states that, regardless of the view they are in, any two replies sent by correct replicas for the same request must be equal.

As informally explained by Castro [Cas01], assuming weak synchrony (which constrains message transmission delays), PBFT is live, i.e., clients will eventually receive replies to their requests.

2.1.2.3 Differences with Castro's Implementation

As mentioned above, besides the normal-case operation, our implementation of PBFT handles garbage collection, view-changes and request batching. However, we slightly deviated from Castro's implementation [Cas01], primarily in the way checkpoints are handled: a replica always require its own checkpoint before clearing its log. We chose to deviate from Castro's protocol in order to simplify our proofs of PBFT's correctness. Assuming the reader is familiar with PBFT, we here detail our deviation and refer the reader to [Cas01] for comparison.

---

[3] Unlike in [Cas01], in [CL99b] view-change messages do not carry the state.

We require a new primary to send its own view-change message updated with its latest checkpoint as part of its new-view message. If not, it may happen that a checkpoint stabilizes after the view-change message is sent and before the new-view message is prepared. This might result in a new primary sending messages in $\mathcal{O} \cup \mathcal{N}$ with a sequence number below its low water mark, which it avoids by updating its own view-change message to contain its latest checkpoint.

2.1.2.4 PBFT and Authentication

The version of PBFT, called PBFT-PK in [Cas01], that we implemented relies on digital signatures. However, we did not have to make any more assumptions regarding the cryptographic primitives than the ones presented above, and in particular we did not assume anything that is true about digital signatures and false about MACs. Therefore, our safety proof works when using either digital signatures or MAC vectors. As discussed below, this is true because we adapted the way messages are verified (we have not verified the MAC version of PBFT but a slight variant of PBFT-PK) and because we do not yet deal with liveness.

As Castro showed [Cas01, Chapter3], PBFT-PK has to be adapted when digital signatures are replaced by MAC vectors. Among other things, it requires "significant and subtle changes to the view-change protocol" [Cas01, Section 3.2]. Also, to the best of our knowledge, in PBFT-PK backups do not check the authenticity of requests upon receipt of pre-prepares. They only check the authenticity of requests before executing them [Cas01, p.42]. This works when using digital signatures but not when using MACs: one backup might not execute the request because its part of the MAC vector does not validate, while another backup executes the request because its part of the MAC vector checks out, which would lead to inconsistent states and break safety. Castro lists other problems related to liveness.

Instead, as in the MAC version of PBFT [Cas01, p.42], in our implementation replicas always check requests' validity when checking the validity of a pre-prepare. If we were to check the validity of requests only before executing them, we would have to assume that two correct replicas would either both be able to verify the data, or both would not be able to do so. This assumption holds for digital signatures but not for MAC vectors.

2.1.3 MinBFT

To show that, our framework supports reasoning about hybrid protocols, we chose to verify the agreement property of the seminal hybrid BFT-SMR protocol called MinBFT [Ver+13]. We believe that verifying MinBFT-like protocols is important because: (1) MinBFT is part of other protocols, such as [Kap+12]; [DCK16][4];

---

[4] MinBFT [Ver+13] is part of the Hyperledger Fabric umbrella [19e].

```
function createUI(msg) : UI {
  counter++;
  H:=hash(msg,id,counter,keys);
  return<<id,counter,H>>; }

function verifyUI(msg, UI) : bool {
  H:=hash(msg,UI.id,UI.counter,keys);
  return(UI.digest == H); }
```

Figure 2.2: Pseudo-code for USIG's operations

(2) many protocols such as [Ver+10]; [Kap+12]; [Ver+13]; [BDK17] rely on the same kind of trusted components as MinBFT (see Section 2.1.3.3); and (3) to the best of our knowledge MinBFT's trusted components (called USIGs) have the smallest TCB compared to other trusted components used in contemporary hybrid protocols.

In this section we first provide brief description of the trusted component on which MinBFT is based on, namely USIG (see Section 2.1.3.1). Next, we introduce a more general version of USIG, called TrInC (see Section 2.1.3.2). Finally, we provide a brief description of MinBFT protocol (see Section 2.1.3.3), as well as a brief description of its properties (see Section 2.1.3.4).

2.1.3.1 USIG

To achieve safety with only $2f + 1$ replicas, every MinBFT replica runs a local service called USIG (Unique Sequential Identifier Generator). Its purpose is to securely count messages so that replicas can know whether they have missed some messages. Every message generated by USIG-component is tagged with a certificate called UI (Unique Identifier). A UI is a triple of: an id (the replica's unique id), a counter value, and a signed hash (of the message/id/counter triple). USIGs provide only two simple operations *createUI* and *verify UI*, which generate and verify UIs (see pseudo-code above). Counter values produced by USIGs are monotonic (and without gaps) and therefore uniquely identify messages. This is guaranteed even when replicas are compromised because by definition USIGs execute inside trusted-trustworthy components, i.e., in tamperproof environments. To the best of our knowledge USIGs have the smallest TCB compared to other trusted components used in contemporary hybrid protocols, such as TrIncs discussed next.

2.1.3.2 TrInc

In [Lev+09], the authors introduced a new kind of trusted components called TrInc (which stands for Trusted Incrementer). TrInc is more general than USIG in the

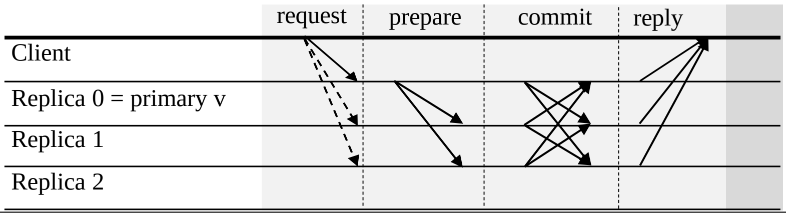


Figure 2.3: MinBFT normal-case

sense that it maintains multiple counters (one can dynamically add new counters through TrInc's interface), and that counters can have gaps: given a counter k, k's next counter value is provided by the client of the trusted component and has to be greater than the current value (see [Lev+09] for uses of these features). This is to contrast with a USIG, which increments its counter by one on each *createUI* call. Note that the fact that counters do not have gaps does not need to be enforced by the trusted components, which is made explicit when using TrInc instead of USIG. TrInc's flexibility comes at the price of slightly more complex trusted components. However, this flexibility makes TrInc compelling and led BFT implementations such as Hybster [BDK17] to be based on TrInc instead of USIG.

2.1.3.3 MinBFT Recap

As other such protocols do, MinBFT works in a succession of configurations called *views*. In each view v, the distinguished replica $p = v \ mod \ n$ (n is the total number of replicas), called the *primary*, is in charge of ordering client requests by assigning sequence numbers (the counter values generated by its USIG) to them. As long as the primary is not suspected to be faulty, MinBFT executes its normal-case operation (see figure above); and switches to a *view-change* operation otherwise. We focus here on the normal-case operation, which works as follows:

1. Request: To execute an operation o with timestamp t, client c sends a message $\langle \texttt{REQUEST}, c, t, o \rangle_{\sigma_c}$ to all replicas and waits for $f + 1$ matching replies from different replicas.

2. Prepare: When the primary p receives a request m, it calls its USIG component to generate a new identifier ui_i and sends $\langle \texttt{PREPARE}, v, m, ui_i \rangle$ to all other replicas (v is the current view).

3. Commit: Upon receipt of $\langle \texttt{PREPARE}, v, m, ui_i \rangle$, replica j calls its USIG to verify ui_i, generates a new identifier ui_j, and sends $\langle \texttt{COMMIT}, v, m, ui_i, ui_j \rangle$ to all other replicas.

4. Execution & Reply: If replica k receives $f+1$ valid $\langle \texttt{COMMIT}, v, m, ui_i, ui_j \rangle$ messages (i.e., the UIs are valid) from different replicas, it executes the request m, and sends the result r of this execution in a reply $\langle \texttt{REPLY}, k, seq, r \rangle_{\sigma_k}$ to the client. In addition, upon receipt of a new commit, k calls its USIG component to generate a new identifier ui_k and sends $\langle \texttt{COMMIT}, v, m, ui_i, ui_k \rangle$ to all others.

In all these steps, replicas only handle messages if: (1) the message is signed properly (for requests); (2) *prepare* messages come from the current primary; (3) the view number is the current one; and (4) upon receipt of a UI from a replica i, replicas check that they have already received all the UIs from i with lower counter values.

2.1.3.4 MinBFT Properties

MinBFT works with partially synchronous, authenticated reliable channels and it tolerates independent network failures. To achieve this, it assumes strong cryptography in the form of collision-resistant digests, and a non-existentially forgeable signature scheme.

2.2 Coq Notation

Coq [19d] is an interactive theorem prover, which implements a programming language called Gallina. Coq also implements a logic called Calculus of Inductive Constructions, which allows a user to prove properties about Gallina programs. It is implemented in OCaml.

In this Chapter we introduce some basic constructs used in this thesis. We refer the interested reader to a quick tutorial [Ber06] and a couple of books [BC04]; [Chl13]; [Pie+18].

Inductive types. In Coq, inductive types contain a list of constructors, such that each constructor from the list specifies how a new object of that type can be built:

```
Inductive TypeName :=
| constructor_1 : ... → TypeName
| constructor_2 : ... → TypeName
...
| constructor_n : ... → TypeName.
```

For example, natural numbers are defined such that a first constructor O corresponds to zero, and the second constructor S corresponds to the successor function:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

Using *pattern-matching* (i.e., expressions of the form `match`...`with`...`end`), one can compute a value of an inductive type for some input I, such that the value of the computed expression will depend on the pattern that input I satisfies. For example, following definition returns true in case a natural number n is different than zero, and otherwise returns false:

```
Definition not_zero (n : nat) :=
  match n with
  | O ⇒ false
  | S n ⇒ true
  end.
```

Type Classes. Similar to Java classes, Coq's classes provide an interface, which is composed of parameters of the class (p_1, p_2, ..., p_n) and the methods (m_1, m_2, ..., m_n) of the class:

```
Class classname (p_1 : Type_1) (p_2 : Type_2) ... (p_n : Type_n) := {
    m_1 : Type_1;
    m_2 : Type_2;
    ...
    m_n : Type_n }
```

One can prove properties that are true for any type a class can be instantiated with, and then later instantiate parameters and methods of the class with concrete types:

```
Instance instancename : classname p_1 p_2 ... p_n := {
    m_1 : Type_1;
    m_2 : Type_2;
    ...
    m_n : Type_n }.
```

Record. Records in Coq are defined and used the same way as any other imperative language. In Coq, record is essentially composed of a list of fields (field_1, field_2, ..., field_k), which can depend on a list of parameters (p_1, p_2, ..., p_n):

```
Record recordname p_1 p_2 ... p_n:= MkRecordname { field_1; field_2; ... field_k }.
```

	Running code	Crash	Byzantine (synch.)	Byzantine (asynch.)	Modular reasoning	Hybrid
DistAlgo	✓	✓	✗	✓	✗	✗
Actor Services	✗	✗	✗	✗	✓	✗
Aneris	✓	✓	✗	✗	✓	✗
Chapar	✓	✗	✗	✗	✗	✗
ConsL/DISEL/EventML/ IronFleet/ModP/ PSync/Verdi	✓	✓	✗	✗	✗	✗
PVS/Sally	✗	✓	✓	✗	✗	✗
ByMC/HO/IOA/TLA$^+$	✗	✓	✓	✓	✗	✗
Event-B	✓	✓	✓	✗	✗	✗
Ivy	✓	✓	✗	✓	✓	✗
This thesis	✓	✓	✓	✓	✓	✓

Figure 2.4: Comparison with related work

2.3 Formal Verification of Distributed Systems

First efforts to formally verify dependable and safety-critical systems were done using CSP [Hoa85]; [But+97]; [BPS98] and CCS [Mil80]; [KB07]. Although over the years many logics, models and tools have been developed to reason about distributed systems, in this section, we lists some of the most prominent ones (see Figure 2.4), and we discuss below how we relate to them. To the best of our knowledge, this thesis presents the first theorem prover based framework for verifying the correctness of implementations of homogeneous and hybrid fault-tolerant protocols under synchronous and asynchronous fault models. Also, we believe that we provide a first theorem prover checked knowledge theory, which allows reasoning about distributed protocols at high level of abstraction without having to worry about low-level details. Moreover, we believe that we are the first ones to provide machine-checked proofs of safety of implementations of the following protocols: SM, PBFT and MinBFT. We end this section by providing more details about related work as well as how others approached modeling Byzantine behavior in asynchronous environments.

2.3.1 Logics and Models

2.3.1.1 DistAlgo

DistAlgo [LSL17] is a high-level language for implementing control flows of distributed algorithms, where complex synchronizations can be expresses using high-level queries over message history sequences. Because, programs containing control flows with synchronization conditions expressed at high-level are extremely inefficient if executed straightforwardly, this paper also presents optimizations that automatically transform complex synchronization conditions into incremental updates of necessary values as messages are sent and received. The authors implemented operational semantics of the language, a prototype of the compiler and optimizations. DistAlgo is used for implementing a variety of distributed protocols including Paxos[Lam98], Multi-Paxos [CLS16] and PBFT [CL99b].

2.3.1.2 Event-B

Event-B [Abr10]; [Für+14b] is a set-theory-based language for modeling reactive systems and for *refining* high-level abstract specifications into low-level concrete ones. It supports code generation [MS11]; [Für+14a]; [Edm+12], with some limitations.[5] For example, code generation introduced in [Edm+12] does not work the with latest version of Rodin and requires specific versions of several other plug-ins; code generation introduced in [Für+14a] is not publicly available; and code generation introduced in [MS11] requires manual postprocessing and covers only a part of Event-B. The Rodin [Abr+10] platform for Event-B provides support for refinement, and automated and interactive theorem proving. Both have been used in a number of projects, such as: to prove the safety and liveness of self-$\star$ systems [AMS14]; and to prove the agreement and validity properties of the synchronous crash-tolerant Floodset consensus algorithm (introduced in [Lyn96]) [Bry11].

Actually, we are not the first ones that proved correctness of the SM protocol. Using Event-B, Krenicky and Ulbrich in [KU10] proved the validity and agreement properties of SM, as well as of ZA [GLR95]. In order to do so, they had to construct 4 contexts (which isolate the parameters of a formal model and their properties, and are assumed to hold for all instances) and 12 machines (which contain definitions of events and invariants which describe the discrete state transition system), as well as 106 invariants. Although they automatically proved some of these invariants using Rodin, the majority was proved manually. Besides the fact that they used a different approach, the authors assume that messages cannot be forged (in PBFT, at most f nodes can forge messages), and they do not verify implementations of these algorithms.

---

[5] Significant progress in the development of the correct-by-construction translation from Event-B to Java, C, C++, C#, Dafny, VHDL has been made in the following project: `http://wiki.event-b.org/index.php/Code_Generation_Activity`

2.3.1.3 Heard-Of model

The Heard-Of (HO) model [CS09]; [Bie+07] requires processes to execute in lock-step through rounds into which the distributed algorithms are divided. For each round r and each process p, the $\mathrm{HO}(p, r)$ set denotes the collection of processes that p has heard from in round r. Systems are modeled as pairs of an algorithm and a communication predicate over the $\mathrm{HO}(p, r)$ sets. This predicate captures both the system (e.g., degree of synchrony) and fault models of the system. For example, if a process crashes, it will never be heard from it anymore. The HO-model was implemented in Isabelle/HOL [CDM11] and used, for example, to verify the Byzantine agreement algorithm for synchronous systems with reliable links called EIGByz [Bar+92]. This formalization uses the notion of *global state of the system* [CL85], while our approach relies on Lamport's *happened before* relation [Lam78], which does not require reasoning about a distributed system as a single entity. Model checking and the HO-model were also used in [TS07]; [TS08]; [CCM09] for verifying the crash fault-tolerant consensus algorithms presented in [CS09]. To the best of our knowledge, there is no tool that allows generating code from algorithms specified using the HO-model.

2.3.1.4 IOA

IOA [GL00]; [Gar+04]; [Tau04]; [Geo+09] is the model used by Castro [Cas01] to prove PBFT's safety. It is a programming/specification language for describing asynchronous distributed systems as I/O automata [LT87] (a form of labeled state transition systems) and stating their properties. While IOA is state-based, the logic we use in this thesis is event-based. IOA can interact with a large range of tools such as type checkers, simulators, model checkers, theorem provers, and there is support for synthesis of Java code [Tau04]. In contrast, our methodology allows us to both implement and verify protocols within the same tool, namely Coq.

2.3.1.5 TLA$^+$

TLA$^+$ [Lam04]; [Cha+10] is a language for specifying and reasoning about systems. It combines: (1) TLA [Lam94], which is a temporal logic for describing systems [Lam04], and (2) set theory, to specify data structures. TLAPS [Cha+10] uses a collection of theorem provers, proof assistants, SMT solvers, and decision procedures to mechanically check TLA proofs. Model checker integration helps catch errors before verification attempts. TLA$^+$ has been used in a large number of projects (e.g., [LMW11]; [Jos+03]; [BDH07]; [New14]; [New+15]; [CLS16]) including in proofs of safety and liveness of Multi-Paxos [CLS16], and safety of a variant of an abstract model of PBFT [18] that does not include garbage collection. Moreover, Lamport and Merz in [LM94], using TLA+ and hierarchical proofs [Lam95], proved the safety of the seminal synchronous Byzantine algorithm OM [LSP82], in the presence of at most one traitor. To the best of our knowledge, TLA$^+$ does not perform program synthesis.

2.3.2 Tools

2.3.2.1 Actor Services

Actor Services [SM16] allows verifying distributed and functional properties of programs communicating via asynchronous message passing at the level of the source code (they use a simple Java-like language). It supports modular reasoning and proving liveness properties. To the best of our knowledge, Actor Services do not deal with faults.

2.3.2.2 Aneris

Aneris [Kro+18]; [Kro18] is a higher-order, concurrent, separation logic built in Coq, whose main goal is to facilitate modular verification of large software systems, including distributed systems. It allows *node-local reasoning* about concurrent distributed systems, i.e., when proving correctness of Aneris programs, one can reason about each node of the system in isolation. Among other things, the authors used Aneris to verify an implementation of the *two-phase-commit* protocol, such that a distributed client of the two-phase-commit provides a replicated logging.

2.3.2.3 ByMC

ByMC is a model checker for verifying the safety and liveness of fault-tolerant distributed algorithms [KVW15]; [Kon+17]; [KVW17]. It applies an automated method for model checking parametrized threshold-guarded distributed algorithms (e.g., processes waiting for messages from a majority of distinct senders). Distributed systems are represented as threshold automata that capture their local control flow. To enable automated verification, all cycles must be *simple* (i.e., there exists exactly one *node-disjoint* directed path[6] between any two locations in a cycle). Threshold automata are then interpreted as *counter systems* where instead of modeling the state of a system as the collection of the current states of the processes as in [CL85], a system is modeled by the number of processes in a given state. Specifications are expressed in a fragment of linear temporal logic, which the authors call *Fault-Tolerant Temporal Logic* (ELTL$_{\mathsf{FT}}$). Again, this restriction to a fragment of LTL enables automated verification. To prove that a system (represented as a threshold automaton) is safe and live, one specifies the negation of

---

[6] Two paths $p1$ and $p2$ are *node-disjoint*, if there does not exist a node i which can be found in both paths $p1$ and $p2$.

the safety and liveness of the system in $\mathrm{ELTL}_{\mathsf{FT}}$, and then ByMC checks whether there are parameters, an initial configuration, and an infinite schedule such that the specification is true about the system; in which case a counterexample is reported to the user. Otherwise, if no counterexample is found, this shows that the system is correct for all parameters that satisfy the required resilience condition (for BFT protocols, the resilience condition is $3f + 1$ processes). ByMC is based on a short counterexample property, which says that if a distributed algorithm violates a temporal specification then there is a counterexample whose length is bounded and independent of the parameters (e.g. the number of tolerated faults). The same overall technique is used to prove both safety and liveness properties. Namely, it relies on the fact that counterexamples all have a lasso shape, i.e., they loop after a finite prefix. In the case of liveness, counterexamples are infinite executions. However, thanks to the short counterexample property, for each such infinite execution, there exists a *representative* execution that has a bounded length.

In [Laz+17], the authors extended ByMC, with a distributed algorithm synthesizer, that works as follows: it takes as inputs (1) a sketch of a distributed algorithm with holes (threshold expressions that are left unspecified), (2) safety and liveness specifications, and (3) a resilience condition relating the total number of participants (a parameter) and the maximum number of faulty participants (also a parameter); and either outputs a correct distributed algorithms where those holes are filled in, or it reports that no such algorithm exists.

In [Sto+19], the authors introduced a variant of the threshold automata presented in [KVW17], which supports synchronous protocols (in the *rendez-vous* sense as opposed to assuming a maximum transmission delay). Those automata are called synchronous threshold automata, or STA for short. Because it turns out that in general the parameterized reachability problem is undecidable for STA, the authors show that many synchronous fault-tolerant algorithms have a bounded diameter (the number of steps needed to reach every configuration of a counter system) and they use bounded model checking to formally verify those algorithms. Moreover, they introduce an SMT-based procedure for finding the diameter of a counter system associated with an STA. As a case study, they formally verified several Byzantine fault-tolerant [ST87]; [BSW11]; [BGP89]; [P B89], crash fault-tolerant [Lyn96]; [Cha+00]; [Ray10] and omission fault-tolerant algorithms [BSW11]; [Ray10].

2.3.2.4 Chapar

Chapar [LBC16] is a framework for modular certification of causal consistency of replicated key-value store implementations and their client programs. The framework is written in Coq, and therefore allows extracting OCaml code (which implies that there is no gap between the verified and the executed code). Moreover, Chapar contains a simple model checker, which can be used to formally verify client applications. Using Chapar, the authors proved the causal consistency of two key-value stores. As opposed to this thesis, Chapar relies on the distributed snapshot semantics of distributed systems, and is specifically tailored to reason about causal consistency (as opposed to the strong linearizability consistency property of BFT-SMR protocols).

2.3.2.5 ConsL

ConsL [MSB17] is a language for expressing crash-fault tolerant asynchronous and partially synchronous consensus algorithms, whose semantics is expressed in HO, and that connects to the Spin model checker [Hol04]. As for ByMC, it relies on guards. The authors proved cutoff bounds that reduce the parameterized verification of consensus algorithms to a guard-depending number of processes. Using ConsL, the authors proved the *zero-one principle* (the algorithm's correctness for binary inputs entails the algorithm's correctness for inputs from any ordered set) and provide a cutoff bounds for the following algorithms: Paxos [Lam98], Ben-Or [Ben83], $\frac{1}{3}$-rule and three algorithms form the Uniform Voting family [CS09], as well as algorithm from [MSB15]. Because the bounds they obtained were small, the authors could leverage model checking to provide the first fully automated decision procedure applicable to a range of consensus algorithms.

2.3.2.6 DISEL

DISEL [WST17]; [SWT18] is a framework for modular verification of implementations of crash fault tolerant distributed systems. It provides a programming language shallowly embedded in Coq, and a separation-style program logic [Rey02] to compositionally reason about distributed systems. Unlike other state-of-the-art tools, DISEL provides supports for reusing correctness proofs of distributed programs, to derive the correctness of larger composite systems. Distributed systems are often designed as the composition of building blocks, where a block (a distributed program) acts as a server that provides a service. Other blocks then act as clients of that server to achieve higher-purpose goals (clients of servers then become servers of other clients). To achieve modular verification of such systems, DISEL provides a domain-specific language for the specification and implementation of distributed systems as the compositions of small building blocks; as well as novel proof techniques. To the best of our knowledge, DISEL does not allow reasoning about components relying on different system and fault models.

DISEL provides a specification language to specify the high-level behavior of systems abstracting away from low-level details. Such high-level specifications are called *protocols*. Protocols can then be refined into running implementations. One advantage of having such a two-layered language is that whenever proving the correctness of a composite system, one can reason about the high-level protocols of its components instead of their low-level implementations. Protocols are state transition systems, essentially composed of transitions to send messages and transitions to receive messages.

Once the correctness of a component has been proved, it is however sometimes not enough to prove the correctness of a larger composite system that uses this component. To overcome this issue, DISEL provides two techniques that enable modular verification of large composite systems: the WITHINV inference rule and *send-hooks*. The WITHINV rule allows adding an invariant to a protocol, thereby adding additional pre/post-conditions to that protocol, that can then be used to verify the composite system in which the protocol is used. Besides this WITHINV rule, DISEL also provides a standard separation logic-like frame rule, which can be used to compositionally reason about components that do not depend on each other. In addition, DISEL provides the notion of *hooks*, which are application-dependent restrictions on the possible interactions between the components of a composite system. Using DISEL, the authors have verified the two phase commit protocol, and showed how it can be reused to prove the correctness of a replicated log. Unlike our model, DISEL' model relies on the notion of global states and small step transitions (i.e., the distributed snapshot semantics).

2.3.2.7 EventML

EventML [BCR12]; [Rah+15]; [Rah+17] is a domain specific language implemented on top of the Nuprl theorem prover [Con+86]. It provides expressive and modular combinators for implementing and reasoning about crash-fault tolerant distributed systems. EventML implements both a Logic Of Events (LoE) to specify and reason about distributed systems; as well as a general process model to execute those specifications (GPM). Using EventML, the authors proved among other things the safety of Multi-Paxos [Rah+12]; [Sch+14]. Moreover, they built within Nuprl a tool that can automatically optimize GPM programs and prove that the optimized and non-optimized programs are bisimilar [RBA13]. As explained in Section 3.1, the model presented in this thesis evolved from EventML to reason about Byzantine fault-tolerant homogeneous/hybrid protocols and to reason about distributed knowledge.

2.3.2.8 IronFleet

IronFleet [Haw+15]; [Haw+17] is a framework for building and reasoning about distributed systems using Dafny [Lei10] and the Z3 theorem prover [MB08]. Because systems are both implemented in and verified using Dafny, IronFleet prevents gaps between running and verified code. It uses a combination of TLA-style state-machine refinements [Lam04] to reason about the distributed aspects of protocols, and Floyd-Hoare-style imperative verification techniques to reason about local behavior. The authors have implemented, among other things, the Paxos-based state machine replication library IronRSL, and verified its safety and liveness. Unlike IronFleet we focus on Byzantine faults.

2.3.2.9 Ivy

Ivy [Pad+16] initially supported debugging infinite-state systems using bounded verification, and verifying their safety by gradually building universally quantified inductive invariants.

In [Pad+17], the authors extended Ivy so that it can automatically verify safety properties of *models* of complex distributed protocols such as Paxos. Their method consists in transforming system models along with their invariants, expressed in an undecidable many-sorted first-order logic over uninterpreted structures, into models and invariants expressed in a decidable fragment of that logic (namely, EPR, which stands for Effectively Propositional Logic). Those transformations essentially consist in eliminating quantifier alternations, and are guided by the user (who provides derived relations to replace existential formulas). Those transformations are also mechanically checked.

Ivy was then further extended with the novel notion of *decidable decomposition* [Tau+18] allowing the tool to automatically verify the correctness of *implementations* of crash-fault tolerant distributed systems such as Raft and Multi-Paxos. The idea of decidable decomposition is that systems, models and proofs should be structured in a modular way to allow Ivy to use different decidable logics.

In [Pad+18], the authors extended Ivy further so that it can also be used to prove liveness properties of infinite-state systems by reducing those to safety properties (in some cases, liveness can even be proved automatically by combining this method with the one presented in [Pad+17]). Their method relies on a sound fair cycle detection mechanism (where a fair cycle is an execution that revisits a state after satisfying all fairness constraints—e.g., about fair scheduling). Liveness follows from the absence of fair cycles. Using this method, the authors proved the liveness of several protocols including several variants of Paxos. This method is related to the short counter-example method of ByMC [Kon+17].

Finally, in [Ber+19], the authors extended Ivy, so that it can automatically verify the correctness of threshold-based protocols by encoding these properties in decidable logics, namely EPR and BAPA. Ivy uses EPR to verify properties of distributed protocols while assuming some properties about sets whose cardinalities adhere to certain threshold, and then uses BAPA to verify if those properties are correct. They used their methodology to verify the safety and liveness of a Byzantine one-step consensus protocol and of a hybrid reliable broadcast protocol; as well as the safety of the Fast Byzantine Paxos protocol. The *hybrid fault model* used by Ivy has a different meaning from the one used in rest of the thesis. Namely, *hybrid fault model* [ST87] in their case distinguishes between asymmetric, symmetric and benign faults, while *hybrid fault model* used in the rest of this thesis assumes that distributed system is composed of components with different failure assumptions.

Most recently, Ivy in combination with Isabelle/HOL, is used for verifying safety and liveness of an open membership Byzantine agreement protocol on which a blockchain-based network called Stellar depends on [Lok+19].

2.3.2.10 ModP

ModP [Des+18] is a programming framework to build, specify and compositionally test dynamic, asynchronous distributed systems. Using ModP, one defines a system as a collection of interacting modules, where a module contains a number of state machines that react to and produce events (messages). State machines implement abstract interfaces, and are allowed to use other machines possibly from different modules through their interfaces. One can also dynamically declare new interfaces and machines. ModP provides frame rules and conditions to allow combining modules. Those conditions for example require that the events of two modules that one wants to combine are “compatible” (e.g., sent events must be disjoint). ModP provides means to ensure this, for example allowing hiding internal events, and renaming interfaces. In addition, ModP provides means to define and show that some modules are refinements of other modules. Using their framework, the authors implemented modularly and validated (through testing) two crash fault-tolerant distributed systems (Multi-Paxos [Lam98]; [RA15] and Chain Replication [RS04]).

2.3.2.11 PSync

PSync [DHZ16] is a domain specific language embedded in Scala that enables executing and verifying fault-tolerant distributed algorithms in synchronous and partially asynchronous networks. PSync is based on the HO-model, and has been used to implement several crash fault-tolerant algorithms. PSync makes use of a notion of global state and supports reasoning based on the multi-sorted first-order *Consensus verification logic* (CL) [Dra+14]. To prove safety, users have to provide invariants, which CL automatically checks for validity. Unlike PSync we focus on Byzantine faults.

2.3.2.12 PVS

PVS is an environment for formal specification and verification. It has been extensively used for verification of synchronous systems that tolerate malicious faults such as in [LR93b]; [LR93a]; [SWR02]; [Rus01], to the extent that its design was influenced by these verification efforts [Owr+95]. In [Rus03], using predecessor of PVS called EHDM [RHO91], Rushby formally verified OM algorithm, which was initially introduced in [LSP82]. Then, Lincoln and Rushby in [LR93b]; [LR93a], using PVS, formally verified version of OM algorithm that distinguishes between manifest, symmetric and arbitrary faults. Finally in [SWR02], the authors formally verified modified version of OM algorithm that distinguishes between manifest, omission, symmetric and arbitrary faulty nodes, as well as between send link faults and receive link faults.

2.3.2.13 Verdi

Verdi [Wil+15]; [Woo+16] is a framework to develop and reason about crash-fault tolerant distributed systems using Coq. As in our framework, Verdi leaves no gaps between verified and running code. Instead, OCaml code is extracted directly from the verified Coq implementation. To reason about faults the authors implemented several network semantics (single-node, reordering, duplicate, dropping and node failure semantics) that can be combined. Verdi provides a compositional way of specifying distributed systems, where the user provides an implementation of a system in an idealized model (in which faults cannot occur) and then applies one of the *verified system transformers* in order to obtain a system that tolerates faults. Verdi supports two types of transformers: (1) *transmission transformers*—used for networks in which packets can be duplicated, reordered or dropped; and (2) *replication transformers*—used in networks in which nodes may crash. For example, Raft [OO14]—an alternative to Paxos—transforms a distributed system into a crash-fault tolerant one. Moreover, in [Woo+16], the authors report that they have completely verified Raft's safety. They also present their methodology to deal with proof maintenance, such as using interface lemmas in order to hide definitions [Woo+16, Section 4]. One difference to our respective method is that they verify a system by reasoning about the evolution of its global state, while we use Lamport's happened before relation. Moreover, to the best of our knowledge, they do not deal with the full spectrum of arbitrary faults (e.g., malicious faults), they do not support reasoning about synchronous protocols.

2.3.2.14 Sally

Sally [DJN18] is a model checker, based on SMT and bounded model checking. It implements PD-KIND algorithm [JD16] that can automatically discover

k-inductive strengthening of a property. Using Sally, the authors specified and automatically verified several synchronous fault-tolerant protocols, including OM [LSP82], which is a Byzantine fault-tolerant protocol. In case of OM protocol, the authors proved agreement and validity assuming existence of one Byzantine-faulty link. Unfortunately, as stressed in [Tau+18], formal verification of complex distributed systems, such as Raft and Multi-Paxos, are beyond the reach of this technique.

2.3.3 Formal Verification of Trusted Components

In addition to the logics, models, and tools mentioned in the sections above, there are many more systems, tools, and techniques related to our work on hybrid systems. We mention some of those below.

2.3.3.1 Trustworthy Component-Based Programs

In our work we assume that trusted local components cannot be compromised, and derive distributed properties from the properties of these components. Orthogonal but complementary to our work are those works aiming at guaranteeing the trustworthiness of component-based local programs. Let us mention here a few relevant projects.

CAmkES [Kuz+07]; [Fer+13b]; [Fer+13a]; [Fer16]; [Kle+18] is a component based platform to reason about embedded systems built on top of seL4 [Kle+09]. It allows reasoning about the interface between trusted/non-trusted components and the payload network [Fer16]. It supports compositional programming and verification, and automatically generates verified "glue" code to connect the different components of a system.

SCC/RSCC [Jug+16]; [Aba+18] are secure compartmentalizing compilation techniques for unsafe languages such as C. Applications are divided into components that communicate via procedure calls, and the compiler ensures that compromised components cannot contaminate the other components.

2.3.3.2 Interfacing With Trusted Components

Orthogonal but related to our work, many models, systems, and tools have been developed to provide safe and secure interfaces between trusted components and payload systems. Given the fact that IBM's CCA API is a standard API used by banks, many researchers have focused on studying whether it is secure [Kei06]; [Tog13]; [You+07]; [You+05]; [CM06].

Many other generic model checking-based bug finding tools have been develop to ensure that APIs are secure, such as [Gan+05]; [CW02]; [Avg14].

Moreover, temporal rules are a standard technique to ensure that clients can only use APIs in a safe manner [Alu+05]; [BLR11].

2.3.3.3 Trusted Component/Environment Verification

Several new trusted environments have been developed this past decade, such as [19l]; [19i]; [19m]. As a result, many papers [Haw+14]; [Sin+15]; [BTZ17]; [Fer+17a]; [Dat+09]; [LCF15]; [Fer+17b]; [Dat+09]; [Jia+15]; [Sub+17]; [SQF18]; [Sha+15]; [Bai+14]; [Del+10]; [Gua15]; [Del+11]; [BJX17]; [XBM13], to mention only a few, are concentrated on proving different properties about these trusted components/environments (e.g. confidentiality, integrity, linearizability, remote equivalence). Although some of these papers were about proving properties of the security protocols, e.g. [Del+11]; [BJX17]; [XBM13], to the best of our knowledge none of them is about proving properties of BFT-SMR protocol.

2.3.4 Knowledge and Distributed Systems

Knowledge is a widely used and studied concept. For example, knowledge based systems have been developed to: (1) analyze distributed systems [Hal87]; [HM90]; [DM90]; [PT92]; [Fag+97]; (2) reason about synchronous systems [Ben11]; [BM14]; [CGM14]; [CGM16]; [DMM17]; [GM18]; (3) study the *principle of full communication* [Roe07]; [ÅW17] (i.e., group members should be able to establish group knowledge through communication); (4) derive protocols [HZ92]; (5) synthesize systems [Bic+04]; (6) reason about blockchain protocols [HP17]; and (8) reason about distributed knowledge using proof systems [HN07]; [Gie14]; [GA19] and type theories [PT12].

The closest work to ours on the verification of synchronous systems using knowledge theory is the one by Ben-Zvi and Moses [BM10]. They define the concept of *Syncausality*, a generalization of Lamport's *happened before* relation [Lam78], which is designed specifically for synchronous systems. In addition, they identify two general communication structures called *centipede* and *centibrooms* that exist in synchronous systems.

Another work close to ours on the verification of synchronous systems using knowledge theory is done by Berman, Garay and Perry [BGP89]. The authors proved agreement and validity of a variation of the Byzantine generals algorithm introduced in [Bar+92]; [LSP82], which they called EIG (Exponential Information Gathering). The authors also proved agreement and validity of ESFM (Early Stopping Fault Masking) protocol, which is actually EIG protocol equipped with *fault masking* [Bar+92]; [Coa86] and *early stopping* [DRS90]. Thanks to these mechanisms, ESFM allows correct processors to reach agreement using less communication steps than using EIG. On top of this, [BGP89] presents two novel protocols, *Cloture Voting* and *Phase King protocol*, which are more optimal than its predecessors. Using knowledge theory, the authors proved agreement and validity of Cloture Voting, and showed that Phase King protocol, solves Byzantine generals problem in $3(f+1)$ rounds, using messages from $\{0, 1, 2\}$ and assuming $n > 3f$.

One major difference between our work and works presented in [BM10] and [BGP89] is that we our proofs are formally verified using theorem prover.

2.3.5 Modeling Byzantine behavior

As mentioned above, we are not the first ones to reason about Byzantine fault-tolerant systems (see Fig. 2.4). In this section we briefly discuss how asynchronous Byzantine behavior is modeled in ByMC and TLA$^+$, as well as how synchronous Byzantine behavior is modeled in Event-B, HO and PVS.

ByMC extends the original Control Flow Automata (CFA), by introducing "a where cond", which states that action a will be executed only if some condition *cond* is meet. This extension allowed reasoning about Byzantine fault-tolerant algorithms, because these protocols only react to messages that are sent by quorums of processes. Additionally, ByMC captures Byzantine behavior as sending messages when is not needed, i.e., when a process does not follow its specification. Unlike our model, the ByMC model [Joh+12] assumes that processes cannot impersonate each other, and that processes communicate reliably.

TLA$^+$. In [Lam11] Lamport presented the idea of *Byzantizing* which enables converting an algorithm with N processes that tolerates crash failure of up to f processes into an algorithm with $N + f$ processes that assumes existence of N correct processes and tolerates up to f Byzantine processes. In order to achieve this, Lamport introduced concepts of *acceptors*, which are equivalent to PBFT replicas, and assumes that up to f acceptors are *fake*, while all the other ones are *real* ones (i.e., non-faulty ones). Additionally, Lamport assumes that one might determine in advance which processes are malicious—this assumption results in no loss of generality, because Byzantine algorithms do not assume any knowledge about which acceptors are real, and which are the fake ones. Moreover, the union of fake and real acceptors is called *byzacceptors*, and their quorums are called *byzquorums*. A set of byzacceptors is built such that: (1) if a quorum consists of any q acceptors, a byzquorum consists of any $q + f$ byzaceptors; and (2) any two byzquorums have a real acceptor in common.

Event-B. As mentioned above, Event-B [Hal08] relies on a refinement technique, which assumes that one starts with a simple abstract model and then introduces details gradually. By dividing a model into several levels of refinements, one can concentrate on a particular aspect of the model instead of dealing with the entire complexity all the time.

Each Event-B model consists of two constructs: (1) *context*—contains types, constants and axioms; and (2) *machines*—contain variables (i.e., state of a machine), invariants (i.e., constrains on variables), theorems (i.e., predicates implied by invariants), events and variants. Similar to our approach, events in Event-B are composed of guards (which describe the necessary condition under which an event might occur) and actions (which describe how state variables evolve when the event occurs). The properties which one wants to prove using Event-B, are formulated as invariants in the machines.

In [KU10] the authors start with a simple machine called *Messages*. This machine records all messages sent in the current round, a current round number, as well as all values that a machine received so far. It assumes that every message contains sender, receiver and a value. Also, it contains two types of events: initial event and round event. Then, the authors refined the machine *Messages* to a machine called *MessagesSigned*. The *MessagesSigned* machine adds additional assumption that only messages which are received can be forwarded. Next, the authors refine machine *MessagesSigned* to a machine called *Histrory*. This machine adds additional assumption that each message caries history (i.e., each message contains a set of machines trough which it has passed). The authors repeat refinement process several times, until they reach a machine called *SM*, which actually models SM algorithm [LSP82].[7] Unfortunately, in most cases invariants have to be adapted and proofs have to be redone.

Because SM is not a "typical" round-based synchronous recursive algorithm, but it is a form of reactive system (i.e., when a lieutenant receives a message, he processes it and then disseminates messages to other participants), their SM machine is similar to our notion of replica. Namely, each SM machine contains a history (i.e., it keeps track of all the messages it received) and input buffer (in which it maintains all messages which should be processed), and it processes one message at a time. Unlike our model, SM machine handles three types of events: the initial event, an event which assumes that current machine is non-faulty, and an event which assumes that the current event drops all messages. Also unlike our model, the machines can be *non-faulty*, *symmetrically faulty* and *arbitrarily faulty*.

Interesting fact is that SM is the only machine for which all proofs were simple enough to be proved automatically by the Rodin's solvers [Abr+10].

HO. As explained above, degree of synchronism and the failure model are formally expressed by a *communication predicate*. For example, a communication predicate might state that no process receives more than α corrupted messages in any round, but that every process receives more than β correct messages at each round. This type of communication predicate will be true for protocols modeled using HO,

---

[7] We refer the readers interested in all refinement steps to the original paper [KU10].

because HO model assumes that all nodes are connected using point-to-point links. Usually both, local and global communication predicates, are defined. While local communication predicate is valid only during one round and is used to prove safety, the global communication predicate is valid only when all rounds are taken into account and is used to prove termination.

HO model is based on a communication-closed rounds, i.e., in each round, every process: (1) sends messages to other processes; (2) receives messages from other processes; and (3) makes a local state transition. In this model, all messages which are not received within that round are discarded. Thanks to a communication predicate, even in a case when a node is lying, HO model makes no assumptions on the reason why a node did not receive a message which was sent by some other node—because the Byzantine nodes will never form a majority (there can be up to f of them), a correct node will always decide upon a correct value.

As it turns out, HO model is not designed to deal with full spectrum of Byzantine failures. Although it is designed to tolerate value faults (i.e., at any round r, the message received by process q from p might be different from the message that p actually sent to q), it cannot tolerate Byzantine transition faults (i.e., processes never deviate from their specification). Because of this, their agreement property is simpler than the one we proved. In their case, if the process decides upon some value, the value it decided upon is always the right one. In our case, we have to assume that the process is not faulty, and only then we can make this kind of claims.

In [CDM11], the authors formally verified two asynchronous Byzantine fault-tolerant algorithms $A_{T,E,\alpha}$ and $U_{T,E,\alpha}$ originally introduced in [Bie+07], as well as synchronous Byzantine fault-tolerant algorithm EIGByz, originally introduced in [Bar+92]. While $A_{T,E,\alpha}$ and $U_{T,E,\alpha}$ are correct under local communication predicate, EIGByz requires validity of its global predicate, which only depends on SHO sets (i.e, messages which a node actually received). The authors pointed out that global predicates, which only rely on SHO sets, correspond to *synchronous approach*. Because EIGByz is designed for synchronous systems with reliable links, the authors left reasoning about authentication for future work.

PVS. Because PVS is a theorem prover, formal specification and verification of Byzantine faults using PVS [LR93b]; [LR93a]; [SWR02]; [Rus01] is quite similar to the formal specification and verification of Byzantine faults presented in this thesis. The main difference between our work and works presented in [LR93b]; [LR93a]; [SWR02]; [Rus01] is that our fault models slightly differ—while we only consider arbitrary faults, they distinguish between several less-than arbitrary fault modes (i.e., the model presented in [LR93b]; [LR93a] distinguishes between manifest, symmetric and arbitrary faults, and the model presented in [SWR02]; [Rus01] distinguishes between manifest, omission, symmetric and arbitrary faulty nodes, as well as between send link faults and receive link faults). As a consequence of this, our reasoning also slightly differs—when proving properties of distributed systems we only consider values coming from correct nodes, and in their proofs they have to consider and combine values coming from nodes which are non-arbitrary faulty.

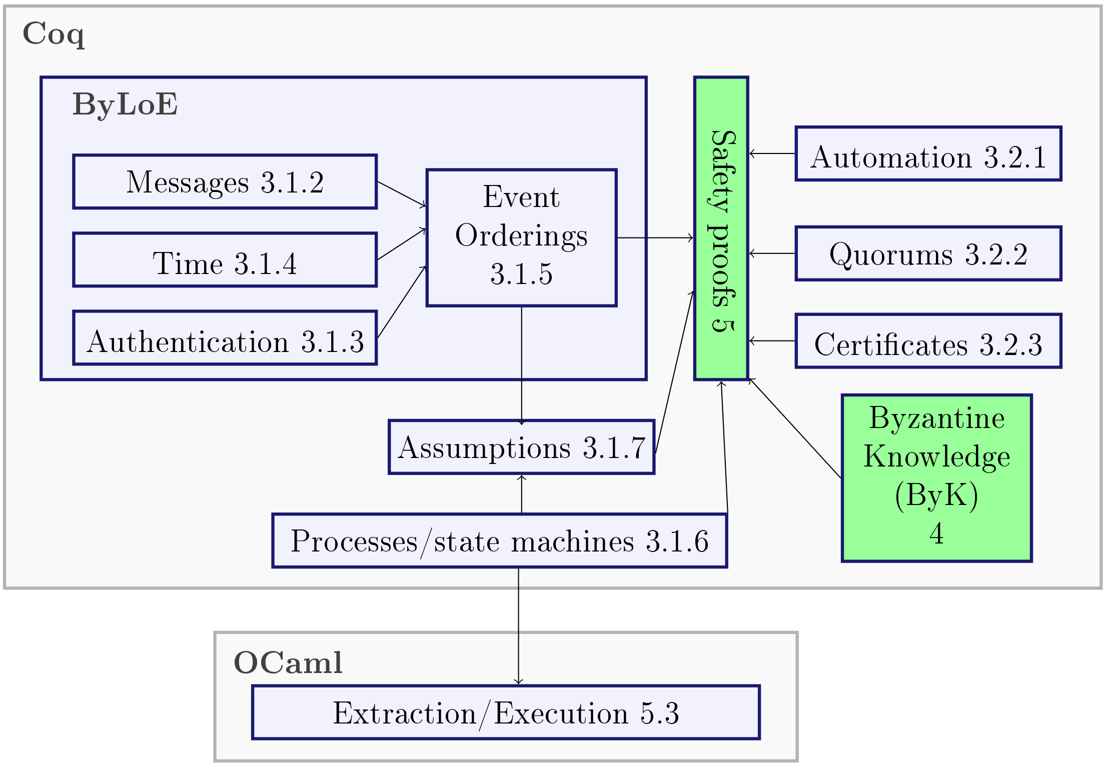


Figure 3.1: Outline of our Byzantine Logic of Events (ByLoE) model

Chapter 3

Verification of Homogeneous BFT protocols

Using PBFT (see Section 2.1.2) as a running example, we now present our Coq model for Byzantine fault-tolerant distributed systems communicating via message passing, which relies on a logic of events. See Figure 3.1 for an outline of our formalization.

3.1 Homogeneous Model

3.1.1 ByLoE: A Byzantine Logic of Events

There are two main models of distributed computing that are prominently used in the literature. Namely, Lamport's *happened before relation* [Lam78], and Chandy and Lamport's *global state semantics* [CL85]. In the first model, a distributed system is modeled as a collection of events happening at the various locations involved in the system and connected by a *happened before relation*, which essentially defines a causal partial order between events. In the second model, a distributed system is modeled as a single state machine: a state is the collection of all processes at a given time, and a transition takes a message in flight and delivers it to its recipient (a process in the collection). Each of these two models has advantages and disadvantages over the other. We chose to base our model on Lamport's happened before relation because in our experience it corresponds more closely to the way distributed system researchers and developers reason about protocols. As such, it provides a convenient communication medium between distributed systems and verification experts.

Our model evolved from the Logic of Events (LoE) used in EventML [Bic09]; [BCR12]; [Rah+17] to not only deal with crash faults, but arbitrary faults in general (including malicious faults). LoE, which is built on top of Lamport's happened before relation [Lam78], and is also related to event structures [NPW81]; [Mat89], was developed to reason about events occurring in the execution of a distributed system, with a particular focus on systems that can tolerate crashes. It has recently been used to verify consensus protocols [Sch+14]; [Rah+17] and cyber-physical systems [AK15].

We call the logic of events we present here, ByLoE, which stands for the Byzantine Logic of Events. Unlike in LoE, where an event is an abstract entity that corresponds to the handling of a received message, in ByLoE, an event is an abstract entity that corresponds either (1) to the handling of a received message, or (2) to some arbitrary activity about which no information is provided (see the discussion about trigger in Section 3.1.5). We use those arbitrary events to model arbitrary/Byzantine faults. An event happens at a specific point in space/time: the space coordinate of an event is called its location, and the time coordinate is given by a well founded partial ordering on events that totally orders all events at the same location. As pointed out in [TG98], we model logical time and not potential causality because local events are totally ordered. Although a timestamp is associated to each event, we use them only when we reason about synchronous protocols, and they are meaningless otherwise. Processes react to the messages that triggered the events happening at their locations one at a time, by updating their states and creating messages to send out, which in turn might trigger other events.

To reason about distributed systems, we use the notion of *event orderings* (see Section 3.1.5), which essentially are collections of ordered events and represent runs of a system. They are abstract entities that are never instantiated. Rather, when proving a property about a distributed system, one has to prove that the property holds for all event orderings corresponding to all possible runs of the system (see Section 3.1.6 and Chapter 5 for examples). Some runs/event orderings are not possible and therefore excluded through assumptions, such as the ones described in Section 3.1.7. For example, AXIOM_exists_at_most_f_faulty excludes event orderings where more than f out of n nodes could be faulty.

In the next few sections, we explain the different components (messages, authentication, time, event orderings, state machines, and correct traces) of ByLoE, and their use in our PBFT case study. Those components are parameterized by abstract types (parameters include the type of messages and the kind of authentication schemes), which we later have to instantiate in order to reason about a given protocol, e.g. PBFT, and to obtain running code. The choices we made when designing ByLoE were driven by our goal to generate running code. For example, we model cryptographic primitives to reason about authentication.

3.1.2 Messages

Model. As mentioned above, processes communicate via message passing. Correctly received messages of type msg, a parameter of our model, trigger events at the locations where the messages are received. Processes react to messages to produce message/destinations pairs called *directed messages*. A directed message is a triple composed of: (1) a message (of type dmMsg); (2) a list of recipients (of type dmDst); and (3) a message delay (of type dmDelay). For simplicity, we assume for now that messages are always sent with delay 0. A directed message is typically handled by a message outbox, which sends the message to the listed destinations.[1] A destination is the name (of type name, which is a parameter of our model) of a node participating in the protocol.

Case Study. In our PBFT implementation, we instantiate the msg type with

---

[1] Message inboxes/outboxes are part of the runtime environment but not part of the model.

the following datatype (we only show some of the normal-case operation messages, leaving out for example the more involved pre-prepare messages—see Section 2.1.2.1):

```
Inductive Bare_Prepare :=
| bare_prepare (v : View) (n : SeqNum) (d : digest) (i : Rep).

Inductive Prepare :=
| prepare (b : Bare_Prepare) (a : list Token).

Inductive PBFTmsg :=
| REQUEST (r : Request)
| PREPARE (p : Prepare)
| REPLY (r : Reply) ...
```

As for prepares, all messages are defined as follows: we first define bare messages that do not contain authentication tokens (see Section 3.1.3), and then authenticated messages are formed as pairs of a bare message and a list of authentication tokens. Views and sequence numbers are nats, while digests are parameters of the specification. PBFT involves two types of nodes: replicas of the form PBFTreplica(r), where r is of type Rep; and clients of the form PBFTclient(c), where c is of type Client. Both Rep and Client are parameters of our formalization, such that Rep is of arity 3f+1, where f is a parameter that stands for the maximum number of tolerated faults.

3.1.3 Authentication

Model. Our model relies on an abstract concept of keys, which we use to implement and reason about authenticated communication. Capturing authenticity at the level of keys allows us to talk about impersonation through key leakage. Keys are divided into *sending keys* (of type sending_key) to authenticate a message for a target node, and *receiving keys* (of type receiving_key) to check the validity of a received message. Both sending_key and receiving_key are parameters of our model.[2] Each node maintains *local keys* (of type local_keys), which consists of two lists of *directed keys*: one for sending keys and one for receiving keys. Directed keys are pairs of a key and a list of node names identifying the processes that the holder of the key can communicate with.

---

[2] Sending and receiving keys must be different when using asymmetric cryptography, and can be the same when using symmetric cryptography.

Sending keys are used to create *authentication tokens* of type Token, which we use to authenticate messages. Tokens are parameters of our model and abstract away from concrete concepts such as digital signatures or MACs. Typically, a message consists of some data plus some tokens that authenticates the data. Therefore, we introduce the following parameters: (1) the type data, for the kind of data that can be authenticated; (2) a create function to authenticate some data by generating authentication tokens using the sending keys; and (3) a verify function to verify the authenticity of some data by checking that it corresponds to some token using the receiving keys.

Once some data has been authenticated, it is typically sent over the network to other nodes, which in turn need to check the authenticity of the data. Typically, when a process sends an authenticated message to another process it includes its identity somewhere in the message. This identity is used to select the corresponding receiving key to check the authenticity of the data using verify. To extract this claimed identity we require users to provide a data_sender function.

It often happens in practice that a message contains more than one piece of authenticated data (e.g., in PBFT, pre-prepare messages contain authenticated client requests). Therefore, we require users to provide a get_contained_auth_data function that extracts all authenticated pieces of data contained in a message. Because we sometimes want to use different tokens to authenticate some data (e.g., when using MACs), an authenticated piece of data of type auth_data is defined as a pair of: (1) a piece of data, and (2) a list of tokens.

Case Study. Our PBFT implementation leaves keys and authentication tokens abstract because our safety proof is agnostic to the kinds of these elements. However, we turn them into actual asymmetric keys when extracting our Coq code to OCaml code (see Section 5.3 for more details). The create and verify functions are also left abstract until we extract the code to OCaml. Finally, we instantiate the data (the objects that can be authenticated, i.e., bare messages here), data_sender, and get_contained_auth_data parameters using:

```
Inductive PBFTdata :=
  | PBFTdata_request (r : Bare_Request)
  | PBFTdata_prepare (p : Bare_Prepare)
  | PBFTdata_reply (r : Bare_Reply) . . .

Definition PBFTdata_sender (m : data) : option name :=
match m with
  | PBFTdata_request (bare_request o t c) ⇒ Some (PBFTclient c)
  | PBFTdata_prepare (bare_prepare v n d i) ⇒ Some (PBFTreplica i)
  | PBFTdata_reply (bare_reply v t c i r) ⇒ Some (PBFTreplica i) . . .
```

```
Definition PBFTget_contained_auth_data (m : msg) : list auth_data :=
match m with
  | REQUEST (request b a) ⇒ [(PBFTdata_request b,a)]
  | PREPARE (prepare b a) ⇒ [(PBFTdata_prepare b,a)]
  | REPLY (reply b a) ⇒ [(PBFTdata_reply b,a)] ...
```

3.1.4 Time

As mentioned above, processes communicate by exchanging messages. The considered system model provides, among other things, a characterization of the possible such exchanges. For example, in asynchronous systems, messages can take an arbitrarily long time to arrive at their destinations, which does not impose any restriction whatsoever on message exchanges. On the other hand, synchronous systems, require among other things that there exists a maximum transmission delay by which all sent messages must be received by their recipients. Therefore, to support reasoning about such synchronous systems, we equipped our model with a notion of time, namely the type dt_T. This type is left abstract in ByLoE, and can be instantiated with any discrete representation of time (e.g. the natural numbers) that, among other things, forms a ring, and that supports transitive and irreflexive less than (dt_lt) and transitive and reflexive less or equal (dt_le) relations.[3]

ByLoE is not the first logic of events that supports reasoning about time. For example, Anand and Knepper [AK15] introduced such a time-based logic of events to reason about the cyber-physical behavior of robotics systems. One major difference between their work and ours is that they do not reason about faults.

3.1.5 Event Orderings

A typical way to reason about a distributed system is to reason about its possible runs, which are sometimes modeled as execution traces [RHB97], and which are captured in ByLoE using *event orderings*. An *event ordering* is an abstract representation of a run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by system designers (see for example Figure 6.1). As opposed to [RHB97], a trace here is not just one sequence of events but instead can be seen as a collection of local traces (one local trace per sequential process), where a local trace is a collection of events all happening at the same location and ordered in time, and such that some events of different local traces are causally ordered. Event orderings are never instantiated. Instead, we

---

[3] For more details about our abstract notion of time, we refer the interested reader to our implementation: `code/model/DTime.v`.

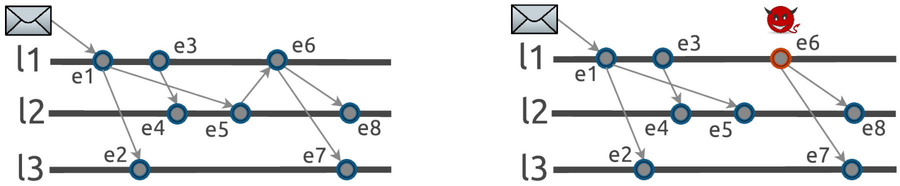


Figure 3.2: Examples of message sequence diagrams: LoE (left); ByLoE (right)

express system properties as predicates on event orderings. A system satisfies such a property if every possible execution of the system satisfies the predicate. We first formally define the components of an event ordering, and then present the axioms that these components have to satisfy.

3.1.5.1 Components

An event ordering is formally defined as the tuple:[4]

```
Class EO := {
    Event : Type;
    happenedBefore : Event → Event → Prop;
    loc : Event → name;
    pred : Event → option Event;
    trigger : Event → option msg;
    time : Event → PosDTime;
    keys : Event → local_keys; }
```

where (1) Event is an abstract type of events; (2) happenedBefore is an ordering relation on events; (3) loc returns the location at which events happen; (4) pred returns the direct local predecessor of an event when one exists, i.e., for all events except initial events; (5) given an event e, trigger either returns the message that triggered e, or it returns None to indicate that no information is available regarding the action that triggered the event (see below); (6) time returns the logical time at which events happened[5]; and (7) keys returns the keys a node can use at a given event to communicate with other nodes. The event orderings presented here are similar to the ones used in [AK15]; [Rah+17], which we adapted to handle Byzantine faults by modifying the type of trigger so that events can be triggered by arbitrary actions and not necessarily by the receipt of a message, and by adding support for authentication through keys. To model that at most f nodes out of

---

[4] A Coq type class is essentially a dependent record.

[5] Because when reasoning about distributed systems one only has to reason about positive time, we also introduced PosDTime which is essentially a subset of dt_T that contains non-negative values.

N can be faulty we use the AXIOM_exists_at_most_f_faulty assumption, which enforces that trigger returns None at most f nodes (see Section 3.1.7).

Moreover, even though non-syntactically valid messages do not trigger events because they are discarded by message boxes, a triggering message could be syntactically valid, but have an invalid signature. Therefore, it is up to the programmer to ensure that processes only react to messages with valid signatures using the verify function. Our AXIOM_authenticated_messages_were_sent_non_byz and AXIOM_exists_at_most_f_faulty assumptions presented in Section 3.1.7 are there to constrain trigger to ensure that at most f nodes out of N can diverge from their specifications, for example, by producing valid signatures even though they are not the nodes they claim to be (using leaked keys of other nodes).

3.1.5.2 Axioms

The following axioms characterize the behavior of these components:
1. Equality between events is decidable. Events are abstract entities that correspond to points in space/time that can be seen as pairs of numbers (one for the space coordinate and one for the time coordinate), for which equality is decidable.
2. The happened before relation is transitive and well founded. This allows us to prove properties by induction on causal time. We assume here that it is not possible to infinitely go back in time, i.e., there is a beginning of (causal) time, typically corresponding to the time a system started.
3. The direct predecessor e_2 of e_1 happens at the same location and before e_1. This makes local orderings sub-orderings of the happenedBefore ordering.
4. If an event e does not have a direct predecessor (i.e., e is an initial event) then there is no event happening locally before e.
5. The direct predecessor function is injective, i.e., two different events cannot have the same direct predecessor.
6. If an event e_1 happens locally before e_2 and e is the direct predecessor of e_2, then either $e = e_1$ or e_1 happens before e. From this, it follows that the direct predecessor function can give us the complete local history of an event.

Notation. The type $A \to B$ is the type of total functions, of the form $\lambda x.b$, from A to B. The type $A * B$ is the type of pairs of the form $\langle a, b \rangle$ of an $a \in A$ and a $b \in B$. We use the standard "let" notation to destruct pairs: `let` $x, y = p$ `in` f. We write p.1 and p.2 for the 1st and 2nd elements of the pair p. $\mathbb{B}$ is the Boolean type with constructors `true` and `false`. We often assume an implicit coercion from $\mathbb{B}$ to $\mathbb{P}$ (the type of propositions). The option(A) type is the usual option type with constructors `None` and `Some`(a), where $a \in A$. The list(A) type is the usual list type, with constructors []—the empty list—and $a :: l$, where $a \in A$ and $l \in$ list(A).

We use $a \prec b$ to stand for (happenedBefore a b); $a \preceq b$ to stand for ($a \prec b$ or $a = b$); and $a \sqsubseteq b$ to stand for ($a \preceq b$ and loc a=loc b). Moreover, let first?(e) be `true` iff pred(e) = `None`; let $e_1 \subset e_2$ be pred(e_2) = `Some`(e_1); let pred$^=$(e) be e' if $e' \subset e$, and e otherwise. We also sometimes write EO instead of EventOrdering.

Some functions take an event ordering as a parameter. For readability, we sometimes omit those when they can be inferred from the context. Similarly, we will often omit type declarations of the form (T : `Type`).

In order to improve readability, in Chapter D we present summary of notation introduced in this thesis.

3.1.5.3 Correct Behavior

To prove properties about distributed systems, one only reasons about processes that have a correct behavior. To do so we only reason about events in event orderings that are correct in the sense that they were triggered by some message:

```
Definition isCorrect (e : Event) :=
  match trigger e with
  | Some m ⇒ True
  | None ⇒ False
  end.

Definition arbitrary (e : Event) := ~ isCorrect e.
```

Next, we characterize correct replica histories as follows: (1) First, we say that an event e has a correct trace if all local events prior to e are correct. (2) Then, we say that a node i has a correct trace before some event e, not necessarily happening at i, if all events happening before e at i have a correct trace:

```
Definition has_correct_bounded_trace (e : Event) :=
  ∀ e', e' ⊑ e
  → isCorrect e'.

Definition has_correct_trace_before (e : Event) (i : name) :=
  ∀ e', e' ≼ e
  → loc e' = i
  → has_correct_bounded_trace e'.
```

3.1.6 Computational Model of ByLoE

Model. We now present our computational model, which we use when extracting OCaml programs. Unlike in EventML [Rah+17] where systems are first specified as *event observers* (abstract processes), and then later refined to executable code, we skip here event observers, and directly specify systems using executable state machines, which essentially consist of an update function and a current state. We define a system of distributed state machines as a function that maps names to state machines. Systems are parametrized by a function that associates state types with names in order to allow for different nodes to run different machines.

```
Definition Update S I O := S → I → (option S * O).

Record StateMachine S I O := MkSM { halted : bool; update : Update S I O; state : S }.

Definition System (F : node → Type) I O := ∀ (i : node), StateMachine (F i) I O.
```

where S is the type of the machine's state, I/O are the input/output types, and halted indicates whether the state machine is still running or not. We sometimes write System(F) for the type of systems that take messages as inputs and output directed messages. Moreover, we sometimes write System(S) for System(`fun` _ ⇒ S).

Let us now discuss how we relate state machines and events. Let $sm@^{-}e$ and $sm@^{+}e$ be the states of the sm state machine before and after the event e, respectively.[6] These states are computed by extracting the local history of events up to e using pred, and then updating the state machine by running it on the triggering messages of those events. These functions return None if some arbitrary event occurs or the machine halts sometime along the way. Otherwise they return Some s, where s is the state of the machine updated according to the events. Therefore, assuming they return Some amounts to assuming that all events prior to e are correct, i.e., we can prove that if $sm@^{+}e =$ Some s then has_correct_trace_before e (loc e). As illustrated below, we use these functions to adopt a Hoare-like reasoning style by stating pre/post-conditions on the state of a process prior and after some event. In addition, we also define $sm \rightsquigarrow e$ to be the outputs of the state machine sm at e. Those are computed by: (1) computing the state $sm@^{-}e$; and (2) running sm on that state and on the input that triggered e (in case of a "correct" event). We sometimes write $S@^{-}e$, where S is a system, for $sm@^{-}e$, where sm is the state machine running at loc e.

---

[6] Those two operations are called state_sm_before_event and state_sm_on_event in our implementation `code/model/Process.v`

Case Study. We implement PBFT replicas as state machines, which we derive from an update function that dispatches input messages to the corresponding handlers. Finally, we define PBFTsys as the function that associates PBFTsm with replicas and a halted machine with clients (because we do not reason here about clients).

```
Definition PBFTupdate (i : Rep) :=
  fun state msg ⇒ match msg with
                   | REQUEST r ⇒ PBFThandle_request i state r
                   | PREPARE p ⇒ PBFThandle_prepare i state p
                   ...
                   end.

Definition PBFTsm (i : Rep) := MkSM false (PBFTupdate i) (initial_state i).

Definition PBFTsys := PBFTsm.
```

Let us illustrate how we reason about state machines through a simple example that shows that they maintain a view that only increases over time. It shows a local property, while Chapter 5 presents the distributed agreement property that makes use of the assumptions presented in Section 3.1.7. As mentioned above we prove such properties for all possible event orderings, which means that they are true for all possible runs of the system. In this lemma, *s1* is the state prior to the event *e*, and *s2* is the state after handling *e*. It does not have pre-conditions, and its post-condition states that the view in *s1* is smaller than or equal to the view in *s2*.

```
Lemma current_view_increases : ∀ (eo : EO) (e : Event) i s1 s2,
    (sm i)@⁻ e = Some s1
    → (sm i)@⁺ e = Some s2
    → current_view s1 ≤ current_view s2.
```

3.1.7 ByLoE Assumptions

Before one starts reasoning about distributed system, it has to first define a system and threat fault model. In this section, we introduce both, system as well as fault model assumptions our framework supports. Moreover, because reasoning about Byzantine behavior requires use of cryptography, in this section we also introduce our assumptions regarding cryptography. The organization of this section is depicted in Figure 3.3.

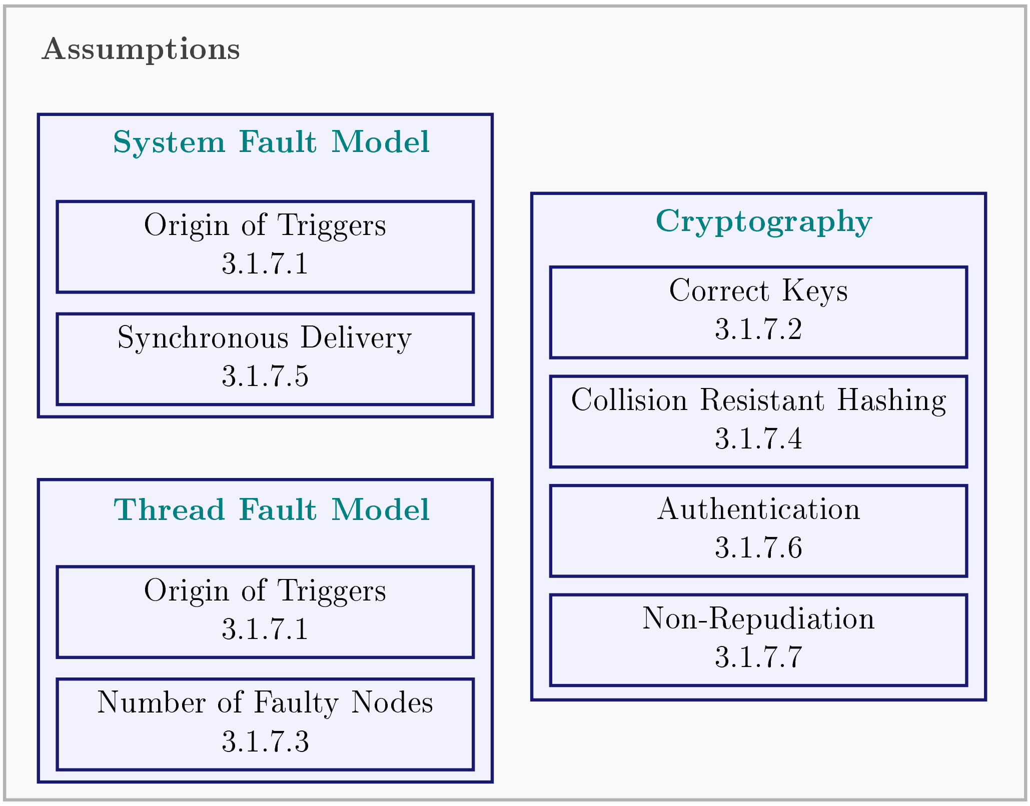

Assumptions
System Fault Model
Origin of Triggers 3.1.7.1
Synchronous Delivery 3.1.7.5
Thread Fault Model
Origin of Triggers 3.1.7.1
Number of Faulty Nodes 3.1.7.3
Cryptography
Correct Keys 3.1.7.2
Collision Resistant Hashing 3.1.7.4
Authentication 3.1.7.6
Non-Repudiation 3.1.7.7

Figure 3.3: Overview of ByLoE assumptions

3.1.7.1 Assumption 1 (Origin of Triggers)

Proving safety properties of crash fault-tolerant protocols that only require reasoning about past events, such as agreement, does not require reasoning about faults and faulty replicas. To prove such properties, one merely has to follow the causal chains of events back in time, and if a message is received by a node then it must have been sent by some node that had not crashed at that time. The state of affairs is different when dealing with Byzantine faults.

One issue is that Byzantine nodes can deviate from their specifications or impersonate other nodes. However, BFT protocols are designed in such a way that nodes only react to collections of messages, called *certificates* (see Section 3.2.3), that are larger than the number of faults. This means that there is always at least one correct node that can be used to track down causal chains of events.

A second issue is that, in general, we cannot assume that some received message was sent as such by the designated (correct) sender of the message because messages can be manipulated while in flight. As captured by the AX-

IOM_authenticated_messages_were_sent_or_byz predicate defined below,[7] we can only assume that the authenticated parts of the received message were actually sent by the designated senders, possibly inside larger messages, provided the senders did not leak their keys. As usual, we assume that attackers cannot break the cryptographic primitives, i.e., that they cannot authenticate messages without the proper keys [Cas01].

```
 1.Definition AXIOM_authenticated_messages_were_sent_or_byz (P : AbsProcess) :=
 2. ∀ e (a : auth_data),
 3.  In a (bind_op_list get_contained_auth_data (trigger e))
 4.  → verify_auth_data (loc e) a (keys e) = true
 5.  → ∃ e', e' ≺ e ∧ am_auth a = authenticate (am_data a) (keys e')
 6.    ∧ ( (∃ dst m,
 7.             In a (get_contained_auth_data m) ∧ In (m,dst) (P eo e')
 8.             ∧ data_sender (loc e) (am_data a) = Some (loc e'))
 9.           ∨
10.           (∃ e”,
11.              e” ⪯ e' ∧ arbitrary e' ∧ arbitrary e” ∧ got_key_for (loc e) (keys e”) (keys e')
12.              ∧ data_sender (loc e) (am_data a) = Some (loc e”)) ).
```

This assumption says that if the authenticated piece of data a is part of the message that triggered some event e (L.3), and a is verified (L.4), then there exists a prior event e' such that the data was authenticated while handling e' using the keys available at that time (L.5). Moreover, (1) either the sender of the data was correct while handling e' and sent the data as part of a message following the process described by P (L.6–8); or (2) the node at which e' occurred was Byzantine at that time, and either it generated the data itself (e.g. when $e'' = e'$), or it impersonated some other replica (by obtaining the keys that some node leaked at event e'') (L.10–12).

We used a few undefined abstractions in this predicate: An AbsProcess is an abstraction of a process, i.e., a function that returns the collection of messages generated while handling a given event: (∀ (eo : EO) (e : Event), list DirectedMsg). The bind_op_list function is wrapped around get_contained_auth_data to handle the fact that trigger might return None, in which case bind_op_list returns nil. The verify_auth_data function takes an authenticated message a and some keys and: (1) invokes data_sender (defined in Section 3.1.3) to extract the expected sender s of a; (2) searches among its keys for a receiving_key that it can use to verify that s indeed authenticated a; and (3) finally verifies the authenticity of a using that key and the verify function. The authenticate function simply calls create and uses the sending keys to create tokens. The got_key_for function takes

---

[7] For readability, we show a slightly simplified version of this axiom. The full axiom can be found in `code/model/EventOrdering.v`.

a name *i* and two local_keys *lk1* and *lk2*, and states that the sending keys for *i* in *lk1* are all included in *lk2*.

However, it turns out that because we never reason about faulty nodes, we never have to deal with the right disjunction of the above formula. Consequently, this assumption about received messages can be greatly simplified when we know that the sender is a correct replica, which is always the case when we use this assumption. This is so because BFT protocols are designed so that there is always a correct node that can be used to track down causal chains of events. We now define the following simpler assumption, which we have proved to be a consequence of AXIOM_authenticated_messages_were_sent_or_byz:

```
Definition AXIOM_authenticated_messages_were_sent_non_byz (P : AbsProcess) :=
  ∀ (e : Event) (a : auth_data) (c : name),
    In a (bind_op_list get_contained_auth_data (trigger e))
    → has_correct_trace_before e c
    → verify_auth_data (loc e) a (keys e) = true
    → data_sender (loc e) (am_data a) = Some c
    → ∃ e' dst m, e' ≺ e ∧ loc e' = c
        ∧ am_auth a = authenticate (am_data a) (keys e')
        ∧ In a (get_contained_auth_data m)
        ∧ In (m,dst) (P eo e')
```

As opposed to the previous formula, this one assumes that the authenticated data was sent by a correct replica, which has a correct trace prior to the event *e*—the event when the message containing *a* was handled.

3.1.7.2 Assumption 2 (Correct Keys)

Because our framework enables reasoning about implementations of protocols, processes need to store their keys in the state in order to sign and verify messages. We must connect those keys to the ones in the model. We do this through the AXIOM_correct_keys assumption, which states that for each event *e*, if a process *sm* has a correct trace up to *e*, then the keys (keys *e*) from the model are the same as the ones stored in its state (which are computed using $sm@^{-}e$).

3.1.7.3 Assumption 3 (Number of Faulty Nodes)

Finally, we present our assumption regarding the number of faulty nodes. There are several ways to state that there can be at most *f* faulty nodes. One simple definition is (where node is a subset of name as discussed in Section 3.2.2):

```
Definition AXIOM_exists_at_most_f_faulty (E : list Event) (f : nat) :=
  ∃ (faulty : list node),
    length faulty ≤ f ∧ ∀ e1 e2, In e2 E
    → e1 ≼ e2
    → ~ In (loc e1) faulty
    → has_correct_bounded_trace e1.
```

This assumption says that at most *f* nodes can be faulty by stating that the events happening at nodes that are not in the list of faulty nodes *faulty*, of length less or equal than *f*, are correct up to some point characterized by the partial cut *E* of a given event ordering (i.e., the collection of events happening before those in *E*).

3.1.7.4 Assumption 4 (Collision Resistant Hashing)

Very often, in order to reduce the amount of information that has to be exchanged, different nodes of distributed protocols exchange hashes of messages instead of sending entire messages. For example, pre-prepare messages in PBFT protocol contain client requests, but prepare and commit messages simply contain digests of client requests. In order to enable application as well as verification of hash function, our model enables adding these kind of functions as parameters of a distributed protocol, as well as assumptions about these functions. For example, our PBFT formalization is parametrized by the following *create* and *verify* functions, and we assume that the create function is collision resistant:

```
Class PBFThash := MkPBFThash {
  create_hash : list PBFTmsg → digest;
  verify_hash : list PBFTmsg → digest → bool; }.

Class PBFThash_axioms := MkPBFThash_axioms {
  AXIOM_create_hash_collision_resistant :
    ∀ msgs1 msgs2, create_hash msgs1 = create_hash msgs2 → msgs1 = msgs2; }.
```

3.1.7.5 Assumption 5 (Synchronous Delivery)

Unlike asynchronous protocols, which make no assumptions regarding time, synchronous algorithms crucially depend on time bounds. In [CGR11, Section 2.5.2], the authors define a synchronous system as one for which the following two properties hold:

1. *Synchronous computation*: There is a known upper bound on processing delays.

2. *Synchronous communication*: There is a known upper bound on message transmission delays.

Alternatively, in [LSP82], the authors assume that the absence of a message can be detected (see assumption A3 in [LSP82, Section 3]). In [LSP82, Section 6], the authors then show that this assumption can be substantiated when the two properties listed above hold. Correspondingly, we model A3 as follows (note that this assumption also covers assumption A1 presented in [LSP82, Section 3]):

```
Definition AXIOM_messages_get_delivered (eo : EO) F (sys : System(F)) :=
  ∀ (e : Event) (m : DirectedMsg) (d : name),
    In m (sys ⇝ e)
    → In d (dmDst m)
    → is_correct_in_near_future d e
    → ∃ e', loc e' = d ∧ (trigger e') = Some (dmDst m)
      ∧ (time e + τ ≤ time e' ≤ time e + (μ + τ)).
```

i.e., if at some event *e*, a message *m* was sent by a correct node and it was not delayed, then there exists some prior event *e'*, that triggered the message *m*, such that it did not take it more than $(\mu+\tau)$, but also not less than τ, for the message to arrive. Here, is_correct_in_near_future covers the usual assumption of synchronous protocol, which states that every message sent by a correct node is delivered within the same round (i.e., by $\mu + \tau$):

```
Definition is_correct_in_near_future (d : name) {eo : EO} (e : Event) :=
    ∃ e', loc e' = d ∧ has_correct_trace_before e' d ∧ (time e + (μ + τ) ≤ time e').
```

3.1.7.6 Assumption 6 (Authentication)

In order to prevent malicious behavior of its participants, many BFT protocols use signed messages (e.g., SM and PBFT). Thanks to those signatures, correct processes can verify the authenticity of the data sent by other correct processes, i.e.:

```
Definition AXIOM_verified_authenticated (eo : EO) :=
    ∀ (e1 e2 : Event) d,
      has_correct_trace_before e1 (loc e1)
      → has_correct_trace_before e2 (loc e2)
      → ∃ rk tok,
          In rk (lookup_receiving_keys (keys e2) (loc e1))
          ∧ In tok (authenticate d (keys e1))
          ∧ verify d (loc e1) rk tok = true.
```

For this assumption to hold, *e2* must have a receiving key for *e1* (which can be extracted using lookup_receiving_keys) and *e1* must have a signing key for *e2*.

3.1.7.7 Assumption 7 (Non-Repudiation)

Many distributed protocols (e.g., SM) also assume that if some authenticated piece of data can be verified at one correct node, then it can be verified at all correct nodes[8]:

```
Definition AXIOM_all_correct_can_verify (eo : EO) :=
  ∀ (e1 e2 : Event) a,
    has_correct_trace_before e1 (loc e1)
    → has_correct_trace_before e2 (loc e2)
    → verify_auth_data (loc e1) a (keys e1) = true
    → verify_auth_data (loc e2) a (keys e2) = true.
```

3.2 Local and Distributed Reasoning

To reason about distributed systems, one has to prove both: local invariants of processes, as well as properties that hold about the collection of processes that form the distributed systems. To simplify proofs of local properties, we have actually developed an automated proof technique (see Section 3.2.1), and to enable distributed reasoning we are relying on the standard quorum reasoning (see Section 3.2.3 and Section 3.2.2). Moreover, to reason about distributed properties at a high-level of abstraction, we have developed a knowledge library, which we discuss in the next chapter (see Chapter 4), and which among other things captures the typical quorum reasoning discussed in this section.

3.2.1 Automated Inductive Reasoning

We use induction on causal time to prove both distributed and local properties. As discussed here, we automated the typical reasoning patterns we use to prove local properties. For example, in our PBFT formalization, we proved following: if a replica has a prepare message in its log, then it either received or generated it. Moreover, using ByLoE we prove local properties about processes by reasoning about all possible paths they can take when reacting to messages. Thus, a typical proof of such a lemma using our framework goes as follows: (1) we go by induction on events; (2) we split the code of a process into all possible execution paths; (3) we prune the paths that could not happen because they invalidate some hypotheses of the lemma being proved; and (4) we automatically prove some other cases by induction hypothesis. We packaged this reasoning as a Coq tactic, which in practice can significantly reduce the number of cases to prove. Besides general

---

[8] Note that AXIOM_all_correct_can_verify is true when using digital signatures, but not when using MACs.

tactics, which can be used to prove properties of any protocol, we also developed protocol specific tactics, which significantly reduced the costs of formal verification of SM, PBFT and MinBFT properties. For example, we used this automation technique to prove local properties of PBFT, such as Castro's A.1.2 local invariants [Cas01]. Because of PBFT's complexity, our Coq tactic typically reduces the number of cases to prove from between 50 to 60 cases down to around 7 cases, sometimes less, as we show in this histogram of goals left to interactively prove after automation:

										avg. # goals/lemma
# of goals left to prove	0	1	2	3	4	5	6	7	8	
# of lemmas	8	1	5	4	4	2	9	17	3	5

3.2.2 Quorums

As usual, we use quorum theory to trace back correct information between nodes. A (Byzantine) quorum w.r.t. a given set of nodes N, is a subset Q of N, such that $f + 1 \leq (2 * |Q|) - |N|$ (where $|X|$ is the size of X), i.e. every two quorums intersect [MR97]; [Vuk10] in sufficiently many replicas. We use here Castro's notation where quorums are *majority* quorums [Tho79] (also called *write quorums*) that require intersections to be non-empty, as opposed to *read quorums* that are only required to intersect with write quorums [Gif79]. Typically, a quorum corresponds to a majority of nodes that agree on some property. In case of state machine replication, quorums are used to ensure that a majority of nodes agree to update the state using the same operation. If we know that two quorums intersect, then we know that both quorums agree, and therefore that the states cannot diverge. In order to reason about quorums, we have proved the following general lemma:[9]

```
Lemma overlapping_quorums :
  ∀ (l1 l2 : NRlist node), ∃ Correct,
    (length l1 + length l2) - num_nodes ≤ length Correct
    ∧ subset Correct l1 ∧ subset Correct l2 ∧ no_repeats Correct.
```

This lemma implies that if we have two sets of nodes *l1* and *l2* (NRlist ensures that the sets have no repeats), such that the sum of their length is greater than the total number of nodes (num_nodes), there must exist an overlapping subset of nodes (*Correct*). We use this result below in Chapter 4.

---

[9] We present here a simplified version for readability.

The node type parameter is the collection of nodes that can participate in quorums. For example, PBFT replicas can participate in quorums but clients cannot. This type comes with a node2name function to convert nodes into names.

Notation. In the rest of the thesis, we will sometimes write $\mathrm{EventN}(N)$, where N is a predicate on process names, for the collection of events e such that $N(\mathrm{loc}(e))$ and such that $\mathrm{loc}(e)$ is a node (in node), i.e., for the collection of events happening at nodes that satisfy the constraint N. In addition, we will write EventN for the type of events happening at nodes.

3.2.3 Certificates

Lemmas that require reasoning about several replicas are much more complex than local properties, because they typically require reasoning about some information computed by a collection of replicas (such as quorums) that vouch for the information. In PBFT, a collection of $2f+1$ messages from different replicas is called a *strong (or quorum) certificate*, and a collection of $f+1$ messages from different replicas is called a *weak certificate*.

When working with strong certificates, one typically reasons as follows: (1) Because PBFT requires $3f+1$ replicas, two certificates of size $2f+1$ always intersect in $f+1$ replicas. (2) One message among those $f+1$ messages must be from a correct replica because at most f replicas can be faulty. (3) This correct replica can vouch for the information of both quorums—we use that replica to trace back the corresponding information to the point in space/time where/when it was generated. We will get back to this in Chapter 4.

When working with weak certificates, one typically reasons as follows: Because, the certificate has size $f+1$ and there are at most f faulty nodes, there must be one correct replica that can vouch for the information of the certificate.

Chapter 4

ByK: Byzantine Knowledge Theory for Homogeneous Systems

Knowledge theories have applications in many areas, such as, as mentioned in [Fag+03], economics, linguistics, artificial intelligence, theoretical computer science, and, evidently, distributed computing. This is because the way humans, machines, distributed systems, etc., manage to achieve tasks, or simply evolve is by making new discoveries and exchanging their knowledge so that others can know about it and benefit from it. Actually, the way they exchange this information forms the high-level *logic* of a system. Understanding and being able to reason about this logic is one of the major difficulties when dealing with distributed systems. Additionally, because the same high-level logic is typically shared by many systems, such theories allow reusing results proved at that level in multiple applications. Therefore, we decided to develop a Byzantine Knowledge library (ByK), to reason at a high-level of abstraction about the *knowledge* exchanged between the nodes of a distributed system.

Knowledge is often captured using *possible-worlds* models, which rely on Kripke structures: an agent knows a fact if that fact is true in all possible worlds. For distributed systems, agents are nodes and a possible world at a given node is essentially one that has the same local history as the one of the current world, i.e., it captures the current state of the node. As Halpern stresses, e.g. in [Hal87], such a definition of knowledge is *external* in the sense that it cannot necessarily be computed, though some work has been done towards deriving programs from knowledge-based specifications [Bic+04]. Therefore, because we want to reason about knowledge propagation, we follow a different, more pragmatic and computational approach, and say that a node knows some piece of data if it is somehow encoded in node's local state, as opposed to the external and logical notion of knowing facts mentioned above. This computational notion of knowledge relies on exchanging messages to propagate it, which is what is required to derive pro-

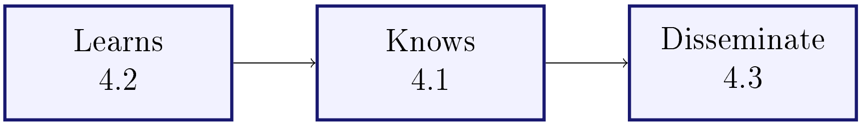


Figure 4.1: Knowledge operators

grams from knowledge-based specifications (i.e., to compute that some knowledge is gained [Hal87]; [CM86]).

Let us now present an excerpt of our distributed epistemic Byzantine Knowledge library that relies on three modal operators, which allow us to reason about gaining some knowledge, holding some knowledge, and finally disseminating some knowledge (see Figure 4.1). We first start by introducing the *knows* modal operator (see Section 4.1), which is most commonly found in knowledge theories, and which allows reasoning about the knowledge held by processes. Next, we introduce our *learns* modal operator (see Section 4.2), which allows reasoning about the knowledge gained by processes. In addition, we introduce two variants of this general *learns* operator: one called *learns_on_time*, which can be used to especially reason about synchronous systems; and one called *learns_list*, which enables reasoning about pieces of data that are themselves combinations of several pieces of data. Finally, we introduce our *disseminate* modal operator, which allows reasoning about the dissemination of knowledge by some process to other processes (see Section 4.3). Moreover, because framework presented in this thesis supports reasoning about Byzantine faults, our ByK library also supports reasoning about authenticated knowledge (see Section 4.4). For each of these modal operators, we present the parameters, as well as axioms *about* these operators. We end each section with most important general purpose lemmas about knowledge.[1] We used these general purpose lemmas to prove agreement of PBFT and SM. Actually, using two knowledge lemmas presented in this chapter (namely, learns_on_time_imp_other_knew and disseminated_before_deadline) we managed to abstract away most of the proof of SM's agreement property. We end this chapter (see Section 4.5 and Section 4.6) by showing how we used ByK to prove properties about SM and PBFT, respectively.

4.1 Knowing

We now extend the model presented in Section 3.1 with an epistemic modal operator *knows* that expresses what it means for a process to hold some information.

---

[1] In this chapter we only present axioms on which our general purpose lemmas depend on. We refer the reader to our implementation for the full list of axioms: `code/model/Disseminate.v`.

Parameters. Formally, we extend our model with the following parameters, which can be instantiated as many times as needed for all the pieces of known data that one wants to reason about—see below for examples:

```
Class KnowingClass := MkKnowingClass {
    byk_data : Type;
    byk_info : Type;
    byk_mem : Type;
    byk_data2info : byk_data → byk_info;
    byk_knows : byk_data → byk_mem → Prop;
    byk_sys : System(byk_mem); }.
```

The byk_data type is the type of "raw" data that nodes have knowledge of; while byk_info is some distinct information that might be shared by different pieces of data. For example, PBFT replicas collect batches of $2f+1$ (pre-)prepare messages from different replicas, that share the same view, sequence number, and digest (see Section 2.1.2). In that case, the (pre-)prepare messages are the raw data that contain the common information consisting of a view, a sequence number, and a digest. The byk_mem type is the type of objects used to store one's knowledge, such as a state machine state. One has to provide a byk_data2info function to extract the information contained in some piece of data. The byk_knows predicate explains what it means to know some piece of data. Finally, byk_sys is the system that one wants to reason about. For simplicity, we require here that all the nodes in that system must maintain a state of type byk_mem, which they use to store their knowledge.

Primitive Modal Operators. Let us now turn to one of the main components of our theory, namely the knows epistemic modal operator. This operator provides an abstraction barrier: it allows us to abstract away from *how* knowledge is stored and computed, in order to focus on the mere *fact* that we have that knowledge.

```
Definition knows (e : Event) (d : byk_data) :=
  ∃ mem i, loc e = node2name i
    ∧ (byk_sys i)@+ e = Some mem
    ∧ byk_knows d mem.
```

This says that the state machine (byk_sys i) knows the data d at event e if its state is *mem* at e and (byk_knows d *mem*) is true.

Non-Primitive Modal Operators. As mentioned in Section 3.2.3, when reasoning about distributed systems, one often needs to reason about certificates, i.e., about collections of messages from different sources. In order to capture this, we introduce knows_certificate predicate, which states that a state machine *sm* (running at loc *e*) knows the information *i* at event *e* if there exists a list *l* of pieces of data of length at least *k* (the certificate size) that come from different sources, and such that *sm* knows each of these pieces of data, and each piece of data carries the common information *i*:

```
Definition knows_certificate (e : Event) (k : nat) (i : byk_info) (P : list byk_data → Prop) :=
  ∃ (l : list byk_data),
    k ≤ length l ∧ P l
    ∧ no_repeats (map byk_data2owner l)
    ∧ ∀ d, In d l
    → (knows e d ∧ i = byk_data2info d).                                                    (4.1)
```

Axioms. Using our knowledge library, we derived a number of general high-level knowledge principles about the way knowledge is exchanged between the nodes of a system. Typically these results rely on assumptions about the behavior of the system. In order to use those lemmas to prove a property about a concrete instance of a distributed system (such as SM or PBFT, for example), one has to prove that those assumptions indeed hold about the distributed system. We briefly describe in this section, as well as in Section 4.2, Section 4.3, and Section 4.4, some "standard" axioms that we defined within ByK. We then show in Section 4.4 how to use those axioms to derive useful results about knowledge, which we used, for example, to prove properties about SM.[2]

Axiom 1 (Preserving Knowledge)*.* The AXIOM_preserves_knows axiom is only valid about protocols that do not forget. It says that if a node knows a piece of data *d*, then it will keep on knowing *d* as long as it stays correct. We formally define this axiom as follows:

```
Definition AXIOM_preserves_knows (eo : EO) : Prop :=
  ∀ (e1 e2 : Event) (d : byk_data),
    has_correct_bounded_trace e2
    → e1 ⊑ e2
    → knows e1 d
    → knows e2 d.
```

---

[2] We refer the reader to our implementation for the full list of axioms provided by ByK: `code/model/Disseminate.v`.

For example, this axiom is true about SM, because as opposed to PBFT, in SM, processes do not garbage collect, i.e., they do not forget (see Section 2.1).[3] We used this axiom to prove disseminated_before_deadline (see Figure 4.2). As explained below, this lemma assumes that a node n learns a piece of data d owned by a node n' at some event e. From the fact that n learned d, we can prove that there exists some event e'' that happened at n' before some deadline D, at which n' disseminated d. Moreover, from the fact that n' disseminated d, we can derive that n' knows d. Finally, thanks to AXIOM_preserves_knows, we conclude that since n' knows d at e'' that happened before the deadline D, then it must also know d at events that happen after the deadline.

Lemmas. We proved the following lemma, which captures the fact that there is always a correct replica that can vouch for the information of a weak certificate:

```
Lemma knows_weak_certificate :
 ∀ (e : Event) (k f : nat) (i : byk_info) (P : list byk_data → Prop) (E : list Event),
  (f < k ∧ AXIOM_exists_at_most_f_faulty E f ∧ In e E ∧ knows_certificate e k i P)
  → ∃ d, has_correct_trace_before e (node2name d) ∧ knows e d ∧ i = byk_data2info d.
```

4.2 Learning

We now build on top of the knowledge theory presented in Section 4.1 by introducing one additional epistemic modal operator, called *learns*, that expresses what it means for a process to gain some information, and which bears some resemblance with the *fact discovery* notion discussed in [HM90]. Additionally, we here present two variants of this operator: (1)*learns_on_time* that is especially useful to reason about synchronous systems; and (2)*learns_list* that is useful to reason about pieces of data that are themselves combinations of several pieces of data.

Parameters. In addition to the parameters introduced in Section 4.1, we here present five additional parameters:

```
Class LearningClass := MkLearningClass {
    byk_data2owner : byk_data → node;
    byk_data2msg : byk_data → msg;
    byk_data2auth : byk_data → auth_data;
    byk_data2auth_list : byk_data → list auth_data;
    byk_verify : ∀ (eo : EO) (e : Event) (d : byk_data), bool; }.
```

---

[3] PBFT's garbage collection does not affect the knowledge majority of correct processes agreed upon, because it only removes the messages that happened before garbage collection. Based on this, we can conclude that PBFT only forgets the knowledge that lead to the garbage collection.

The byk_data2owner function extracts the "owner" of some piece of data, typically the node that generated the data. Next, byk_data2msg enables converting a piece of data to a message. This essentially allows us to consider an entire message as a piece of data because we can then convert the underlying piece of data of a message to that message whenever we need to manipulate the message—see below its use in learns_on_time. While, the byk_data2auth function extracts some piece of authenticated data from some piece of raw data, the byk_data2auth_list extracts all pieces of authenticated data from some piece of raw data. In most cases protocols assume that a node signs the whole message before it disseminates that message to other nodes (e.g., PBFT and $\mathrm{SM}_{\mathsf{seq}}$), so it is enough to extract authenticated data that was created before message was sent (i.e., byk_data2auth is used). In case one is reasoning about protocols which assume that each received message is composed of a collection of authenticated pieces of data (e.g, $\mathrm{SM}_{\mathsf{mul}}$) we need means to extract all of these pieces of authenticated data (i.e., byk_data2auth is used). Finally, because typically in Byzantine fault-tolerant protocols messages are authenticated using signatures, the byk_verify function allows verifying the correctness of authenticity of a piece of data. For convenience, we define the following wrapper around byk_data2owner:

```
Definition byk_data2node (d : byk_data) : name := node2name (byk_data2owner d).
```

Primitive Modal Operators. Let us now turn to the second main component of our theory, namely the learns epistemic modal operator. We define learns as follows:

```
Definition learns (e : Event) (d : byk_data) :=
  In (byk_data2auth d) (bind_op_list get_contained_auth_data (trigger e))
  ∧ byk_verify e d.
```

This states that a node, at which event e happened, learns d, if e was triggered by a message that contains the data d. Moreover, to handle Byzantine faults, we require that to learn some data one has to be able to verify its authenticity.

Although most of the knowledge lemmas we proved depend on this learns operator, we used a variant of this operator called learns_on_time to prove properties of synchronous systems (e.g., see learns_on_time_imp_other_knew in Figure 4.3). While learns requires that the event e be triggered by a message that contains the data d, the learns_on_time operator requires that the event e be triggered by the data d itself. For example, for both implementations of the SM protocol we instantiated ByK such that: if a node j_k sends some value v that was previously signed by senders j_0,...,j_{k-1}, then a node learns_on_time a piece of data that consists of the value v accompanied by all the signatures (i.e., j_0,...,j_k), while a node learns a piece of data that contains the value v along with only one of those signatures. In addition, our new modal operator learns_on_time requires that the data d be delivered at a timely "epoch" (provided by the E parameter) and that this epoch number is less than some bound that essentially corresponds to a deadline by which the data must have been delivered (provided by the D parameter), i.e., the proposition on_time e d E D is true. This operator is formally defined as follows:

```
Definition learns_on_time {eo : EO}
  (e : Event) (d : byk_data) (E : byk_data → nat) (D : nat) :=
  trigger e = Some (byk_data2msg d)
  ∧ byk_verify e d = true
  ∧ on_time e d E D.
```

where, on_time is defined as follows:

```
Definition on_time {eo : EO} (e : Event) (d : byk_data) (E : byk_data → nat) (D : nat) :=
  (time e <= (E d) * (μ + τ)) ∧ (E d <= D).
```

As mentioned in Section 2.1.1, we implemented two variants of SM, which we named $\text{SM}_{\textsf{seq}}$ and $\text{SM}_{\textsf{mul}}$. Although the operators we introduced so far were enough to prove properties (at the level of knowledge) of PBFT and $\text{SM}_{\textsf{seq}}$, in order to verify $\text{SM}_{\textsf{mul}}$ we extended ByK with an additional operator called learns_list. As it turns out, our main reason for introducing this operator was due to the way $\text{SM}_{\textsf{mul}}$ works: whenever some node n learns a piece of data d, it learns about a list of signatures that correspond to all the nodes that have also learned about d. In this case, we can say that n actually learned a list of authenticated pieces of data. Here is our formal definition of learns_list operator:

```
Definition learns_list (e : Event) (d : byk_data) :=
  subset (byk_data2auth_list d) (bind_op_list get_contained_auth_data (trigger e))
  ∧ byk_verify e d.
```

Non-Primitive Modal Operators. Here, we present a few predicates that are useful to track down knowledge by navigating trough chains of corresponding *learns* and *knows* propositions. The learns_or_knows property is a local predicate that states that for a state machine to know about a piece of information it has to either have learned about it or generated it.

```
Definition learns_or_knows :=
  ∀ (d : byk_data) (e : Event),
    knows e d
    → (∃ e', e' ⊑ e ∧ learns e' d) ∨ (byk_data2node d = loc e).          (4.2)
```

The learns_if_knows property is a distributed predicate that states that if one learns about some piece of information that is owned by a correct node, then that correct node must have known that piece of information:

```
Definition learns_if_knows :=
  ∀ (d : byk_data) (e : Event),
    learns e d
    → has_correct_trace_before e (byk_data2node d)
    → ∃ e', e' ≺ e ∧ loc e' = byk_data2node d ∧ knows e' d.          (4.3)
```

Axioms. As in Section 4.1, we define here some “standard” axioms that we defined within ByK, and which we use to derive useful results about knowledge, for example, to prove properties about SM in Section 4.4.

Axiom 2 (Relating Verify). The AXIOM_verify_implies_verify_auth_data relates the two verify functions mentioned above, namely byk_verify and verify_auth_data. It states that if one can verify a piece of knowledge using byk_verify, then one must also be able to verify the authenticity of the corresponding authenticated piece of data using verify_auth_data. We formally define this axioms as follows:

```
Definition AXIOM_verify_implies_verify_auth_data (eo : EO) :=
  ∀ (e : Event) (d : byk_data),
    byk_verify e d = true
    → verify_auth_data (loc e) (byk_data2auth d) (keys e) = true.
```

For example, we used this axiom in our proof of disseminated_before_deadline (see Figure 4.2). This lemma assumes that a node n learns a piece of data d owned by a node n' at some event e. Because n learned d, using our *Origin of Triggers* communication assumption (see Section 3.1.7.1) we can prove that there exists some prior event e'' at which this piece of data was sent. Finally, because the verify function used in our communication assumption is different from the one used in our learns operator, we need AXIOM_verify_implies_verify_auth_data to derive one from the other.

Axiom 3 (Knows if Learns on Time). The AXIOM_learns_on_time_implies_knows axiom states that nodes get to know about pieces of data that they learn on time (in synchronous system, one would typically discard pieces of data that arrived late). We formally state this axiom as follows:

```
Definition AXIOM_learns_on_time_implies_knows (eo : EO) N K E D :=
  ∀ (e : EventN(N)) d,
    has_correct_trace_before e (loc e)
    → K(d)
    → learns_on_time e d E D
    → knows e d.
```

Here $K(d)$ is a data constraint. For example, we used this assumption to prove learns_on_time_imp_other_knew (see Figure 4.3), which is true about SM. This lemma assumes that a node n learns about a piece of data d on time at some event e. Based on this assumption, we can prove that d is first extended with n's signature, and then disseminated to all nodes that have not already signed d. If a node n' receives this message, it will learn about it. Thanks to AXIOM_learns_on_time_implies_knows we can derive that n' knows the piece of data d extended by n.

Lemmas. Using Equation (4.2) and Equation (4.3), we have proved this general lemma about knowledge propagating through nodes:

```
Lemma knows_propagates :
  ∀ (e : Event) (d : byk_data),
    learns_or_knows
    → learns_if_knows
    → knows e d
    → has_correct_trace_before e (byk_data2node d)
    → ∃ e', e' ⪯ e ∧ loc e' = byk_data2node d ∧ knows e' d.
```

This lemma says that, assuming Equation (4.2) and Equation (4.3), if one knows at some event e some data d that is owned by a correct node, then that correct node must have known that data at a prior event e'. We use this lemma to track down information through correct nodes.

Moreover, using Equation (4.1), we can combine the quorum and knowledge theories to prove the following lemma, which captures the fact that if there are two quorums for information *i1* (known at *e1*) and *i2* (known at *e2*), and the intersection of the two quorums is guaranteed to contain a correct node, then there must be a correct node (at which *e1'* and *e2'* happen) that owns and knows both *i1* and *i2*—this lemma follows from knows_propagates and overlapping_quorums (see Section 3.2.2):

```
Lemma knows_in_intersection :
  ∀ (e1 e2 : Event) (i1 i2 : byk_info) (k f : nat) (P : list byk_data → Prop) (E : list Event),
    (learns_or_knows ∧ learns_if_knows)
    → (k ≤ num_nodes ∧ num_nodes + f < 2 * k)
    → (AXIOM_exists_at_most_f_faulty E f ∧ In e1 E ∧ In e2 E)
    → (knows_certificate e1 k i1 P ∧ knows_certificate e2 k i2 P)
    → ∃ e1' e2' d1 d2,
        loc e1' = loc e2' ∧ e1' ⪯ e1 ∧ e2' ⪯ e2
        ∧ loc e1' = byk_data2node d1 ∧ loc e2' = byk_data2node d2
        ∧ knows e1' d1 ∧ knows e2' d2
        ∧ i1 = byk_data2info d1 ∧ i2 = byk_data2info d2.
```

4.3 Disseminating

In this section we present our *disseminate* modal operator, which expresses what it means for a process to propagate some piece of information, and which bears some resemblance with the *fact publication* notions discussed in [HM90].

Primitive Modal Operators. We are interested in distributed systems that communicate via message-passing. Disseminating messages allow processes to propagate some knowledge they have gained, to other processes. Consequently, we now define several disseminate operators that capture several ways processes disseminate their knowledge to other processes.[4] For example, our primitive modal operator disseminate is defined as follows:

```
Definition disseminate {eo : EO} (e : Event) (d : byk_data) :=
  ∃ (m : DirectedMsg),
    In m (byk_sys ⇝ e)
    ∧ In (byk_data2auth d) (get_contained_auth_data (dmMsg m)).
```

This operator states that the authenticated piece of data *d* was disseminated at the event *e* if there exists a message *m*, such that *m* was sent at *e* and such that the authenticated piece of data corresponding to *d* is part of *m*'s authenticated parts.

---

[4] We refer reader to our implementation in `code/model/Disseminate.v`

Although disseminate is already useful in many contexts, some distributed systems require a more fine-grained reasoning about the recipients of disseminated pieces of knowledge. For example, some distributed protocols require that a piece of knowledge is disseminated to a list of recipients. To achieve this, we introduce here another primitive modal operator, called disseminate_to_list, which is defined as follows:

```
Definition disseminate_to_list {eo : EO} (e : Event) (d : byk_data) (L : list node_type) :=
  ∃ (m : DirectedMsg),
    In m (byk_sys ⇝ e)
    ∧ In (byk_data2auth d) (get_contained_auth_data (dmMsg m))
    ∧ subset (map node2name L) (dmDst m).
```

Again, even though disseminate and disseminate_to_list are already useful in many contexts, some distributed systems require an even more fine-grained reasoning about the entire disseminated pieces of knowledge, and not just the authenticated pieces of data contained in the disseminated pieces of knowledge. For example, it is not always enough to know that an authenticated piece of data signed by (at least) k processes was disseminated by a process. Sometimes we need to know that this process did not disseminate more than k signatures. To achieve this, we introduce yet another primitive modal operator, called disseminate_top_to_list, which is defined as follows:

```
Definition disseminate_top_to_list {eo : EO} (e : Event) (d : byk_data) (L : list node_type) :=
  ∃ dst,
    In (MkDMsg (byk_data2msg d) dst ('0)) (byk_sys ⇝ e)
    ∧ subset (map node2name L) dst.
```

Here, as explained in Section 3.1.2, (MkDMsg *msg dst delay*) creates a directed message, which contains a message *msg*, which will be sent to the list of recipients *dst* after a delay *delay*.

Non-Primitive Modal Operators. Let us now define a few useful non-primitive operators on top of the primitive modal operators introduced above. For example, because sometimes we only care about the fact that some piece of knowledge was disseminated by a process *src* to another process *dst*, let us define the following instances of the disseminate_to_list and disseminate_top_to_list operators:

```
Definition disseminate_to {eo : EO} (e : Event) (d : byk_data) (dst : node_type) :=
  disseminate_to_list e d [dst].

Definition disseminate_top_to {eo : EO} (e : Event) (d : byk_data) (dst : node_type) :=
  disseminate_top_to_list e d [dst].
```

In addition, let us define the following instances of the disseminate_to_list and disseminate_top_to_list operators, which express that a piece of data d is disseminated to all processes except the ones mentioned in L (where nodes is the collection of all the nodes participating in the byk_sys system):

```
Definition disseminate_to_except
  {eo : EO} (e : Event) (d : byk_data) (L : list node_type) :=
    disseminate_to_list e d (nodes\L).

Definition disseminate_top_to_except
  {eo : EO} (e : Event) (d : byk_data) (L : list node_type) :=
    disseminate_top_to_list e d (nodes\L).
```

Axioms. As in Section 4.1, we define here some "standard" axioms that we defined within ByK, and which we use to derive useful results about knowledge, for example, to prove properties about SM in Section 4.4.

Axiom 4 (On Time). The AXIOM_messages_are_disseminated_before_deadline axiom states that if a piece of data d is disseminated at an event e by a correct node, then e happened before the deadline D. We formally define this axiom as follows:

```
Definition AXIOM_messages_are_disseminated_before_deadline
  (eo : EO) (N : name → Prop) (D : dt_T) :=
    ∀ (e : EventN(N)) (d : byk_data),
      has_correct_trace_before e (loc e)
      → disseminate e d
      → time e <= D.
```

This assumption holds about the class of protocols where all messages are disseminated before a given deadline. For example, it holds for any round-based synchronous protocol that completes in a bounded number of rounds. The deadline D is then the time by which those rounds must have completed. In such protocols, one only cares about messages that were sent before D. We used this assumption, for example, to prove disseminated_before_deadline (see Figure 4.2), which is true about SM. As mentioned above, this lemma assumes that a node n learns about a piece of data d at some event e. Because n learned about d, we can derive that there must exist some prior event e'' at which d was disseminated. Using AXIOM_messages_are_disseminated_before_deadline, we can further derive that *d* was disseminated on time at *e"*, in the sense that this happened before the deadline *D*.

Axiom 5 (Knows if Disseminate). The AXIOM_knows_if_disseminate axiom states that a node that disseminates a piece of data must also know about it. We formally define this axiom as follows:

```
Definition AXIOM_knows_if_disseminate (eo : EO) : Prop :=
  ∀ (e : EventN) (d : byk_data),
    disseminate e d
    → knows e d.
```

This axiom holds for any protocol that "remembers" (e.g., logs) any piece of data it disseminated. For example, we used this assumption to prove disseminated_before_deadline (see Figure 4.2), which is true about SM. As mentioned above, this lemma assumes that a node *n* learns about a piece of data *d* at event *e*. Because *d* was learned about, we can conclude that there exists some prior event *e"* at which *d* was disseminated. Thanks to AXIOM_knows_if_disseminate, we can deduce that the node that disseminated this piece of data, knows about it.

4.4 Authenticated Knowledge

Although reasoning about knowledge dissemination is useful when reasoning about distributed systems in various contexts (for example under different system assumptions such as synchronous or asynchronous, or under different failure assumptions such as no faults, crash faults, or Byzantine faults); we are especially interested here in a version of *disseminate* that allows one tracing authenticated pieces of data in the context of Byzantine fault tolerant systems.

Parameters. To trace disseminated authenticated pieces of data, we introduce the following parameters on top of the ones presented above in Section 4.1 and Section 4.2:

```
Class AuthenticatedKnowledge := MkAuthenticatedKnowledge {
  byk_max_sign : nat;
  byk_ext_info : byk_info → Sign → byk_data;
  byk_ext_data : byk_data → Sign → byk_data;
  byk_data2data : byk_data → node_type → data;
  byk_data2sign : byk_data → Sign;
  byk_data2can : byk_data → list (byk_data_or_info * Sign); }.
```

As above, these parameters can be instantiated as many times as needed for all the disseminated authenticated pieces of data that one wants to reason about. Here Sign consists of two fields: (1) the name of the node that is signing a message; and (2) a list of Tokens. The byk_max_sign parameter abstracts away the total number of processes that can sign messages in the system that one wants to reason about (i.e., byk_sys). The byk_ext_info parameter creates a piece of data from some piece of information and a signature (e.g., in case of the SM protocol, when a commander wants to disseminate a value to the other generals, it appends its signature to the value and then sends it to all lieutenants). The byk_ext_data parameter creates a piece of data from a piece of data and a signature (e.g., in SM, this function is used when a lieutenant appends its signature to a message it received). The byk_data2data parameter allows casting a pieces of knowledge byk_data into a pieces of data. As mentioned above (see Section 3.1.3), in case of PBFT, we instantiated the data type as the disjoint union of bare messages, i.e., the disjoint union of objects that can be authenticated, while in one of our instantiation of this knowledge theory (see Section 4.1), we instantiated byk_data with *signed* (pre-)prepare messages. The byk_data2sign parameter extracts a signature from a piece of data. The byk_data2can parameter converts a piece of data into its canonical form, which is a list of pairs of (1) a piece of data/information and (2) a signature. Here byk_data_or_info is the disjoint union of byk_data and byk_info.

Axioms. As in Section 4.1, we define here some "standard" axioms that we defined within ByK, and which we use to derive useful results about knowledge, for example, to prove properties about SM in the next paragraph.

Axiom 6 (Authenticate Extended). The AXIOM_in_auth_data_trigger_extend axiom states that if an event is triggered by a message *m* that contains a piece of data *d*, extended with a signature *s*, then *m* must also contain the piece of data *d* itself (i.e., without the extension with the signature *s*).[5] We formally define this axiom as follows:

```
Definition AXIOM_in_auth_data_trigger_extend (eo : EO) :=
  ∀ (e : Event) d s,
    auth_data_in_trigger (byk_data2auth (byk_ext_data d s)) e
    → auth_data_in_trigger (byk_data2auth d) e.
```

---

[5] As it turns out, AXIOM_in_auth_data_trigger_extend is not true about SM_{mul}. Therefore, to prove SM_{mul}, we defined similar axiom called AXIOM_in_auth_data_trigger_extend_learns_list. We refer reader to `code/model/Disseminate.v` in our implementation for more details.

The (auth_data_in_trigger *a* *e*) proposition is true if the event *e* is triggered by a message that contains the authenticated piece of data *a*. This axiom is useful to reason about nested pieces of data, i.e., pieces of data that are extended using byk_ext_data (*a* is nested inside (byk_ext_data *d* *s*)). For example, we used this axiom to prove disseminated_before_deadline (see Figure 4.2), which is true about SM. This lemma assumes that a node *n* learns about a piece of data *d* owned by a node *n'*, at event *e* (i.e., *d* is contained in the message that triggered *e*). Because *n'* owns *d*, we can derive that it must have signed it, i.e., *d* must be some piece of data *d'* extended by a list of signatures *l*, such that the last signature in *d'* belongs to *n'*. Therefore, by induction on *l* and using AXIOM_in_auth_data_trigger_extend, we can derive that *d'* itself must be contained in the message that triggered *e*. We can then use our *Origin of Triggers* communication assumption (see Section 3.1.7.1) to get back to the time when *d'* was disseminated by *n'*.

Axiom 7 (Verify Extended Data). The AXIOM_verify_extend_implies axiom states that if one can verify a piece of data *d* extended with a signature *s*, then one can also verify *d* itself. We formally state this axiom as follows:

```
Definition AXIOM_verify_extend_implies (eo : EO) :=
  ∀ (e : Event) (d : byk_data) (s : Sign),
    byk_verify e (byk_ext_data d s) = true
    → byk_verify e d = true.
```

This axiom is also useful to reason about nested pieces of data. Similarly to the way we used our axiom 6 to prove disseminated_before_deadline (see Figure 4.2), we used AXIOM_verify_extend_implies to derive that *n* verified the piece of data *d'*, so that we can then use our *Origin of Triggers* communication assumption (see Section 3.1.7.1) to get back to the time when *d'* was disseminated by *n'*.

Axiom 8 (Extended Knows). The AXIOM_knows_extend axiom says that extensions should not influence whether or not pieces of knowledge are know. It states that if one knows a piece of data *d*, then one must also know any extension of that piece of data with a signature. We formally define this axiom as follows:

```
Definition AXIOM_knows_extend :=
  ∀ d s m,
    byk_knows d m
    → byk_knows (byk_ext_data d s) m.
```

As above, this axiom is also useful when reasoning about nested pieces of data. For example, we used this axiom to prove disseminated_before_deadline (see Figure 4.2), which is true about SM. This lemma assumes that a node *n* learns about a piece of data *d* owned by a node *n'* at some event *e*. Because *n'* owns *d*, we can derive that it must have signed it, i.e., *d* must be some piece of data *d'* extended by a list of signatures *l*, such that the last signature in *d'* belongs to *n'*. As explained above, using the axioms described above, we can further derive that *n'* must have disseminated, and know, *d'*. Finally, using AXIOM_knows_extend, we can prove by induction on *l*, that *n'* must also know about *d*.

Axiom 9 (Verify within Epoch). The AXIOM_same_epoch_implies_verify_extend axiom says that a correct node *n'* (running at loc *e'* below) will always be able to verify the extension of a piece of data *d* with a signature from some other node *n* (running at loc *e* below), provided that: (1) *n* is also correct, (2) it can verify *d*, and (3) those two verifications happen within the same epoch (i.e. events_in_same_epoch *e e'* is true). We formally define this axiom as follows:

```
Definition AXIOM_same_epoch_implies_verify_extend (eo : EO) :=
  ∀ (e e' : EventN) (d : byk_data),
    loc e <> loc e'
    → has_correct_trace_before e (loc e)
    → has_correct_trace_before e' (loc e')
    → ¬ In (loc e) (byk_data2senders d)
    → ¬ In (loc e') (byk_data2senders d)
    → events_in_same_epoch e e'
    → byk_verify e d = true
    → byk_verify e' (sign_data d (loc e) (keys e)) = true.
```

Here, sign_data *d* (loc *e*) (keys *e*) stands for the extension of the piece of data *d* with a signature from the node running at loc *e*, and byk_data2senders returns the names of all the nodes that signed the given piece of data. For example, we used this assumption to prove learns_on_time_imp_other_knew lemma (see Figure 4.3), which is true about SM. This lemma assumes that a node *n* learns about a piece of data *d* on time at some event *e*. If *n* has not yet signed *d*, it does so at *e*. It then disseminates *d* extended with its signature to all nodes that have not signed *d* yet—let us call this extended piece of data *d'*. Any other correct node *n'* that has not yet signed *d* will receive *d'*. Now, to prove that *n'* learns about *d'*, we need to show that it can verify its authenticity, which we prove using AXIOM_same_epoch_implies_verify_extend.

Axiom 10 (Just on Time). The AXIOM_just_learned_on_time_implies_disseminate axiom states that pieces of data that are newly learned about (in the sense that they were not known before, i.e., predicate didnt_know *e* *d* is true) must be extended and disseminated right away. We formally define this axiom as follows:

```
Definition AXIOM_just_learned_on_time_implies_disseminate (eo : EO) N E D :=
  ∀ (e : EventN(N)) d,
    just_learned_on_time e d E D
    → disseminate_top_to_except e (sign_data d (loc e) (keys e)) (loc e :: byk_data2senders d).
```

For example, we used this assumption to prove learns_on_time_imp_other_knew (see Figure 4.3), which is true about SM. This lemma assumes that a node *n* learns about a piece of data *d* on time. Thanks to the above axiom, we can conclude that *d* is first extended with *n*'s signature, and then disseminated to all nodes that have not signed *d*.

Axiom 11 (Epoch Numbers Increase). The AXIOM_extend_data_raises_epoch axiom states that extending a piece of data increases its epoch number. We formally define this axiom as follows:

```
Definition AXIOM_extend_data_raises_epoch E :=
  ∀ d s, E (byk_ext_data d s) = S (E d).
```

For example, we used this assumption to prove learns_on_time_imp_other_knew (see Figure 4.3), which is true about SM. This lemma assumes that a node *n* learns about a piece of data *d* on time at some event *e*. If *n* has not yet signed *d*, it does so at *e*. It then disseminates *d* extended with its signature to all nodes that have not signed *d* yet—let us call *d'* this extended piece of data. Any other correct node *n'* that has not yet signed *d* will receive *d'*. Finally, to prove that *n'* learns about *d'* on time, we use our axiom AXIOM_extend_data_raises_epoch.

Lemmas. We end this section by presenting two knowledge lemmas that we used to prove SM_{seq}'s agreement property (the safety part of IC1). We used similar lemmas to prove SM_{mul}'s agreement property as well. We refer the interested reader to our implementation (see `code/SM` and `code/SM2`) for more details.

The first lemma essentially states that if a node learns about some piece of data *d* at some event *e*, and that some other node *n'* owns that piece of data (namely loc *e'*), such that *e'* happened after some bound *D* (e.g., a deadline), then *n'* must have known that piece of data at *e'*. This lemma uses AXIOM_authenticated_messages_were_sent_or_byz (see Section 3.1.7), as well as following Assumptions: (1), (2), (4), (5), (6), (7) and (8). Those are elided here for readability. We formally state this lemma in Figure 4.2.

```
Lemma disseminated_before_deadline :
  ∀ {eo : EO} (N : name → Prop)
    (e : Event) (e' : EventN(N))
    (d : byk_data) (D : dt_T),
    has_correct_trace_before e (loc e')
    → has_correct_bounded_trace_lt e'
    → learns e d
    → owns e d (loc e')
    → D < time e'
    → knew e' d.
```

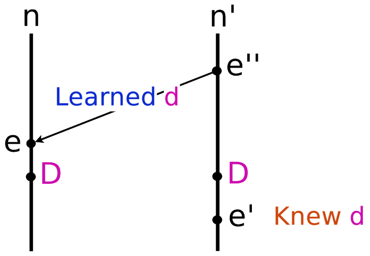


Figure 4.2: Knowledge lemma disseminated_before_deadline

The has_correct_trace_bounded_lt *e* proposition states that *e* has a correct trace, i.e., all local events prior to *e*, excluding *e*, are correct. Moreover, owns *e d n* states that *d* must be the extension of a piece of data *d'* that contains a signature from *n* as its last signature. Finally, while the modal operator knows talks about the knowledge of a node right after some event, the (non-primitive) modal operator knew talks about the knowledge of a node right before some event. Although the reader might notice that this lemma does not directly mention any variant of the disseminate operator, different variants of this operator are used in the axioms on which this lemma depends on in order to track down the piece of data *d* through the *eo* execution of the system.

The second lemma essentially states that if a correct node *n* (running at loc *e* below) just learned (on time but strictly before the *D* bound—a deadline) at some event *e* about some correct piece of data *d*, then any other correct node that is correct in *e*'s "near future" must get to know about *d* too in later "epochs". This lemma relies on AXIOM_messages_get_delivered (see Section 3.1.7), as well as on following Assumptions: (1), (3), (9), (10) and (11). Those are elided here for readability. We formally state this second lemma in Figure 4.3.

The proposition (just_learned_on_time *e d E D*) states that the node running at loc *e* learned on time about the piece of data *d* at *e* (i.e., learns_on_time *e d E D* is true), which it did not know before *e*, and such that *d*'s epoch number is strictly less

```
Lemma learns_on_time_imp_other_knew :
  ∀ {eo : EO} N K E D (e e' : EventN(N)) d,
    loc e <> loc e'
    → ¬ In (loc e) (byk_data2senders d)
    → ¬ In (loc e') (byk_data2senders d)
    → byk_verify e d = true
    → just_learned_on_time e d E D
    → has_correct_trace_before e (loc e)
    → is_correct_in_near_future (loc e') e
    → has_correct_trace_bounded_lt e'
    → K (sign_data d (loc e) (keys e))
    → events_in_later_epoch e e'
    → knew e' (sign_data d (loc e) (keys e)).
```

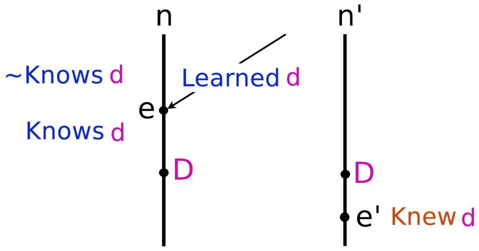


Figure 4.3: Knowledge lemma learns_on_time_imp_other_knew

than the deadline D. Finally, events_in_later_epoch e e' states that e' happened in later epoch than e's epoch.

4.5 Knowledge and SM

As we already mentioned above, we used ByK to prove the agreement property (i.e. the safety part of *IC1*) of two implementations of SM—here we focus on our implementation called SM$_{\mathsf{seq}}$. This property states that if two correct generals $g1$ and $g2$ decided upon some values $v1$ and $v2$, then those two values must be equal. To prove this lemma using our knowledge library, we first have to instantiate the parameters of KnowingClass, LearningClass, and AuthenticatedKnowledge. We instantiated KnowingClass and LearningClass as follows: byk_data is the type of SM messages; byk_info is the type of SM values; byk_mem is the type of states maintained by generals; byk_data2info extracts the value contained in an SM message; byk_knows states that some value v is stored in the vector V maintained by a general; byk_sys is a collection of state machines (i.e., SMsys, which we defined similarly to PBFTsys in Section 3.1.6); byk_data2owner extracts the general that generated a given SM message; byk_data2msg is essentially the identity function; while byk_data2auth extracts only authenticated piece of data created by the last general that signed a SM message, byk_data2auth_list extracts all authenticated pieces of data contained inside a SM message; and byk_verify is a function that verifies if SM message is signed properly. In addition, we instantiated AuthenticatedKnowledge's parameters as follows: byk_max_sign is $f + 1$, i.e. the number of signatures that need to be collected; byk_ext_info concatenates a signature to an SM value; byk_ext_data concatenates a signature to an SM message;

byk_data2data m n concatenates the signature of lieutenant n to a message m, that was already signed by some other lieutenant (as explained in Section 2.1.1.1, this function is used every time a lieutenant receives a message from another lieutenant, in order to create a message that will be disseminated to other lieutenants); byk_data2sign extract the last signature contained in an SM message. Finally, we instantiated byk_data2can inductively as follows:

```
Fixpoint sm_data2can (d : sm_data) : list (byk_data_or_info * Sign) :=
  match d with
    | sm_signed_msg_sign v s ⇒ [(byk_is_info v, s)]
    | sm_signed_msg_cons d' s ⇒ snoc (sm_data2can d') (byk_is_data d', s)
  end.
```

This sm_data2can function returns all the signed pieces of data contained in an SM message as a list of pairs of either (1) a value and a signature of that value, or (2) an SM message and a signature of that message.

Moreover, all the axioms presented in Section 4.1, Section 4.2 and Section 4.3 are true about these instantiations of KnowingClass, LearningClass and AuthenticatedKnowledge. For example, lemma `SM_preserves_knows` in `code/SM/IC1.v` shows that $\text{SM}_{\textsf{seq}}$ indeed satisfies axiom AXIOM_preserves_knows. The safety part of SM's *IC1* property is then a straightforward consequence of the lemmas presented in Section 4.4. Section 5.1 provides a sketch of the proof of this property.

4.6 Knowledge and PBFT

One of the key lemmas to prove PBFT's safety says that if two correct replicas have prepared some requests with the same sequence and view numbers, then the requests must be the same [Cas01, Inv.A.1.4]. As mentioned in Section 2.1.2.1, a replica has prepared a request if it received pre-prepare and prepare messages from a quorum of replicas. To prove this lemma, we instantiated KnowingClass and LearningClass as follows: byk_data can either be a pre-prepare or a prepare message; byk_info is the type of triples view/sequence number/digest; byk_mem is the type of states maintained by replicas; byk_data2info extracts the view, sequence number and digest contained in pre-prepare and prepare messages; byk_knows states that the pre-prepare or prepare message is stored in the state; byk_sys is a collection of state machines (see PBFTsys defined in Section 3.1.6); byk_data2owner extracts the sender of the message; byk_data2auth and byk_data2auth_list are similar to the PBFTget_contained_auth_data function presented in Section 3.1.3; and byk_verify is a function that verifies if pre-prepare/prepare message is signed properly. The two equations Equation (4.2) and Equation (4.3), which we proved using the tactic discussed in Section 3.2.1, are true about these instances of KnowingClass and LearningClass. Inv.A.1.4 is then a straightforward consequence of knows_in_intersection (see Section 4.2) applied to the two quorums.

Chapter 5

Homogeneous Protocol Case Studies: SM and PBFT

The main goal of this thesis is to design a general, reusable, and extensible framework that can be instantiated to prove the correctness of any synchronous or asynchronous BFT protocol. To show its usability, we proved crucial properties of the seminal synchronous BFT protocol called SM (see Section 5.1), as well as crucial properties of the seminal practical asynchronous BFT protocol called PBFT (see Section 5.2). We conclude this chapter by explaining how we obtain executable PBFT code, and by comparing our PBFT implementation with a state-of-the-art BFT library, called BFT-SMaRt [BSA14]. Although we verified implementations of both SM and PBFT, we only evaluated the performance of our PBFT implementation for the main reason that, as opposed to SM, it allows executing multiple instances of consensus. The core SM protocol allows executing a single instance of consensus only, and requires an additional layer to handle multiple instances, which is out of the scope of this thesis.

We start this section by explaining how we proved agreement of SM (see Section 5.1). Next, we explain how we proved agreement of PBFT (see Section 5.2) and how we obtained executable PBFT code (see Section 5.3). We finish this section with a brief discussion about our trusted computing base and proof efforts (see Section 5.4).

5.1 Verified SM properties

We demonstrate that framework presented in this thesis can be used to prove properties of synchronous BFT protocols by proving that both our implementations of the SM protocol satisfy the safety part of the *IC1* property, originally introduced in [LSP82] (see Section 2.1.1.2 for more details). This property states that any two values *v1* and *v2*, sent by correct generals *g1* and *g2* at events *e1* and *e2*, respectively, have to be equal.[1] We proved that this property is true in any event ordering that satisfies the assumptions introduced in Section 3.1.7, i.e:

```
Lemma IC1_safety :
  ∀ (eo : EO)(e1 e2 : Event) (v1 v2 : sm_value) (g1 g2 : Gen),
  AXIOM_authenticated_messages_were_sent_or_byz_sys eo SMsys
  → has_correct_trace_before e1 (loc e1)
  → has_correct_trace_before e2 (loc e2)
  → In (send_result g1 v1) (output_system_on_event SMsys e1)
  → In (send_result g2 v2) (output_system_on_event SMsys e2)
  → v1 = v2.
```

Note that for simplicity, we assume that each lieutenant sends the value it decided upon to itself using send_result.

Proof Sketch 1. Let us first consider the special case where *g1*=*g2*. In that case, *v1* and *v2* must be equal, because otherwise there would exist a correct general that decided twice upon two different values before the deadline.

Next, let us assume that *g1* and *g2* are different generals. If the values voted upon by the two generals are not the same, it must be that there exists one value in the vector of values of one of the generals, which is not in the vector of values of the other general. Without loss of generality, we will assume that *g1* received a value *v* that *g2* has not. We can then get back to the point when *g1* received that value *v*. From there we proceed by cases on whether or not *g1* received a signature from *g2* for *v*.

The case where *g2* signed *v* and sent it to *g1*, is a direct consequence of our disseminated_before_deadline knowledge lemma presented in Section 4.4.

Let us now consider the case where *g1* did not collect a signature from *g2* for *v*. We will again go by cases here, this time on whether or not the message containing *v* that *g1* received, contained at least $f + 1$ signatures.

If that message contained strictly less than $f + 1$ signatures, upon processing that message, *g1* will sign it and disseminate it to the generals that did not yet sign it, including *g2*, which will therefore receive it. This case is a direct consequence of lemma learns_on_time_implies_other_knew presented in Section 4.4.

If the message containing *v* received by *g1* contained at least $f + 1$ signatures, it must contain exactly $f + 1$ signatures by design (see Section 2.1.1.1). Therefore, there must be at least one correct general that signed *v*. That correct general might be either a lieutenant or the commander. The case when that correct general is a lieutenant is a consequence of our knowledge lemma disseminated_before_deadline (see Section 4.4).

---

[1] See agreement in `code/SM/IC1.v`.

Finally, we derive the case when that correct general is the commander directly within ByLoE. In this case we reason as follows: if general *g1* learned piece of data *d*, and commander owns that piece of data, then there exists a prior event *e0* at which commander disseminated that piece of data. In case of SM protocol, event *e0* is initial event. Moreover, if a commander disseminated a piece of data *d* to the general *g1*, then it also disseminated the same piece of data to all other correct generals, including general *g2*. Based on this, we can prove that general *g2* actually learned data *d* on time, which implies that general *g2* knows data *d* before the deadline.

5.2 Verified PBFT properties

We demonstrated that framework presented in this thesis can be used to prove properties of asynchronous BFT protocol by proving that our PBFT implementation satisfies the standard agreement property, which is the crux of linearizability (see Section 2.1.2.2 for a high-level definition). Agreement states that, regardless of the view, any two replies sent by correct replicas *i1* and *i2* at events *e1* and *e2* for the same timestamp *ts* to the same client *c* contain the same replies. We proved that this is true in any event ordering that satisfies the assumptions from Section 3.1.7:[2]

```
Lemma agreement :
  ∀ (eo : EO) (e1 e2 : Event) (v1 v2 : View) (ts : Timestamp)
        (c : Client) (i1 i2 : Rep) (r1 r2 : Request) (a1 a2 : list Token),
  AXIOM_authenticated_messages_were_sent_or_byz_sys eo PBFTsys
  → AXIOM_correct_keys eo
  → AXIOM_exists_at_most_f_faulty [e1,e2] f
  → loc e1 = PBFTreplica i1
  → loc e2 = PBFTreplica i2
  → In (send_reply v1 ts c i1 r1 a1) (output_system_on_event PBFTsys e1)
  → In (send_reply v2 ts c i2 r2 a2) (output_system_on_event PBFTsys e2)
  → r1 = r2.
```

where Timestamps are nats; and send_reply builds a reply message. To prove this lemma, we proved most of the invariants stated by Castro in [Cas01, Appx.A]. In addition, we proved that if the last executed sequence number of two correct replicas is the same, then these two replicas have, among other things, the same service state.[3]

As mentioned above, because our model is based on ByLoE, we only ever prove such properties by induction on causal time. Similarly, Castro proved most of his

---

[2] See agreement in `code/PBFT/PBFTagreement.v`.

[3] See same_states_if_same_next_to_execute in `code/PBFT/PBFTsame_states.v`.

invariants by induction on the length of the executions. However, he used other induction principles to prove some lemmas, such as Inv.A.1.9, which he proved by induction on views [Cas01, p.151]. This invariant says that prepared requests have to be consistent with the requests sent in pre-prepare messages by the primary. A straightforward induction on causal time was more natural in our setting.

Castro used a simulation method to prove PBFT's safety: he first proved the safety of a version without garbage collection and then proved that the version with garbage collection implements the one without. This requires defining two versions of the protocol. Instead, we directly proved the safety of the one with garbage collection. This involved proving further invariants about stored, received and sent messages, essentially that they are always within the water marks.

5.3 PBFT Extraction and Evaluation

We start this section by explaining how we obtain executable OCaml code from our Coq implementations. Next, we compare the performance of our PBFT implementation with the one of the BFT-SMaRt state-of-the-art library [BSA14]. We end this section by discussing our trusted computing base.

Extraction. To evaluate our PBFT implementation (i.e., PBFTsys defined in Section 3.1.6—a collection of state machines), we generate OCaml code using Coq's extraction mechanism. Most parameters, such as the number of tolerated faults, are instantiated before extraction. Note that not all parameters need to be instantiated. For example, as mentioned in Section 3.1.1, neither do we instantiate our assumptions (such as AXIOM_exists_at_most_f_faulty), nor do we instantiate event orderings, because they are not used in the code but are only used to prove that properties are true about all possible runs. Also, keys, signatures, and digests are only instantiated by stubs in Coq. We replace those stubs when extracting OCaml code by implementations provided by the nocrypto [19f] library, which is the cryptographic library we use to hash, sign, and verify messages (we use RSA).

Evaluation. To run the extracted code in a real distributed environment, we implemented a small trusted runtime environment in OCaml that uses the Async library [19a] to handle sender/receiver threads. We show among other things here that the average latency of our implementation is acceptable compared to the state of the art BFT-SMaRt [BSA14] library (which is not formally verified). Note that because we do not offer a new protocol, but essentially a re-implementation of PBFT, we expect that on average the scale will be similar in other execution scenarios such as the ones studied by Castro in [Cas01]. We ran our experiments using desktops with 16GB of memory, and 8 i7-6700 cores running at 3.40GHz.

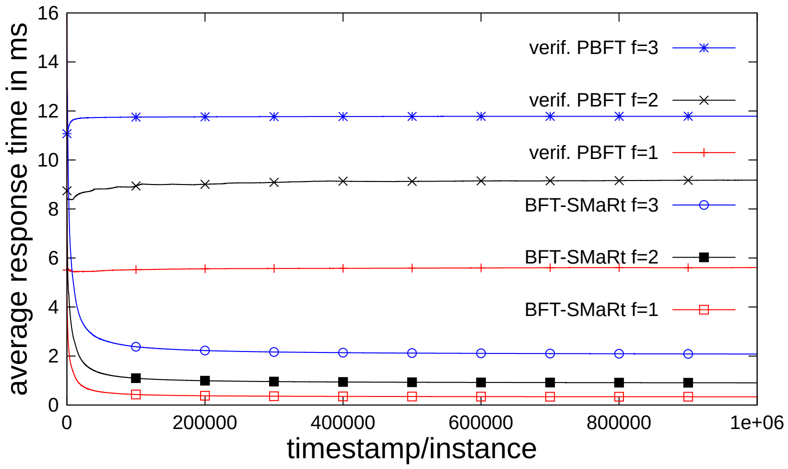


Figure 5.1: Performance of verified PBFT ($f \in \{1,2,3\}$) on a single machine

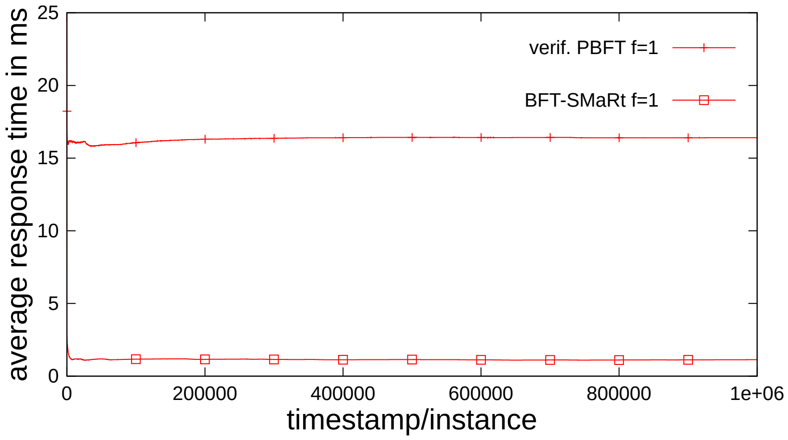


Figure 5.2: Performance of verified PBFT ($f = 1$) on several machines

We report some of our experiments where we used a single client, and a simple state machine where the state is a number, and an operation is either adding or subtracting some value.[4]

We ran a local simulation to measure the performance of our PBFT implementation without network and signatures: when 1 client sends 1 million requests, it takes on average 1.7 microseconds for the client to receive $f+1$ ($f=1$) replies.

Figure 5.1 shows the experiment where we varied f from 1 to 3, and replicas

---

[4] We chose to implement a simple operation, because execution of a complex operation on a state machine would significantly influence on the performance of PBFT.

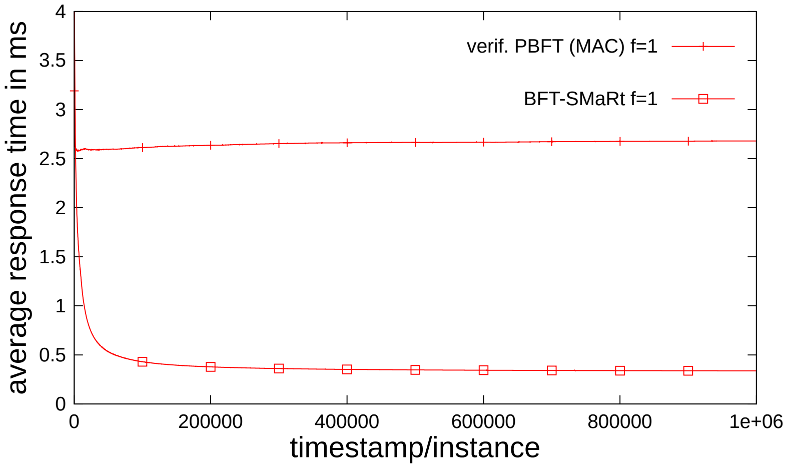


Figure 5.3: Performance of verified PBFT based on MACs on a single machine

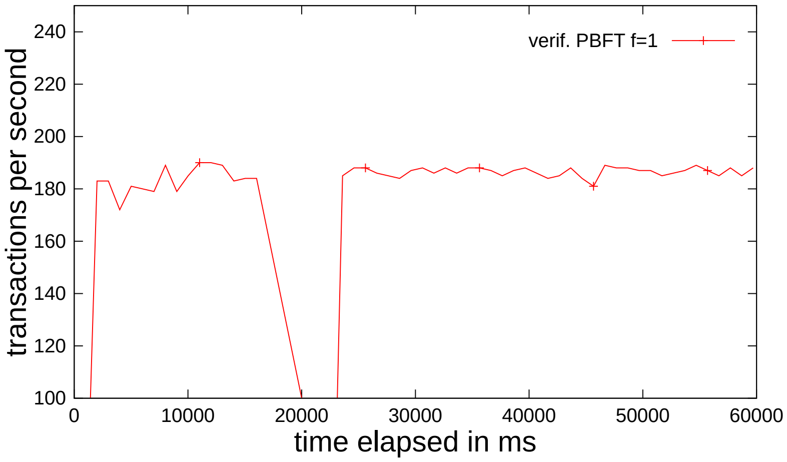


Figure 5.4: Performance of verified PBFT in case of a view-change

sent messages, signed using RSA, through sockets, but on a single machine. As mentioned above, we implemented the digital signature-based version of PBFT, while BFT-SMaRt uses a more efficient MAC-based authentication scheme, which in part explains why BFT-SMaRt is around one order of magnitude faster than our implementation. As in [Cas01, Tab.8.9], we expect a similar improvement when using the more involved, and as of yet not formally verified, MAC-based version of PBFT (Figure 5.3 shows the average response time when replacing digital signatures by MACs, without adapting the rest of the protocol). Figure 5.2 presents results when running our version of PBFT and BFT-SMaRt on several machines, connected via gigabit Ethernet network, for $f = 1$. Finally, Figure 5.4 shows the response time of our view-change protocol. In this experiment, we killed the primary after 16 sec of execution, and it took around 7 sec for the system to recover.

5.4 Homogeneous Protocols: TCB & Proof Effort

Trusted Computing Base. The TCB of our system includes: (1) the fact that our ByLoE model faithfully reflects the behavior of distributed systems (see Section 3.1.5); (2) the validity of the assumptions discussed in Section 3.1.7; (3) Coq's logic and implementation; (4) OCaml and the nocrypto and Async libraries we use in our runtime environment, and the runtime environment itself (Section 5.3); (5) the hardware and software on which our framework is running.

Proof Effort. In terms of proof effort, developing parts of framework presented in Chapters 3, Chapter 4 and Chapter 5 and verifying agreement properties of SM and PBFT took us around 1 and a half person years. Our generic framework consists of around 4500 lines of specifications and around 4000 lines of proofs. Each of our verified implementations of SM consists of around 2000 lines of specification and around 2000 lines of proofs. Our verified implementation of PBFT consists of around 20000 lines of specifications and around 22000 lines of proofs.

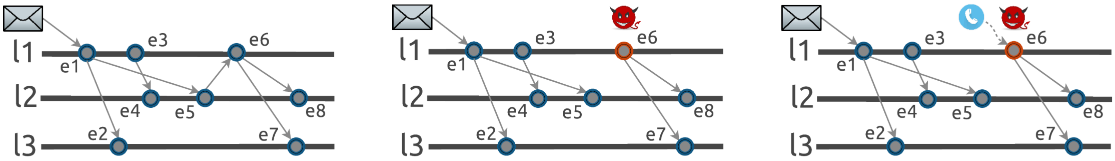
Chapter 6

Verification of Hybrid BFT protocols

As explained above, hybrid Byzantine fault-tolerant distributed protocols significantly reduce the message/time/space complexity of BFT-SMR protocols. Unfortunately, there is no lunch for free, i.e., these protocols are way more structurally complex than homogeneous ones. In Chapter 3, we presented: (1) a general model of processes, where each local subsystem is a state machine; and (2) a Byzantine logic of events that supports arbitrary (Byzantine) events, i.e., events for which no information is available, which could for example have been triggered by malicious or corrupted nodes. As is turns out, model and logic presented in Chapter 3 cannot be used for formal verification of hybrid Byzantine fault-tolerant protocols, for several reasons.

First of all, while homogeneous protocols assume that a local state machine is essentially a single component, hybrid protocols assume that a local machine can be composed of any number of components. Moreover, while homogeneous protocols assume that all its participants have the same system and failure assumptions, hybrid protocols assume that some parts of a system can rely on stronger assumptions compared to the rest of the system (e.g., some components can be tagged as *trusted*, while all the other components are *non-trusted* ones). Unfortunately, the model presented in Section 3.1 does not provide support for compositional programming and reasoning, nor for tagging participants as trusted. This motivated us to developed a Monadic Component language (MoC). MoC allows implementing distributed systems as collections of local systems, which are themselves collections of components, some of them being tagged as trusted. In addition, MoC enables lifting properties of trusted components to the level of a local state machine, via deep embeddings of fragments of MoC.

To capture the behavior of trusted components, we also had to modify logic of events presented in Section 3.1, to allow the non-trusted components of processes



(a) Correct (kind 1) (b) Byzantine (kind 2a) (c) Hybrid (kind 2b)

Figure 6.1: Examples of message sequence diagrams

to misbehave, while the trusted components keep following their specification. We captured this by changing the semantics of events (namely the trigger function described in Section 3.1.5) to also handle events at which a trusted component of a compromised process is called. This led us to developing the a Hybrid Logic of Events (HyLoE).

In addition to reasoning about hybrid systems, using framework presented in this chapter one can also reason about homogeneous Byzantine systems by not using trusted components, and about crash fault tolerant systems by assuming that there are no Byzantine events (see Section 6.1).

We start this chapter by presenting our Hybrid Logic of Events (HyLoE) (see Section 6.1) which we use to reason about hybrid fault model, and by comparing it to our Byzantine Logic of Events (ByLoE) presented in Section 3.1.1. Next, in Section 6.2 we present our Monadic Component language (MoC), which we use to implement systems as collections of interacting hybrid components, and in Section 6.3, we explain how to relate the execution of systems with event orderings. Finally, in Section 6.4 we explain how to reason about systems compositionally by lifting properties of components of a local subsystem to the level of that subsystem. In this chapter we use MinBFT (see Section 2.1.3) as a running example.

Notation. We use Event to represent a set of events; AuthData to represent a set of authenticated pieces of data; and Keys to represent a set of keys.

6.1 HyLoE: A Hybrid Logic of Events

As explained in Section 3.1.1, one of the most fundamental concepts to reason about distributed systems is the concept of an *event*, which can be seen as a point in space/time at which something happened. In ByLoE, an event is either an abstract objects that only correspond to the handling of a message by a node that follows its specification (kind 1—see Figure 6.1a), or it corresponds to some arbitrary behavior, in which case no further information regarding this event is available/provided (kind 2—see Figure 6.1b). HyLoE further extends ByLoE by providing means to reason about three kinds of events. As in ByLoE, HyLoE supports events of kind 1 (see the constructor `TImsg` below). Furthermore, the kind 2 events of ByLoE, that are happening at a compromised node, are now split into two categories: (1) those that did not call a trusted component, and therefore for which no information is available (kind 2a—see Figure 6.1b and `TIarbitrary` below); and (2) those that called a trusted component (kind 2b—see Figure 6.1c and `TItrust` below). Assuming that trusted components only receive inputs of some abstract type InputTrusted, ranged over by it, we introduce the type:

$$nfo \in \mathsf{TriggerInfo} ::= \mathtt{TImsg}(msg) \mid \mathtt{TItrust}(it) \mid \mathtt{TIarbitrary}$$

Also, as explained in Section 3.1.1, to prove a property about a distributed system, one has to reason about *all its possible execution traces*. Therefore, we need to provide a model of those traces. As in ByLoE, we model a run of a distributed system essentially as a partial order on events. Such an abstract representation of a run is called an *event ordering*. Therefore, to prove a property P about a distributed system, one has to prove that P is true for all event orderings that correspond to this system (among other things, all possible assignments of TriggerInfos to events have to be considered).

Figure 6.1 provides examples of message sequence diagrams. Figure 6.1a, depicts an event ordering with three locations l_1, l_2, l_3, where all events are correct and are triggered by messages. Because here the network is asynchronous, even though l_1 sent a message to l_2 at event e_1 before it sent a message to l_2 at e_3, l_2 received the first message at e_5 after it received the second message at e_4. In this figure, e_6 is triggered by the receipt of a message sent by l_2 at e_5. Instead, in Figure6.1b, e_6 is a Byzantine event for which no information is available and at which no trusted component was called; and in Figure 6.1c, e_6 is a hybrid event at a Byzantine location and at which a trusted component was called.

6.2 MoC: Component-Based Programming

As explained above, to enable reasoning about distributed systems, where local sub-systems are composed of multiple components that can have different failure assumptions, we developed MoC.

In MoC, components are referred to by their names. Let CompName be the set of component names, ranged over by cn. A component name is a triple of: (1) a kind (a string), which is used to enforce constraints on the I/O behavior of components (e.g., the "MSG" kind enforces that the component must receive messages and output directed messages); (2) a space (a number) allowing different components to be of the same kind (e.g., have the same I/O behavior), while maintaining states of different types (e.g., a component of kind "MSG" and space

0 might be maintaining a state of type $\mathbb{N}$, while a component of kind "MSG" and space 1 might be maintaining a state of type $\mathbb{B}$); and (3) a tag (a Boolean) describing whether the component is trusted (trusted components are constrained to only react to inputs of type InputTrusted—see Section 6.1). Moreover, a component's name specifies its behavior: we assume some functions $\mathcal{S}$, $\mathcal{I}$, and $\mathcal{O}$ from component names to types, which enforce that a component named cn must have a state of type $\mathcal{S}(cn)$; take inputs of type $\mathcal{I}(cn)$; and produce outputs of type $\mathcal{O}(cn)$.

6.2.1 Computational Model of MoC

In this section, we first introduce MoC components and explains how they interact through a monad. Then, we explain how to build local/distributed systems as collections of components.

Components. A component is a named state machine, which essentially consists of an update function and the current state of the machine. To support the fact that components are allowed to call each other, we define state machines using a state monad [Mog89]. Therefore, instead of traditionally defining update functions as functions that take an input and a state and return an output and an updated state, we combine those with a monad (see $\mathsf{M}^n(T)$'s definition below), such that in addition update functions take components as input and return possibly modified components. Consequently, state machines can call other state machines through this state monad. Therefore, to avoid a circularity in the definition of state machines, we now use step indexing [DAB11] to define them, requiring that machines at level n can only use machines of lower levels. Let $\mathsf{Component}^n$ (ranged over by $comp$) be the collection of components at level n, which we define recursively over n below. This definition uses the monad mentioned above, which looks like this (where T is a type):

$$\mathsf{M}^n(T) = \mathsf{list}(\mathsf{Component}^n) \to (\mathsf{list}(\mathsf{Component}^n) * T)$$

Going back to state machines, a machine at level $n+1$ (of type $\mathsf{Component}^{n+1}$—by definition there are no level 0 machines) with name cn is either a state machine at level n, or a pair of: (1) an update function of type $\mathsf{Upd}^n(cn) = \mathcal{S}(cn) \to \mathcal{I}(cn) \to \mathsf{M}^n(\mathcal{S}(cn) * \mathcal{O}(cn))$; and (2) a state of type $\mathcal{S}(cn)$.[1]

Monad operators. As mentioned above, the components of a local subsystem interact through a monad. MoC's monad provides three main operators to build a component: (1) a *return* operator (ret) to turn a Coq expression into a component;

---

[1] State machines also have the ability to halt on their own. However, we do not discuss this feature here for simplicity.

(2) a *bind* operator ($\gg=$) to compose two components; and (3) a *call* operator (call) to allow components to call their components. Those operators can be combined in any way one wants using any Coq function one desires, as long as the resulting code has the right *component* type. The return and bind operators of our (state) monad are defined as usual: $\mathsf{ret}(a) = \lambda s.\langle s, a\rangle$ takes a $a \in A$ and outputs a $\mathsf{M}^n(A)$; and $m \gg= f = (\lambda s.\texttt{let}\ s', a = m(s)\ \texttt{in}\ f(a, s'))$ takes a $m \in \mathsf{M}^n(A)$ and a $f \in A \to \mathsf{M}^n(B)$ and outputs a $\mathsf{M}^n(B)$. A call operator takes a component name cn and an input $i \in \mathcal{I}(cn)$ and returns a monadic output of type $\mathsf{M}^n(\mathcal{O}(cn))$. It first looks for a component with name cn within its components $subs$, provided by the returned monad. If it finds one, say $comp$, it then applies $comp$ to the input i and to the subset $subs_1$ of $subs$ containing the components of levels strictly lower than n (the only components that $comp$ can use because of its level). This computation produces an output o and a list of updated components $subs_2$. Finally, call returns the output o, as well as the list of components $subs$, where $subs_1$ is replaced by $subs_2$.

Local & Distributed Systems. A *local subsystem* of type LocalSystem is a pair of a main component at level n and a list of components at lower levels, such that some of these components can be tagged as trusted. The level of a local subsystem is the level of its main component. We enforce that main components send and receive messages. A (distributed) *system* of type System is a function from node names to local subsystems, i.e., of type Node → LocalSystem.

Case Study. In our MoC implementation of MinBFT (see `code/MinBFT/MinBFT.v` for more details), a replica is a local subsystem called MinBFTlocalSys. Each local subsystem is composed of: (1) a main component (called MAINcomp), which among other things maintains the replicated service; (2) a USIG component (called USIGcomp—the only trusted component) as described in Section 2.1.3.1; and (3) a log component (called LOGcomp) that stores all sent and received messages. Finally, the distributed system MinBFTsys is the function mapping each replica name to MinBFTlocalSys.

6.2.2 Example: A Simple MoC Local Subsystem

We provide here a simple example of a local subsystem composed of multiple components, implemented using MoC (see the file called `code/model/ComponentSMExample2.v` in our implementation). The local subsystem is composed of (1) a trusted level 1 component called $\mathsf{ST} = \langle\texttt{"STATE"}, 0, \texttt{true}\rangle$, which maintains a state—simply a number here; (2) of two other non-trusted level-2 components, one to add a value once to the state called $\mathsf{OP1} = \langle\texttt{"OP"}, 0, \texttt{false}\rangle$, and one to add a value twice from the state called $\mathsf{OP2} = \langle\texttt{"OP"}, 1, \texttt{false}\rangle$; and finally (3) of a main level-3 component, called $\mathsf{M} = \langle\texttt{"MSG"}, 0, \texttt{false}\rangle$, that dispatches incoming messages to either of the "OP" components. Because "STATE" is the only stateful component, all the other components maintain a trivial state of type Unit, which is a singleton type inhabited by $\texttt{tt}$. Messages are of the form $\mathsf{ADD1}(n)$, or $\mathsf{ADD2}(n)$, or $\mathsf{TOTAL}(n)$, where $n \in \mathbb{N}$. Let ST's update function be defined as follows:

$$\lambda s, i.\mathsf{ret}(\langle s + i, s + i\rangle)$$

Let $\mathsf{OP1}$'s update function be defined as follows:

$$\lambda s, i.\mathsf{call}(\langle\texttt{"STATE"}, 0, \texttt{true}\rangle, i) \ggg\!= \lambda o.\mathsf{ret}(\langle\texttt{tt}, o\rangle)$$

Let $\mathsf{OP2}$'s update function be defined as follows:

$$\begin{aligned}\lambda s, i.\ & \mathsf{call}(\langle\texttt{"STATE"}, 0, \texttt{true}\rangle, i)\\ & \quad \ggg\!= \lambda\_.\mathsf{call}(\langle\texttt{"STATE"}, 0, \texttt{true}\rangle, i) \ggg\!= \lambda o.\mathsf{ret}(\langle\texttt{tt}, o\rangle)\end{aligned}$$

Finally, the update function of M is defined as follows:

$$\begin{aligned}\lambda s, i.\ \ & \texttt{match}\ i\ \texttt{with}\\ & \mid \mathsf{ADD1}(n) \Rightarrow \mathsf{call}(\langle\texttt{"OP"}, 0, \texttt{false}\rangle, n)\\ & \mid \mathsf{ADD2}(n) \Rightarrow \mathsf{call}(\langle\texttt{"OP"}, 1, \texttt{false}\rangle, n)\\ & \mid \mathsf{TOTAL}(n) \Rightarrow \mathsf{ret}(n)\\ & \texttt{end}\\ & \ggg\!= \lambda o.\mathsf{ret}(\langle\texttt{tt}, [\langle\mathsf{TOTAL}(o), []\rangle]\rangle)\end{aligned}$$

where $\langle\mathsf{TOTAL}(o), []\rangle$ is a directed message, in this case, the instruction to send the message $\mathsf{TOTAL}(o)$ to the empty list of recipients $[]$.

Whenever this local subsystem receives a message m, it applies M's update function to m and to the list of its three components $\mathsf{OP1}$, $\mathsf{OP2}$, and ST. If m is, for example, of the form $\mathsf{ADD1}(n)$, then M calls $\mathsf{OP1}$ on the input n. This results in looking for a component with that name in the list of M's components. Because such a component exists, namely $\mathsf{OP1}$, we create a new local subsystem with main component $\mathsf{OP1}$ and component ST (the only component with level lower than $\mathsf{OP1}$'s). We then apply $\mathsf{OP1}$'s update function to n and to the list containing its single component, namely ST. This results in calling ST on the input n. Because ST is present in the list of $\mathsf{OP1}$'s components, we then create a new local subsystem with main component ST and no components (because there are no components with level lower than ST's), and we apply this system to n. This results in applying ST's update function to n and to the empty list of components. If this call is the first call, and if ST's initial state is 0, then its update function returns the new state n and outputs n. It also returns the empty list of components that it took

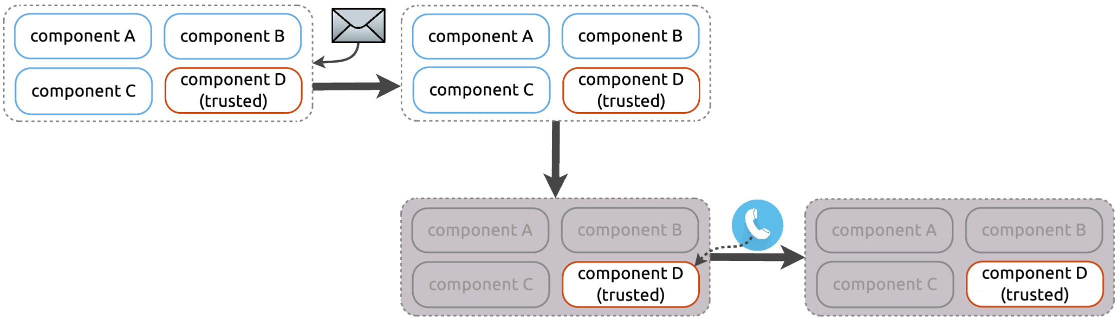


Figure 6.2: An execution of a local subsystem

as input. Going back to $\mathsf{OP1}$, we then update the state of its component ST from 0 to n. Finally, $\mathsf{OP1}$ return this updated list of components, it outputs the value n, and its state remains $\mathtt{tt}$. Going back to M, we then update the state of $\mathsf{OP1}$ from $\mathtt{tt}$ to $\mathtt{tt}$, and we replace the component ST with state 0 that M took as input, with the one with state n that $\mathsf{OP1}$ returned. Finally, M returns the list of updated components $\mathsf{OP1}$ with state $\mathtt{tt}$; $\mathsf{OP2}$ with state $\mathtt{tt}$ (which it did not call here); and ST with state n. M also returns the state $\mathtt{tt}$ and outputs a single directed message: $\langle \mathsf{TOTAL}(n), [] \rangle$.

If the input had been of the form $\mathsf{ADD2}(n)$ then instead we would have called ST twice in a row. This would have resulted in $\mathsf{OP2}$ updating twice its list of components (containing only ST). The first time, ST's state would have been updated to n, and the second time, because ST's state would have then be n, ST's state would have then been updated to $n + n$.

6.3 Relating MoC Systems and HyLoE Events

As mentioned above, to prove a property about a distributed system S, one has to prove that this property holds for all “possible” event orderings. Therefore, given an event ordering eo, one has to be able to compute the inputs, outputs, and states of S's local sub-systems at all events in eo in order to reason about S's “trace” provided by eo. Inputs are provided by the trigger function. We now explain how to compute outputs and states, and provide an example showing how to combine these definitions to prove systems' properties in a compositional manner.

Computing systems' states. First, $ls@^{-}e$ runs the local subsystem ls by applying its main component to its components and to the list of events locally preceding e and excluding e (similarly, $ls@^{+}e$ computes ls's state after e, by applying ls to the list of events locally preceding e, including e). It either (1) returns a local subsystem ls' if all those events have been triggered by information of the form

`TImsg`(msg), i.e., non-Byzantine events; or (2) it returns a trusted component in case at least one of those preceding events has been triggered by some information of the form `TItrust`(it) (in case the trusted component[2] is called) or `TIarbitrary` (in case the trusted component is not called), in which case some Byzantine event happened, and we cannot know what state the rest of the local subsystem is in; or (3) it is undefined if one of those preceding events is a Byzantine event and ls does not include a trusted component. Figure 6.2 provides an example of the status of the components of a local subsystem (composed of 3 non-trusted blue components and a trusted orange one) after handling the events caused by: (1) the receipt of a message; (2) some arbitrary behavior; and (3) a call to the trusted component D. As shown in Figure 6.2, in case one of those preceding events is Byzantine, $ls@^{-}e$ keeps on running the trusted component because it cannot be compromised. However, $ls@^{-}e$ loses track of the rest of the system since a Byzantine event has occurred, and the other non-trusted components could be in any state.

Computing components' states. We can then access the state of a component named cn of a local subsystem ls using the operator $ls\lfloor_{cn}$. Also, let $comp\lfloor_{cn}$ be $comp$ if it has name cn, and undefined otherwise. Therefore, $ls@^{-}e\lfloor_{cn}$ returns the state of ls's component called cn before the event e (if it exists, i.e., if the component is trusted or no Byzantine event has occurred, otherwise the component could be in any state); and similarly for $ls@^{+}e\lfloor_{cn}$. Finally, we can compute the state of a component cn of a system S before a given event e simply by calling $S(\mathrm{loc}(e))@^{-}e\lfloor_{cn}$, which we write as $S@^{-}e\lfloor_{cn}$, and similarly for after the event.

Computing systems' outputs. Let $ls \leadsto e$ be the outputs produced by ls's main component at e, when all the events preceding e are non-Byzantine (these outputs are obtained by running the system on $ls@^{-}e$). In case one of those events is Byzantine, $ls \leadsto e$ produces instead the outputs of the trusted component, which we are keeping track of (as explained above). We write $S \leadsto e$ for $S(\mathrm{loc}(e)) \leadsto e$; and $d \in ls \leadsto e$ to mean that d occurs within the outputs computed by $ls \leadsto e$.

As illustrated below, framework presented in this thesis allows composing the specifications of components to derive local and distributed system specifications, which are fully specified in terms of (1) their states using $S@^{-}e\lfloor_{cn}$ and $S@^{+}e\lfloor_{cn}$; (2) their inputs using trigger; and (3) their outputs using $S \leadsto e$.

Case Study. As shown in Section 6.2.1, MinBFTlocalSys is a distributed system

---

[2] For simplicity, we currently only support systems with at most one trusted component per local system—the typical case in the literature on hybrid systems. This can easily be extended to systems with multiple trusted components if needed.

composed of local sub-systems, each of which is composed of three components called main, log, and usig. Let us prove that if $\mathtt{accept}(r, i) \in \text{MinBFTlocalSys} \rightsquigarrow e$, i.e., if a backup accepts a request r with sequence number i, then r is logged, i.e., it is in $\text{MinBFTlocalSys}@^{+} e\lfloor_{\mathsf{log}}$. First, (1) we prove that whenever log is called, it logs the commit given as input. We prove this about the local subsystem composed of log only (which does not use any components). Then, (2) from $\mathtt{accept}(r, i) \in \text{MinBFTlocalSys} \rightsquigarrow e$, we obtain that this output, as well as $\text{MinBFTlocalSys}@^{+} e$, was produced by running MinBFTlocalSys on $\text{MinBFTlocalSys}@^{-} e$. We then inspect the code run by MinBFTlocalSys, and we see that log, through the use of call, was requested to log a commit containing r. Finally, (2) we compose this proof with the one in step (1), and conclude by showing that $\text{MinBFTlocalSys}@^{+} e\lfloor_{\mathsf{log}}$ is the new state computed in step (1).

6.4 “Deep” Restrictions

We now describe a compositional method to lift properties proved about (trusted) components of a local subsystem to the level of that subsystem. One advantage of MoC is its expressiveness and flexibility: one can build a component essentially from any update function of type $\mathsf{Upd}^{n}(cn)$. Indeed, our framework provides a shallow embedding of components that can make use of any Coq expression as long as it has the right type. Unfortunately, this is also sometimes a disadvantage because it entails that we cannot prove many general lemmas about the behavior of components. For example, a component could simply throw away all its components. However, often components simply use their components, and return them updated. This is useful information, which we would like to easily derive. A standard technique to prove such generic results about such “well-behaved” components is to: (1) define a deep embedding of these “well-behaved” components; (2) define an “interpretation” function from the deep embedding to the shallow embedding; and (3) prove that these generic properties hold for the deep embedding.

One can define as many deep embeddings as needed. We define here a simple one (which we used to implement MinBFT) that contains only three operators: return/bind/call. Namely, let $\mathsf{Proc}(A)$ be the set of terms p of the following form:

$$\begin{array}{lll}
\mathtt{RET}(a) & \text{where} & a \in A \\
\mathtt{BIND}(p_1, p_2) & \text{where} & p_1 \in \mathsf{Proc}(B)\ \&\ p_2 \in B \rightarrow \mathsf{Proc}(A) \\
\mathtt{CALL}(cn, i) & \text{where} & i \in \mathcal{I}(cn)\ \&\ \mathcal{O}(cn) = A
\end{array}$$

We straightforwardly interpret this language as follows (this defines the function

$\mathbb{I}$ from $\mathsf{Proc}(A)$ to $\mathsf{M}^n(A)$, and this for any level n):

$$\begin{array}{lcl} \mathbb{I}(\mathtt{RET}(a)) & = & \mathsf{ret}(a) \\ \mathbb{I}(\mathtt{CALL}(cn, i)) & = & \mathsf{call}(cn, i) \\ \mathbb{I}(\mathtt{BIND}(m, f)) & = & \mathbb{I}(m) \gg\!= \lambda x.\mathbb{I}(f(x)) \end{array}$$

Then, given a component name cn, a level n (indicating what components cn will be able to use—it will only be able to use lower-level components), and a "deep" update function $u \in \mathcal{S}(cn) \to \mathcal{I}(cn) \to \mathsf{Proc}(\mathcal{S}(cn) * \mathcal{O}(cn))$, we can build a "shallow" update function of type $\mathsf{Upd}^n(cn)$ using $\lambda s, i.\mathbb{I}(u\ s\ i)$. Thanks to this language, we can now prove the preservation lemma mentioned above, i.e., that when a component is applied to components $subs_1$ then it produces components $subs_2$ such that $subs_1$ and $subs_2$ only differ by their states—components cannot be thrown away or spawned and the names and update functions remain the same.

Most importantly, this language allows us to reason compositionally about local and distributed systems (see Section 6.2.1). For example, we proved the following general result,[3] which we in turn used to prove that our MinBFT implementations satisfy the `Mon` property presented in Equation 7.4 in Section 7.6:[4]

Theorem 1 (Local Lifting)**.** *Given a local subsystem* ls*, if (1) all its components are built as above and have different names; and (2)* cn *is a trusted level* 1 *component in* ls *(i.e., it does not call other components); then for all event* e*, there must exist a list of inputs* $l \in \mathsf{list}(\mathcal{I}(cn))$ *such that the state* $ls@^{+}e\lfloor_{cn}$ *is obtained by running* cn *on* l*, starting from the state* $ls@^{-}e\lfloor_{cn}$*.*

Remark 1. Trusted components do not need to be at level 1. However, this constraint in Theorem 1 is convenient to obtain a simple lifting theorem. Otherwise, without this constraint, i.e., for higher-level components, such a local lifting theorem would be more complicated because it would have to also take into consideration the components such higher-level components use to compute their new state. More precisely, it would not be enough to run the subsystem ls' composed of cn and its components $subs$ (the components of ls that cn relies on) because the execution of ls on an event e might involve other components than those in ls'. Those other components might also call some of the components in $subs$. In that case it might not be enough to call ls' on a list of inputs to get to $ls@^{+}e\lfloor_{cn}$, because in between each call, we might have to also update the states of the components $subs$. It is worth noting that all the "standard" trusted components used in the literature [Chu+07]; [Lev+09]; [Ver+13] are level 1 components.

---

[3] See the lemma called `M_byz_compose_step_trusted` in the file called `code/model/ComponentSM3.v` in our implementation.

[4] See `ASSUMPTION_monotonicity_true` in `code/MinBFT/MinBFTass_mon.v` and `code/MinBFT/TrIncass_mon.v`.

The level 1 constraint is somewhat similar to the condition on the frame rule in separation logic, which requires the heap to be split into two disjoint resources. Here the resources are the components. We could allow trusted components to have components, as long as they're not used in the rest of the system. We could also have a more general lemma that in general allows splitting a system into two disjoint subsystems.

6.5 Spawning Components

The simple language presented in the section above is not the only choice. Note however that it is enough for a large number of protocols. To allow other features, one can simply introduce extensions of this language. For example, to allow spawning sub-processes, one could define the following language (see the file called `code/model/ComponentSM5.v` in our implementation): let $\mathsf{SpawnProc}(A)$ be the set of terms sp of the form:

$$\begin{array}{lll}
\texttt{SRET}(a) & \text{where} & a \in A \\
\texttt{SBIND}(sp_1, sp_2) & \text{where} & sp_1 \in \mathsf{SpawnProc}(B) \\
 & \quad\& & sp_2 \in B \to \mathsf{SpawnProc}(A) \\
\texttt{SCALL}(cn, i) & \text{where} & i \in \mathcal{I}(cn) \;\&\; \mathcal{O}(cn) = A \\
\texttt{SSPAWN}(cn, u, s, a) & \text{where} & u \in \mathcal{S}(cn) \to \mathcal{I}(cn) \to \mathsf{SpawnProc}(\mathcal{S}(cn) * \mathcal{O}(cn)) \\
 & \quad\& & s \in \mathcal{S}(cn) \;\&\; a \in A
\end{array}$$

We can interpret this language as follows (this defines the function $\mathbb{I}$ form $\mathsf{SpawnProc}(A)$ to $\mathsf{M}^n(A)$, and this for any level n):

$$\begin{array}{lcl}
\mathbb{I}(n, \texttt{SRET}(a)) & = & \mathsf{ret}(a) \\
\mathbb{I}(n, \texttt{SBIND}(m, f)) & = & \mathbb{I}(n, m) \gg\!= \lambda x.\mathbb{I}(n, f(x)) \\
\mathbb{I}(n, \texttt{SCALL}(cn, i)) & = & \mathsf{call}(cn, i) \\
\mathbb{I}(n, \texttt{SSPAWN}(cn, u, s, a)) & = & \;\texttt{if}\ n \;=\; 0\ \texttt{then}\ \mathsf{ret}(a) \\
 & & \;\texttt{else}\ \mathsf{spawn}(\lambda s, i.\mathbb{I}(n-1, (u\ s\ i)), s, a)
\end{array}$$

where spawn is a new monadic operator defined as follows ($\mathsf{mkComp}(u, s)$ builds a component from an update function u and a state s):

$$\mathsf{spawn}(u, s, a) = \lambda subs.\langle \mathsf{mkComp}(u, s) :: subs, a\rangle$$

One simple property that one can for example derive about components built this way is that when a component is applied to components $subs_1$ then it produces components $subs_2$ such that $subs_1$ is a subset of $subs_2$, modulo the states of the components. Investigating such variants is left for future work.

Chapter 7

LoCK: A Hybrid Knowledge Calculus

As explained in Chapter 4, we chose to develop a knowledge theory, because those theories allow reasoning about systems at a high-level of abstraction, and focusing on the fundamental reasons, in terms of knowledge, as why those systems are correct. Inspired by knowledge library presented in Chapter 4 (ByK), we equipped our framework with LoCK, a sound (hybrid) knowledge sequent calculus, which differs and goes beyond ByK knowledge library in several ways.

First of all, as opposed to ByK, where the knowledge operators are simply definitions within its logic of events, LoCK provides a more *principled* theory of knowledge because designing it forced us to identify the primitive constructs, as constructors of the language (see Section 7.2 for syntax and Section 7.3 for semantics) and principles, as derivation rules (see Section 7.4), of the theory. Moreover, LoCK enforces an abstraction barrier, thanks to the fact that it is deeply embedded in Coq, which does not exist in our simple knowledge library presented in Chapter 4. In addition, LoCK allows reasoning at a high-level of abstraction about trusted and non-trusted knowledge (among other things), while ByK does not distinguish between trusted and non-trusted knowledge.

Additionally, because hybrid systems have a particular architecture, whereby generic components rely on (the *trust* part of such systems) tamperproof components to correctly provide functionalities (the *trustworthy* part of such systems) that are inherited by the rest of the system (such as counting messages in MinBFT), LoCK, among other things, captures this inheritance mechanism at a high-level of abstraction (i.e., the knowledge exchanged between the nodes of a system) through general reasoning principles, called *lifting*, which we discuss in Section 6.4 (local lifting) and Section 7.7 (distributed lifting). For MinBFT, this means that if a node A receives two messages from B with the same counter, then those messages must be equal. This in turn is used to ensure that nodes cannot lie by sending

Types:
Data (ranged over by d)
$\mathsf{Trust} \subseteq \mathsf{Data}$ (ranged over by t)
$\mathsf{Identifier}$ (ranged over by i)

Functions:
$\mathsf{lt} \in \mathsf{Identifier} \to \mathsf{Identifier} \to \mathbb{P}$
$\mathsf{trustHasId} \in \mathsf{Trust} \to \mathsf{Identifier} \to \mathbb{P}$
$\mathsf{sys} \in \mathsf{System}$
$\mathsf{mem} \in \mathsf{CompName}$
$\mathsf{trust} \in \mathsf{CompName}$
$\mathsf{owner} \in \mathsf{Data} \to \mathsf{Node}$
$\mathsf{verify} \in \mathsf{Event} \to \mathsf{AuthData} \to \mathbb{B}$
$\mathsf{genFor} \in \mathsf{Data} \to \mathsf{Trust} \to \mathbb{P}$
$\mathsf{know} \in \mathsf{Data} \to \mathcal{S}(\mathsf{mem}) \to \mathbb{P}$
$\mathsf{trusted2id} \in \mathcal{S}(\mathsf{trust}) \to \mathsf{Identifier}$
$\mathsf{initId} \in \mathsf{Identifier}$
$\mathsf{auth2data} \in \mathsf{AuthData} \to \mathsf{list}(\mathsf{Data})$

Axioms:
(1) lt is transitive and anti-reflexive
(2) all initial identifiers of sys's trusted components are equal to initId
(3) $\forall t, d_1, d_2 . \mathsf{genFor}(d_1, t) \to \mathsf{genFor}(d_2, t) \to d_1 = d_2$
(4) $\mathsf{know}(d, m)$ is decidable
(5) $\neg\mathsf{know}(d, m)$ for all initial states m of sys's components

Figure 7.1: LoCK's parameters

different messages to different destinations without being noticed.

Note that LoCK provides an optional, but convenient, abstract layer to reason about crash/Byzantine/hybrid fault tolerant distributed systems without having to worry about low-level details. Using such an abstract layer allows reusing results proved once and for all at the abstract knowledge level, to derive properties of multiple concrete implementations: (1) by adequately instantiating the parameters of the abstract model (LoCK's parameters in our case—see Section 7.1); and (2) by proving that the assumptions made within the abstract model are satisfied by the concrete implementations (see Section 7.6 and Chapter 8 for examples of such assumptions). The high-level results we present here (such as the lifting property presented in Section 7.7) can be instantiated for many implementations of hybrid systems, i.e., they will be reusable for other implementations than MinBFT. We already used those results to prove the safety of two versions of MinBFT that rely on two different trusted components (see Chapter 8).

7.1 LoCK's Parameters

To be as general as possible, LoCK is parametrized by the types and functions described in Figure 7.1. Section 7.8 explains how we can instantiate those parameters to derive high-level properties of several versions of MinBFT. LoCK can be

$$
\begin{array}{lllllllllll}
\theta \in \mathsf{KType} & ::= & \texttt{KTi} & \mid & \texttt{KTn} & \mid & \texttt{KTd} & \mid & \texttt{KTt} & & \\
\upsilon \in \mathsf{KVal} & ::= & i & \mid & \boldsymbol{a} & \mid & d & & & & \\
\tau \in \mathsf{KExp} & ::= & \top & \mid & \bot & & & & & & \\
 & \mid & \tau_1 \wedge \tau_2 & \mid & \tau_1 \vee \tau_2 & \mid & \tau_1 \rightarrow \tau_2 & \mid & \exists\phi & \mid & \forall\phi \\
 & \mid & \subset\tau & \mid & \prec\tau & \mid & \sqsubset\tau & \mid & \odot & \mid & @(\boldsymbol{a}) \\
 & \mid & \upsilon_1 = \upsilon_2 & \mid & i_1 < i_2 & & & & & & \\
 & \mid & \boldsymbol{\mathcal{K}}^{+}(d) & \mid & \boldsymbol{\mathcal{L}}(d) & \mid & \boldsymbol{\mathcal{O}}(d, \boldsymbol{a}) & \mid & \boldsymbol{\mathcal{D}}(d) & & \\
 & \mid & \boldsymbol{\mathcal{I}}^{+}(i) & \mid & \boldsymbol{\mathcal{HI}}(t, i) & \mid & \boldsymbol{\mathcal{G}}(d, t) & & & &
\end{array}
$$

$$
\phi \in (\theta \in \mathsf{KType}) * \{\upsilon \in \mathsf{KVal} \mid \mathsf{oftype}(\upsilon, \theta)\} \rightarrow \mathsf{KExp}
$$

Figure 7.2: LoCK's syntax

instantiated for any kind of data (Data), trusted data[1] (Trust—a subset of Data), and identifier (Identifier—a totally ordered set, whose ordering relation is lt). Identifiers are used to identify trusted pieces of data through the trustHasId relation. In addition, LoCK is parameterized over the following operators: (1) sys is the distributed system we want to reason about; (2) mem is the name of sys's component holding the knowledge, while trust is the name of its trusted component (these could be straightforwardly generalized to lists of component names if necessary); (3) each piece of data is tagged by a node (extracted using owner) meant to be the one that generated the data; (4) verify(e, $auth$) is true iff the authenticated piece of data $auth$ can indeed be authenticated at e; (5) genFor captures the fact that trusted pieces of data are meant to correspond to non-trusted pieces of data, e.g. in MinBFT, a UI essentially corresponds to a non-trusted request (see Section 2.1.3.3); (6) know expresses what it means to hold some information; (7) the trust component is in charge of recording the last trusted identifier it generated, which is computed using trusted2id, with initial value initId; (8) auth2data extracts the list of pieces of data contained within an authenticated piece of data. We assume that if some trusted knowledge t is generated for two different pieces of data d_1 and d_2, then they must be equal. In addition, we assume that know is decidable, and that sys's nodes have no initial memory.

7.2 LoCK's Syntax

As shown in Figure 7.2, besides standard first-order logic operators ($\top$, $\bot$, $\wedge$, $\vee$, $\rightarrow$, $\exists$, $\forall$), LoCK also provides HyLoE-specific operators to state properties

---

[1] A piece of data is trusted if generated by a trusted component (e.g. UIs generated by USIGs in MinBFT—see Section 2.1.3.3).

relating different points in space/time: $\subset$, $\prec$, $\sqsubset$; to talk about initial events: $\odot$; and to relate space/time coordinates: $@$. A quantifier of the form $\exists\phi$ or of the form $\forall\phi$ takes a dependent pair ϕ as argument: (1) a type θ and (2) a function from values of type θ to expressions. The predicate $\mathsf{oftype}(\upsilon, \theta)$ is true iff $(\upsilon, \theta) \in \{(i, \mathtt{KTi}), (d, \mathtt{KTd}), (t, \mathtt{KTt}), (\boldsymbol{a}, \mathtt{KTn})\}$.

LoCK also provides general operators to capture properties about distributed knowledge. LoCK supports the standard knowledge *knows* ($\boldsymbol{\mathcal{K}}^+$) operator, which is at the core of several knowledge calculi such as the ones mentioned above. LoCK also adopts *learns* ($\boldsymbol{\mathcal{L}}$), *owns* ($\boldsymbol{\mathcal{O}}$) and *disseminate* ($\boldsymbol{\mathcal{D}}$) operator supported by ByK. In addition, LoCK also includes the *knows identifier* ($\boldsymbol{\mathcal{I}}^+$), *has identifier* ($\boldsymbol{\mathcal{HI}}$), and *generated for* ($\boldsymbol{\mathcal{G}}$) operators to state properties about trusted knowledge, which were not part of any of the systems mentioned above.

In order to enable reasoning about any point in space/time some of our operators come in two flavors, one annotated with a $^-$ and the other with $^+$. The ones annotated with $^-$ are used to state properties about the knowledge of a system right before handling an event, and are defined below; while the ones annotated with $^+$ are used to state properties once events have been handled, and are primitives of the language.

Notation. Let us now define some notation. Let $\exists_{\mathtt{i}} f$ stand for $\exists\langle \mathtt{KTi}, f\rangle$, and $\exists_{\mathtt{i}}\lambda i_1, \ldots, i_n.\tau$ for $\exists_{\mathtt{i}}\lambda i_1. \ldots \exists_{\mathtt{i}}\lambda i_n.\tau$; and similarly for the other quantifiers. In addition, let

$$\begin{array}{lcl}
\neg\tau & = & \tau \rightarrow \bot \\
\preceq\tau & = & \prec\tau \vee \tau \\
\sqsubseteq\tau & = & \sqsubset\tau \vee \tau \\
\subseteq\tau & = & \subset\tau \vee (\tau \wedge \odot) \\
i_1 \leq i_2 & = & i_1 < i_2 \vee i_1 = i_2
\end{array}
\qquad
\begin{array}{lcl}
\boldsymbol{\mathcal{K}}^-(\tau) & = & \subset\boldsymbol{\mathcal{K}}^+(\tau) \\
\boldsymbol{\mathcal{I}}^-(i) & = & \subset\boldsymbol{\mathcal{I}}^+(i) \vee (i = \mathsf{initId} \wedge \odot) \\
\boldsymbol{\mathcal{O}}(d) & = & \exists_{\mathtt{n}}\lambda\boldsymbol{a}.@(\boldsymbol{a}) \wedge \boldsymbol{\mathcal{O}}(d, \boldsymbol{a}) \\
\boldsymbol{\mathcal{OD}}(d) & = & \boldsymbol{\mathcal{O}}(d) \wedge \boldsymbol{\mathcal{D}}(d)
\end{array}$$

These abstractions are interpreted as follows: $\boldsymbol{\mathcal{O}}(d)$ means that "we" own the data d, i.e., the node at which this expression is interpreted owns the data; and $\boldsymbol{\mathcal{OD}}(d)$ means that "we" disseminated the data d, i.e., the node at which this expression is interpreted disseminated the data.

7.3 LoCK's Semantics

Figure 7.3, 7.4, and 7.5 describe LoCK's semantics: $[\![\tau]\!]_e$ is a proposition expressing that τ is true at event e. First-order logic and HyLoE operators are interpreted as expected. Let us now describe the semantics of the other knowledge operators. First, $\boldsymbol{\mathcal{L}}$'s semantics is defined in terms of the learns predicate:

$$\mathsf{learns}(e, d) = \exists auth.auth \in \mathrm{nfo2auth}(\mathrm{trigger}(e)) \wedge d \in \mathsf{auth2data}(auth) \wedge \mathsf{verify}(e, auth)$$

$$
\begin{array}{lcl}
\llbracket \top \rrbracket_e & = & \mathsf{True} \\
\llbracket \bot \rrbracket_e & = & \mathsf{False} \\
\llbracket \tau_1 \wedge \tau_2 \rrbracket_e & = & \llbracket \tau_1 \rrbracket_e \wedge \llbracket \tau_2 \rrbracket_e \\
\llbracket \tau_1 \vee \tau_2 \rrbracket_e & = & \llbracket \tau_1 \rrbracket_e \vee \llbracket \tau_2 \rrbracket_e \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_e & = & \llbracket \tau_1 \rrbracket_e \rightarrow \llbracket \tau_2 \rrbracket_e
\end{array}
\qquad
\begin{array}{lcl}
\llbracket \exists \phi \rrbracket_e & = & \exists \upsilon \in \{\upsilon \in \mathsf{KVal} \mid \mathsf{oftype}(\upsilon, \phi.1)\}.\phi.2(\upsilon) \\
\llbracket \forall \phi \rrbracket_e & = & \forall \upsilon \in \{\upsilon \in \mathsf{KVal} \mid \mathsf{oftype}(\upsilon, \phi.1)\}.\phi.2(\upsilon)
\end{array}
$$

Figure 7.3: LoCK’s semantics (predicate logic)

$$
\begin{array}{lcl}
\llbracket \prec\tau \rrbracket_e & = & \exists e' \prec e.\llbracket \tau \rrbracket_{e'} \\
\llbracket \sqsubset\tau \rrbracket_e & = & \exists e' \sqsubset e.\llbracket \tau \rrbracket_{e'} \\
\llbracket \odot \rrbracket_e & = & \mathrm{first?}(e) = \mathtt{true} \\
\llbracket \boldsymbol{@}(\boldsymbol{a}) \rrbracket_e & = & \mathrm{loc}(e) = \boldsymbol{a}
\end{array}
\qquad
\llbracket \subset\tau \rrbracket_e \;=\; \begin{cases} \llbracket \tau \rrbracket_{e'}, \text{ if } \mathrm{pred}(e) = \mathtt{Some}(e') \\ \mathsf{False} \text{ otherwise} \end{cases}
$$

Figure 7.4: LoCK’s semantics (logic of events)

$$
\begin{array}{lcl}
\llbracket \boldsymbol{\mathcal{L}}(d) \rrbracket_e & = & \mathsf{learns}(e, d) \\
\llbracket \boldsymbol{\mathcal{K}}^{+}(d) \rrbracket_e & = & \mathsf{knows}^{+}(e, d) \\
\llbracket \boldsymbol{\mathcal{I}}^{+}(i) \rrbracket_e & = & \mathsf{ident}^{+}(e, i) \\
\llbracket \boldsymbol{\mathcal{D}}(d) \rrbracket_e & = & d \in \mathsf{sys} \rightsquigarrow e \\
\llbracket \boldsymbol{\mathcal{HI}}(t, i) \rrbracket_e & = & \mathsf{trustHasId}(t, i) \\
\llbracket \boldsymbol{\mathcal{O}}(d, \boldsymbol{a}) \rrbracket_e & = & \mathsf{owner}(d) = \boldsymbol{a} \\
\llbracket \boldsymbol{\mathcal{G}}(d, t) \rrbracket_e & = & \mathsf{genFor}(d, t)
\end{array}
\qquad
\begin{array}{lcl}
\llbracket \upsilon_1 = \upsilon_2 \rrbracket_e & = & \upsilon_1 = \upsilon_2 \\
\llbracket i_1 < i_2 \rrbracket_e & = & \mathsf{lt}(i_1, i_2)
\end{array}
$$

Figure 7.5: LoCK’s semantics (knowledge)

This states that a node learns d at some event e, if e was triggered by an input that contains the data d. Moreover, in order to deal with Byzantine faults, we also require that to learn some data one has to be able to verify its authenticity. Then, $\boldsymbol{\mathcal{K}}^{+}$ is interpreted by the knows^{+} predicate:

$$\mathsf{knows}^{+}(e, d) = \exists m \in \mathcal{S}(\mathsf{mem}).\mathsf{sys}@^{+} e\downharpoonright_{\mathsf{mem}} = m \wedge \mathsf{know}(d, m)$$

where $\mathsf{knows}^{+}(e, d)$ states that a node knows d at some event e, if it holds d in its memory m (i.e. $\mathsf{know}(d, m)$ is true), such that its memory m is the state of the component mem right after e. Finally, $\boldsymbol{\mathcal{I}}^{+}$ is interpreted by the ident^{+} predicate:

$$\mathsf{ident}^{+}(e, i) = \exists m \in \mathcal{S}(\mathsf{trust}).\mathsf{sys}@^{+} e\downharpoonright_{\mathsf{trust}} = m \wedge \mathsf{trusted2id}(m) = i$$

This states that the trusted component trust remembers the current trusted identifier i after e.

7.4 LoCK’s Rules

Syntax. Figure 7.6 presents the syntax of rules. Expressions are annotated with

$$\alpha \in \mathsf{EventRel} \quad ::= \quad e_1 \equiv e_2 \mid e_1 \subset e_2 \mid e_1 \prec e_2 \mid e_1 \preceq e_2 \mid e_1 \sqsubset e_2 \mid e_1 \sqsubseteq e_2$$

$$\begin{array}{llllll}
x \in \mathsf{HypName} & \text{(a set of hypothesis names)} & & y \in \mathsf{GuardName} & \text{(a set of guard names)} & \\
\sigma \in \mathsf{KExpAt} & ::= & \tau \mathbin{@} e & g \in \mathsf{Guard} & ::= & y : \alpha \\
h \in \mathsf{Hyp} & ::= & x : \sigma & G \in \mathsf{Guards} & ::= & \oslash \mid G, g \\
H \in \mathsf{Hyps} & ::= & \oslash \mid H, h & \mathit{seq} \in \mathsf{Sequent} & ::= & \langle G \rangle\, H \vdash \sigma
\end{array}$$

$$R \in \mathsf{Rule} \quad ::= \quad \frac{\Lambda[\overline{e}, \overline{t}, \overline{i}] \quad \mathit{seq}_1 \quad \cdots \quad \mathit{seq}_n}{\mathit{seq}}$$

Figure 7.6: Syntax of knowledge calculus rules

events allowing different expressions to be true at different points in space/time in a single sequent/rule. In a sequent of the form $\langle G \rangle\, H \vdash \sigma$, the list of guards G is used to relate the different events mentioned in the hypotheses H and the conclusion σ. Note that for convenience we use the same symbols for guards and for the corresponding knowledge expressions ($e_1 \prec e_2$ is a guard, while $\prec\tau$ is an expression).[2] For convenience, hypotheses and guards are all named in a sequent, allowing rules to point to them (expressions do not depend on names). We write H_1, H_2 for the list H_1 appended with the list H_2, and similarly for guards. A rule R is essentially a pair of a list of sequents (R's hypotheses) and a sequent (R's conclusion). In addition, the hypotheses of a rule can depend on a list of events $\overline{e}$, a list of trusted values $\overline{t}$, and a list of trusted identifiers $\overline{i}$, allowing rules to introduce new symbols. We omit the $\Lambda[\_]$ part in rules that do not introduce new symbols. We sometime write $H[\sigma]$, for a list of hypotheses H that contains an hypothesis of the form $x : \sigma$, and similarly for guards. We then sometimes write $H[\sigma']$ to denote the same list of hypotheses where $x : \sigma$ is replaced by $x : \sigma'$.

Semantics. Guards, hypotheses, and sequents are interpreted as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
[\![e_1 \square e_2]\!] & = & e_1 \circ e_2 & [\![G]\!] & = & \forall g \in G. [\![g]\!] \\
[\![x : \tau \mathbin{@} e]\!] & = & [\![\tau]\!]_e & [\![H]\!] & = & \forall h \in H. [\![h]\!] \\
[\![\langle G \rangle\, H \vdash \sigma]\!] & = & [\![G]\!] \to [\![H]\!] \to [\![\sigma]\!] & & &
\end{array}$$

where $(\square, \circ) \in \{(\sqsubset, \sqsubset), (\sqsubseteq, \sqsubseteq), (\prec, \prec), (\preceq, \preceq), (\subset, \subset), (\equiv, =)\}$. Note that $\square$ is a guard operator, while $\circ$ is a HyLoE operator. Finally, a rule R (see Figure 7.6) is true if $[\![\mathit{seq}]\!]$ (R's conclusion) follows from $[\![\mathit{seq}_1]\!] \wedge \cdots \wedge [\![\mathit{seq}_n]\!]$ (R's hypotheses) for all possible instances of $\overline{e}$, $\overline{t}$, and $\overline{i}$.

Primitive Rules. We now provide a sample of LoCK's derivation rules. As men-

---

[2] Note also that the collection of guards is not minimal for convenience.

$$\dfrac{\langle G\rangle\; H_1, H_2 \vdash \sigma_2}{\langle G\rangle\; H_1, h, H_2 \vdash \sigma_2}\;\texttt{thin}_{\texttt{h}} \qquad \dfrac{\langle G_1, G_2\rangle\; H \vdash \sigma}{\langle G_1, g, G_2\rangle\; H \vdash \sigma}\;\texttt{thin}_{\texttt{g}}$$

$$\dfrac{\langle G\rangle\; H[x_2 : \sigma'] \vdash \sigma}{\langle G\rangle\; H[x_1 : \sigma'] \vdash \sigma}\;\texttt{ren}_{\texttt{h}} \qquad \dfrac{\langle G[y_2 : \alpha]\rangle\; H \vdash \sigma}{\langle G[y_1 : \alpha]\rangle\; H \vdash \sigma}\;\texttt{ren}_{\texttt{g}}$$

$$\dfrac{}{\langle G\rangle\; H[\sigma] \vdash \sigma}\;\texttt{hyp} \qquad \dfrac{\langle G\rangle\; H \vdash \sigma_2 \quad \langle G\rangle\; H, x : \sigma_2 \vdash \sigma_1}{\langle G\rangle\; H \vdash \sigma_1}\;\texttt{cut}$$

Figure 7.7: LoCK's structural rules

$$\dfrac{}{\langle G\rangle\; H \vdash \top \;@\; e}\;\top_{\texttt{I}} \qquad \dfrac{}{\langle G\rangle\; H[\bot \;@\; e] \vdash \sigma}\;\bot_{\texttt{E}}$$

$$\dfrac{\langle G\rangle\; H, x : \tau_1 \;@\; e \vdash \tau_2 \;@\; e}{\langle G\rangle\; H \vdash \tau_1 \rightarrow \tau_2 \;@\; e}\;\rightarrow_{\texttt{I}} \qquad \dfrac{\langle G\rangle\; H_1, H_2 \vdash \tau_1 \;@\; e \quad \langle G\rangle\; H_1, x : \tau_2 \;@\; e, H_2 \vdash \sigma}{\langle G\rangle\; H_1, x : \tau_1 \rightarrow \tau_2 \;@\; e, H_2 \vdash \sigma}\;\rightarrow_{\texttt{E}}$$

$$\dfrac{\langle G\rangle\; H \vdash f(\upsilon) \;@\; e \quad \mathsf{oftype}(\upsilon, \theta)}{\langle G\rangle\; H \vdash \exists\langle\theta, f\rangle \;@\; e}\;\exists_{\texttt{I}} \qquad \dfrac{\Lambda[\upsilon] \quad \langle G\rangle\; H_1, x : f(\upsilon) \;@\; e, H_2 \vdash \sigma \quad \mathsf{oftype}(\upsilon, \theta)}{\langle G\rangle\; H_1, x : \exists\langle\theta, f\rangle \;@\; e, H_2 \vdash \sigma}\;\exists_{\texttt{E}}$$

$$\dfrac{\langle G\rangle\; H \vdash \tau_1 \;@\; e \quad \langle G\rangle\; H \vdash \tau_2 \;@\; e}{\langle G\rangle\; H \vdash \tau_1 \wedge \tau_2 \;@\; e}\;\wedge_{\texttt{I}} \qquad \dfrac{\langle G\rangle\; H_1, x : \tau_1 \;@\; e, x' : \tau_2 \;@\; e, H_2 \vdash \sigma}{\langle G\rangle\; H_1, x : \tau_1 \wedge \tau_2 \;@\; e, H_2 \vdash \sigma}\;\wedge_{\texttt{E}}$$

$$\dfrac{\langle G\rangle\; H \vdash \tau_1 \;@\; e}{\langle G\rangle\; H \vdash \tau_1 \vee \tau_2 \;@\; e}\;\vee_{\texttt{Il}} \qquad \dfrac{\langle G\rangle\; H_1, x : f(\upsilon) \;@\; e, H_2 \vdash \sigma \quad \mathsf{oftype}(\upsilon, \theta)}{\langle G\rangle\; H_1, x : \forall\langle\theta, f\rangle \;@\; e, H_2 \vdash \sigma}\;\forall_{\texttt{E}}$$

$$\dfrac{\langle G\rangle\; H \vdash \tau_2 \;@\; e}{\langle G\rangle\; H \vdash \tau_1 \vee \tau_2 \;@\; e}\;\vee_{\texttt{Ir}} \qquad \dfrac{\Lambda[\upsilon] \quad \langle G\rangle\; H \vdash f(\upsilon) \;@\; e \quad \mathsf{oftype}(\upsilon, \theta)}{\langle G\rangle\; H \vdash \forall\langle\theta, f\rangle \;@\; e}\;\forall_{\texttt{I}}$$

$$\dfrac{\begin{array}{c}\langle G\rangle\; H_1, x : \tau_1 \;@\; e, H_2 \vdash \sigma \\ \langle G\rangle\; H_1, x : \tau_2 \;@\; e, H_2 \vdash \sigma\end{array}}{\langle G\rangle\; H_1, x : \tau_1 \vee \tau_2 \;@\; e, H_2 \vdash \sigma}\;\vee_{\texttt{E}}$$

Figure 7.8: LoCK's predicate logic rules

tioned above, LoCK is sound in the sense that we have proved that its inference rules are sound w.r.t. the HyLoE-based semantics introduced above (we skip those proofs here since they are all straightforward).

Figure 7.7 presents LoCK's structural rules, while Figure 7.8 presents LoCK's predicate logic rules, which are all standard. For example, `hyp` represents hypothesis rule, $\rightarrow_{\texttt{E}}$ represents implication elimination; $\vee_{\texttt{Il}}/\vee_{\texttt{Ir}}$ or introduction left/right; and $\vee_{\texttt{E}}$ represents or elimination.

Moreover, Figure 7.9 presents LoCK's event relation rules. The collection of

Let $\Box \in \{\sqsubset, \subset, \prec\}$ and $(\lhd, \blacktriangleleft) \in \{(\prec, \preceq), (\sqsubset, \sqsubseteq), (\sqsubset, \prec), (\sqsubseteq, \preceq), (\subset, \sqsubset), (\equiv, \sqsubseteq)\}$

$$\frac{\Lambda[e'] \quad \langle G, y : e' \Box e\rangle\; H[x : \tau \mathrel{@} e'] \vdash \sigma}{\langle G\rangle\; H[x : \Box\tau \mathrel{@} e] \vdash \sigma}\;\Box_{\mathtt{E}} \qquad \frac{\langle G[e' \Box e]\rangle\; H \vdash \tau \mathrel{@} e'}{\langle G[e' \Box e]\rangle\; H \vdash \Box\tau \mathrel{@} e}\;\Box_{\mathtt{I}}$$

$$\frac{\langle G[e' \Box e]\rangle\; H \vdash \Box\tau \mathrel{@} e'}{\langle G[e' \Box e]\rangle\; H \vdash \Box\tau \mathrel{@} e}\;\Box_{\mathtt{It}} \qquad \frac{\begin{array}{l}\langle G, y : \mathrm{pred}^{=}(e) \equiv e\rangle\; H \vdash \sigma \\ \langle G\rangle\; H \vdash \odot \mathrel{@} e\end{array}}{\langle G\rangle\; H \vdash \sigma}\;\mathtt{if}\odot$$

$$\frac{\begin{array}{l}\langle G, y : \mathrm{pred}^{=}(e) \subset e\rangle\; H \vdash \sigma \\ \langle G\rangle\; H \vdash \neg\odot \mathrel{@} e\end{array}}{\langle G\rangle\; H \vdash \sigma}\;\mathtt{if}\neg\odot \qquad \frac{\langle G[e' \blacktriangleleft e]\rangle\; H \vdash \sigma}{\langle G[e' \lhd e]\rangle\; H \vdash \sigma}\;\mathtt{weak}$$

$$\frac{\langle G[e_1 \equiv e_2]\rangle\; H[\tau \mathrel{@} e_2] \vdash \sigma}{\langle G[e_1 \equiv e_2]\rangle\; H[\tau \mathrel{@} e_1] \vdash \sigma}\;\mathtt{sub}_{\mathtt{H}} \qquad \frac{\langle G[y : e_1 \equiv e_2]\rangle\; H \vdash \tau \mathrel{@} e_1}{\langle G[y : e_1 \equiv e_2]\rangle\; H \vdash \tau \mathrel{@} e_2}\;\mathtt{sub}_{\mathtt{C}}$$

$$\frac{\langle G, y : e \equiv e\rangle\; H \vdash \sigma}{\langle G\rangle\; H \vdash \sigma}\;\equiv_{\mathtt{refl}} \qquad \frac{\langle G\rangle\; H[x : \tau \mathrel{@} \mathrm{pred}^{=}(e)] \vdash \sigma}{\langle G\rangle\; H[x : \subset\tau \mathrel{@} e] \vdash \sigma}\;\subset_{\mathtt{E}}$$

$$\frac{\langle G[e_1 \equiv e_2]\rangle\; H \vdash \sigma \quad \langle G[e_1 \prec e_2]\rangle\; H \vdash \sigma}{\langle G[e_1 \preceq e_2]\rangle\; H \vdash \sigma}\;\mathtt{STR}_{\preceq} \qquad \frac{\langle G[e_1 \equiv e_2]\rangle\; H \vdash \sigma \quad \langle G[e_1 \sqsubset e_2]\rangle\; H \vdash \sigma}{\langle G[e_1 \sqsubseteq e_2]\rangle\; H \vdash \sigma}\;\mathtt{STR}_{\sqsubseteq}$$

$$\frac{\begin{array}{l}\langle G[y : e_1 \sqsubseteq e_2]\rangle\; H \vdash \sigma \\ \langle G[y : e_1 \preceq e_2]\rangle\; H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathrel{@} e_1 \\ \langle G[y : e_1 \preceq e_2]\rangle\; H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathrel{@} e_2\end{array}}{\langle G[y : e_1 \preceq e_2]\rangle\; H \vdash \sigma}\;\mathtt{STRl}_{\preceq} \qquad \frac{\begin{array}{l}\langle G[y : e_1 \sqsubset e_2]\rangle\; H \vdash \sigma \\ \langle G[y : e_1 \prec e_2]\rangle\; H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathrel{@} e_1 \\ \langle G[y : e_1 \prec e_2]\rangle\; H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathrel{@} e_2\end{array}}{\langle G[y : e_1 \prec e_2]\rangle\; H \vdash \sigma}\;\mathtt{STRl}_{\prec}$$

$$\frac{\langle G[y : e_2 \equiv e_1]\rangle\; H \vdash \sigma}{\langle G[y : e_1 \equiv e_2]\rangle\; H \vdash \sigma}\;\equiv_{\mathtt{sym}} \qquad \frac{\langle G[y : e_1 \equiv \mathrm{pred}^{=}(e_2)]\rangle\; H \vdash \sigma}{\langle G[y : e_2 \subset e_1]\rangle\; H \vdash \sigma}\;\equiv_{\mathrm{pred}^{=}}$$

$$\frac{\begin{array}{l}\langle G[y : e_1 \subset e_2]\rangle\; H \vdash \sigma \\ \langle G[y : e_1 \sqsubset e, e \subset e_2]\rangle\; H \vdash \sigma\end{array}}{\langle G[y : e_1 \sqsubset e_2]\rangle\; H \vdash \sigma}\;\mathtt{split}_{\sqsubset}$$

$$\frac{\langle G[y : e_1 \sqsubseteq \mathrm{pred}^{=}(e_2), \mathrm{pred}^{=}(e_2) \subset e_2]\rangle\; H \vdash \sigma}{\langle G[y : e_1 \sqsubset e_2]\rangle\; H \vdash \sigma}\;\mathtt{splitPred}_{\sqsubset}$$

Figure 7.9: LoCK's event relation rules

rules is by no means complete. The family of elimination rules $\Box_{\mathtt{E}}$ allows turning HyLoE operators into guards, while the families of introduction rules $\Box_{\mathtt{I}}$ and $\Box_{\mathtt{It}}$ allow using those guards to navigate between points in space/time to prove HyLoE expressions. The two rules `if`$\odot$ and `if`$\neg\odot$ provide an axiomatization of $\mathrm{pred}^{=}$. The `weak` family of rules allows weakening guards, e.g., from $\prec$ to $\preceq$. Finally, the

$$\frac{\begin{array}{l}\Lambda[e'] \quad \langle G, y : e' \sqsubseteq e \rangle\, H \vdash \odot \rightarrow \tau \mathbin{@} e' \\ \qquad\quad\; \langle G, y : e' \sqsubseteq e \rangle\, H \vdash \subset\tau \rightarrow \tau \mathbin{@} e' \end{array}}{\langle G \rangle\, H \vdash \tau \mathbin{@} e}\ \texttt{ind} \qquad\qquad \frac{}{\langle G[e_1 \sqsubset e_2] \rangle\, H \vdash \neg\odot \mathbin{@} e_2}\ \neg\odot$$

$$\frac{\begin{array}{l}\langle G \rangle\, H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathbin{@} e_1 \\ \langle G \rangle\, H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathbin{@} e_2 \end{array} \quad \begin{array}{l} \langle G, y : e_1 \equiv e_2 \rangle\, H \vdash \sigma \\ \langle G, y : e_1 \sqsubset e_2 \rangle\, H \vdash \sigma \\ \langle G, y : e_2 \sqsubset e_1 \rangle\, H \vdash \sigma \end{array}}{\langle G \rangle\, H \vdash \sigma}\ \texttt{tri} \qquad\qquad \frac{}{\langle G \rangle\, H \vdash \odot \vee \neg\odot \mathbin{@} e}\ \odot_{\texttt{dec}}$$

$$\frac{\begin{array}{l}\langle G \rangle\, H \vdash \boldsymbol{@}(\boldsymbol{a}_1) \mathbin{@} e \\ \langle G \rangle\, H \vdash \boldsymbol{@}(\boldsymbol{a}_2) \mathbin{@} e \end{array}}{\langle G \rangle\, H \vdash \boldsymbol{a}_1 = \boldsymbol{a}_2 \mathbin{@} e}\ \texttt{loc} \qquad\qquad \frac{\langle G[y : e_1 \sqsubseteq e_2] \rangle\, H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathbin{@} e_2}{\langle G[y : e_1 \sqsubseteq e_2] \rangle\, H \vdash \boldsymbol{@}(\boldsymbol{a}) \mathbin{@} e_1}\ \boldsymbol{@}_{\texttt{loc}}$$

Figure 7.10: LoCK's logic of events rules

Let $(\pi, \kappa, \rho) \in \{(=,<,<),(<,=,<),(<,<,<),(=,=,=)\}$.

Let $\tau \in \{(\upsilon_1 = \upsilon_2, i_1 < i_2, \boldsymbol{\mathcal{HI}}(t,i), \boldsymbol{\mathcal{O}}(d,\boldsymbol{a}), \boldsymbol{\mathcal{G}}(d,t)\}$.

$$\frac{}{\langle G \rangle\, H \vdash \boldsymbol{\mathcal{K}}^{+}(d) \vee \neg\boldsymbol{\mathcal{K}}^{+}(d) \mathbin{@} e}\ \texttt{K}_{\texttt{dec}} \qquad\qquad \frac{\begin{array}{l}\langle G \rangle\, H \vdash \boldsymbol{\mathcal{O}}(d, \boldsymbol{a}_1) \mathbin{@} e \\ \langle G \rangle\, H \vdash \boldsymbol{\mathcal{O}}(d, \boldsymbol{a}_2) \mathbin{@} e \end{array}}{\langle G \rangle\, H \vdash \boldsymbol{a}_1 = \boldsymbol{a}_2 \mathbin{@} e}\ \texttt{lowner}$$

$$\frac{\begin{array}{l}\langle G \rangle\, H \vdash \boldsymbol{\mathcal{G}}(d_1, t) \mathbin{@} e \\ \langle G \rangle\, H \vdash \boldsymbol{\mathcal{G}}(d_2, t) \mathbin{@} e \end{array}}{\langle G \rangle\, H \vdash d_1 = d_2 \mathbin{@} e}\ \texttt{ldata} \qquad\qquad \frac{\begin{array}{l}\langle G \rangle\, H \vdash \boldsymbol{\mathcal{I}}^{+}(i_1) \mathbin{@} e \\ \langle G \rangle\, H \vdash \boldsymbol{\mathcal{I}}^{+}(i_2) \mathbin{@} e \end{array}}{\langle G \rangle\, H \vdash i_1 = i_2 \mathbin{@} e}\ \texttt{lid}$$

$$\frac{\langle G \rangle\, H \vdash \tau \mathbin{@} e_2}{\langle G \rangle\, H \vdash \tau \mathbin{@} e_1}\ \texttt{change} \qquad\qquad \frac{\begin{array}{l}\langle G \rangle\, H \vdash \upsilon_1 = \upsilon_2 \mathbin{@} e \\ \langle G \rangle\, H \vdash \tau[\upsilon_2] \mathbin{@} e \end{array}}{\langle G \rangle\, H \vdash \tau[\upsilon_1] \mathbin{@} e}\ \texttt{valSub}$$

$$\frac{}{\langle G \rangle\, H[i < i] \vdash \sigma}\ \texttt{irrefl} \qquad\qquad \frac{\langle G \rangle\, H \vdash i_1\ \pi\ i \mathbin{@} e \qquad \langle G \rangle\, H \vdash i\ \kappa\ i_2 \mathbin{@} e}{\langle G \rangle\, H \vdash i_1\ \rho\ i_2 \mathbin{@} e}\ \texttt{trans}$$

Figure 7.11: LoCK's knowledge rules

substitution rules $\texttt{sub}_{\texttt{H}}$ and $\texttt{sub}_{\texttt{C}}$ allow substituting events in a sequent's hypotheses and conclusion. The $\subset_{\texttt{E}}$ rule is the standard elimination rule for $\subset$, allowing to navigate to previous events. The $\texttt{STR}_{\preceq}$ and $\texttt{STR}_{\sqsubseteq}$ allow strengthening $\preceq$ and $\sqsubseteq$. The $\texttt{STRl}_{\preceq}$ and $\texttt{STRl}_{\prec}$ allow strengthening $\preceq$ and $\prec$. The $\texttt{split}_{\sqsubset}$ and $\texttt{splitPred}_{\sqsubset}$ allow splitting guards to get intermediate events.

Figure 7.10 presents LoCK's HyLoE rules. The `ind` rule is an induction rule on causal time. It says that to prove that a property is true at some event e, it is enough to prove that it is true at the first event prior to e (the base case), and that for any event e' prior to e, if it is true right before e', then it is also true at e' (the inductive case). The $\neg\odot$ rule states that if some event e_1 happened strictly and locally before some event e_2, then e_2 cannot be the first event at that location. The `tri` rule axiomatizes the HyLoE fact that if two events e_1 and e_2 happen at the same location $\boldsymbol{a}$, then either the events are equal, or one happened before the other. The rule $\odot_{\texttt{dec}}$ states that $\odot$ is decidable. The `loc` rule states that each event happens at a single location. Finally, $@_{\texttt{loc}}$ rule (note that this rule is invertible) states that if $e_1 \sqsubseteq e_2$ then e_1 and e_2 happen at the same location.

Figure 7.11 presents LoCK's knowledge rules. The $\texttt{K}_{\texttt{dec}}$ rule says that $\mathcal{K}^+$ is decidable. The `1owner` rule states that a given piece of data can only be owned by a single node. The `1data` rule states that trusted pieces of data can only be related to a single piece of data. The `1id` rule states that one can only know about a single identifier at any point in time. The `change` rule allows changing the current event for event agnostic expressions. Finally, the `valSub` rule allows substituting equal values in any expression.

Using this calculus we derived Theorem 2 in Section 7.7, among others. In addition, Section 7.6 provides further examples of expressions that can be derived using LoCK.

7.5 Examples of Derivations Within LoCK

Let us now provide a few simple examples to illustrate the expressiveness of our calculus, as well as the usefulness of some of its features, such as guards.[3]

7.5.1 Non-initial-events

We start by proving that if τ happened before, then the current event cannot be the initial event, i.e.: $\sqsubset\tau \rightarrow \neg\odot$ (see figure on below).[4] In this first example, we only navigate between events in the hypothesis x: we use the $\Box_{\texttt{E}}$ elimination rule to introduce a guard, that allows navigating from the point in space/time where $\sqsubset\tau$ is true (i.e., e), to the point where τ is true (i.e., e'). We conclude using $\neg\odot$, which says that a point that has predecessors cannot be the first event.

---

[3] We omit here the $\Lambda[\_]$ part for readability.

[4] See `DERIVED_RULE_local_before_implies_not_first_true` in `code/model/CalculusSM_derived3.v`.

$$\dfrac{\dfrac{\dfrac{}{\langle y : e' \sqsubset e \rangle\; x : \tau \mathbin{@} e' \vdash \neg\odot \mathbin{@} e}\;\neg\odot}{\langle \oslash \rangle\; x : \sqsubset\tau \mathbin{@} e \vdash \neg\odot \mathbin{@} e}\;\Box_{\mathtt{E}}}{\langle \oslash \rangle\; \oslash \vdash \sqsubset\tau \to \neg\odot \mathbin{@} e}\;\to_{\mathtt{E}}$$

7.5.2 Collapsing

We now prove another simple, though slightly more involved, example (see figure below), where we use guards to navigate through events in multiple formulas: both in hypothesis x and in the conclusion. Namely, we prove: $\sqsubset\sqsubset\tau \to \sqsubset\tau$, which says that if it happened before that τ happened before, then τ happened before.[5] We use the $\Box_{\mathtt{E}}$ elimination rules twice to go from the point where $\sqsubset\sqsubset\tau$ is true (i.e., e), to the point where τ is true (i.e., e''). We then use the $\Box_{\mathtt{It}}$ introduction rule to navigate to the e' intermediary point. Finally, we use the $\Box_{\mathtt{I}}$ introduction rule to navigate to e'', while eliminating $\sqsubset$ (as opposed to the previous step, which keeps the operator).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle y : e' \sqsubset e, y' : e'' \sqsubset e' \rangle\; x : \tau \mathbin{@} e'' \vdash \tau \mathbin{@} e''}\;\mathtt{hyp}}{\langle y : e' \sqsubset e, y' : e'' \sqsubset e' \rangle\; x : \tau \mathbin{@} e'' \vdash \sqsubset\tau \mathbin{@} e'}\;\Box_{\mathtt{I}}}{\langle y : e' \sqsubset e, y' : e'' \sqsubset e' \rangle\; x : \tau \mathbin{@} e'' \vdash \sqsubset\tau \mathbin{@} e}\;\Box_{\mathtt{It}}}{\langle y : e' \sqsubset e \rangle\; x : \sqsubset\tau \mathbin{@} e' \vdash \sqsubset\tau \mathbin{@} e}\;\Box_{\mathtt{E}}}{\langle \oslash \rangle\; x : \sqsubset\sqsubset\tau \mathbin{@} e \vdash \sqsubset\tau \mathbin{@} e}\;\Box_{\mathtt{E}}}{\langle \oslash \rangle\; \oslash \vdash \sqsubset\sqsubset\tau \to \sqsubset\tau \mathbin{@} e}\;\to_{\mathtt{E}}$$

7.5.3 Weakening

Our next example illustrates how our `weak` rules become handy when navigating between points in space/time. We show here that we can derive $\langle G \rangle\, H[x : \sqsubseteq\tau \mathbin{@} e] \vdash \sigma$ from $\langle G, y : e' \sqsubseteq e \rangle\, H[x : \tau \mathbin{@} e'] \vdash \sigma$, i.e., we derive $\sqsubseteq$'s elimination rule. We weaken here both $\sqsubset$ and $\equiv$, to $\sqsubseteq$, in order to obtain the same guard in both branches of our derivation:[6]

$$\dfrac{\dfrac{\dfrac{\Lambda[e']\; \langle G, y : e' \sqsubseteq e \rangle\; H[x : \tau \mathbin{@} e'] \vdash \sigma}{\Lambda[e']\; \langle G, y : e' \sqsubset e \rangle\; H[x : \tau \mathbin{@} e'] \vdash \sigma}\;\mathtt{weak}}{\langle G \rangle\; H[x : \sqsubset\tau \mathbin{@} e] \vdash \sigma}\;\Box_{\mathtt{E}} \qquad \dfrac{\dfrac{\dfrac{\Lambda[e']\; \langle G, y : e' \sqsubseteq e \rangle\; H[x : \tau \mathbin{@} e'] \vdash \sigma}{\langle G, y : e \sqsubseteq e \rangle\; H[x : \tau \mathbin{@} e'] \vdash \sigma}}{\langle G, y : e \equiv e \rangle\; H[x : \tau \mathbin{@} e'] \vdash \sigma}\;\mathtt{weak}}{\langle G \rangle\; H[x : \tau \mathbin{@} e] \vdash \sigma}\;\equiv_{\mathtt{refl}}}{\langle G \rangle\; H[x : \sqsubseteq\tau \mathbin{@} e] \vdash \sigma}\;\vee_{\mathtt{E}}$$

---

[5] See `DERIVED_RULE_twice_local_before_implies_once_true` in `code/model/CalculusSM_derived3.v`.

[6] See `DERIVED_RULE_unlocal_before_eq_hyp_true` in `code/model/CalculusSM.v`.

7.5.4 Predecessor

Next, we prove that if τ was true at $\mathsf{pred}^{=}(e)$ (denoted e_p below) then it must be that τ happened before or at e.[7] Once again, we use here LoCK's feature that different expressions in a sequent can be true at different events: x is true at e_p, while the conclusion of the root is true at e. In the following proof, Π_1 is a proof that $\odot$ is decidable (using $\odot_{\texttt{dec}}$); Π_2 is a proof of $\odot$ (using `hyp`); and Π_3 is a proof of $\neg\odot$ (using `hyp`)—those are eluded here for readability:

$$\dfrac{\Pi_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle y : e_p \equiv e\rangle\, x : \tau @ e, o : \odot @ e \vdash \tau @ e}\,\texttt{hyp}}{\langle y : e_p \equiv e\rangle\, x : \tau @ e_p, o : \odot @ e \vdash \tau @ e}\,\texttt{sub}_{\texttt{H}} \qquad \Pi_2}{\langle\oslash\rangle\, x : \tau @ e_p, o : \odot @ e \vdash \tau @ e}\,\texttt{if}\odot}{\langle\oslash\rangle\, x : \tau @ e_p, o : \odot @ e \vdash \sqsubseteq\tau @ e}\,\vee_{\texttt{Ir}} \qquad \Pi}{\langle\oslash\rangle\, x : \tau @ e_p, o : \odot \vee \neg\odot @ e \vdash \sqsubseteq\tau @ e}\,\vee_{\texttt{E}}}{\langle\oslash\rangle\, x : \tau @ e_p \vdash \sqsubseteq\tau @ e}\,\texttt{cut}$$

where Π is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle y : e_p \sqsubset e\rangle\, x : \tau @ e_p, o : \neg\odot @ e \vdash \tau @ e_p}\,\texttt{hyp} \qquad \Pi_3}{\langle y : e_p \sqsubset e\rangle\, x : \tau @ e_p, o : \neg\odot @ e \vdash \sqsubset\tau @ e}\,\Box_{\texttt{I}}}{\langle y : e_p \subset e\rangle\, x : \tau @ e_p, o : \neg\odot @ e \vdash \sqsubset\tau @ e}\,\texttt{weak}}{\langle\oslash\rangle\, x : \tau @ e_p, o : \neg\odot @ e \vdash \sqsubset\tau @ e}\,\texttt{if}\neg\odot}{\langle\oslash\rangle\, x : \tau @ e_p, o : \neg\odot @ e \vdash \sqsubseteq\tau @ e}\,\vee_{\texttt{Il}}$$

7.5.5 Acquired knowledge

Finally, let us present another useful fact that allows getting back to the point where the knowledge was acquired (because it was locally generated or because it was received): if we know some piece of data d, then there was a point e' in the past, where we did not know d before e' but we knew it after e'.[8] We state this fact as a derived rule as follows:

$$\dfrac{\langle G\rangle\, H \vdash \mathcal{K}^{+}(d) @ e}{\langle G\rangle\, H \vdash \sqsubseteq(\mathcal{K}^{+}(d) \wedge \neg\mathcal{K}^{-}(d)) @ e} \tag{7.1}$$

which we prove by induction on causal time using `ind`. To prove the base case, we first eliminate $\sqsubseteq$ using $\vee_{\texttt{Ir}}$. The left conjunct follows trivially from our hypothesis, and we prove the right conjunct using `weak` and $\neg\odot$. The inductive case follows from $\texttt{K}_{\texttt{dec}}$, i.e. that knowledge is decidable.

---

[7] See `DERIVED_RULE_at_pred_implies_local_before_eq_true` in `code/model/CalculusSM_derived3.v`.

[8] See the lemma called `DERIVED_RULE_knowledge_acquired_true` in the filed called `code/model/CalculusSM.v`.

7.5.6 Owns Propagated

In the following example, we show how owning some knowledge propagates trough time, i.e., if a node owns a piece of data at event e_1, then it will know that piece of data at all events that happened after event e_1. To do so, we show here that we can derive $\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \boldsymbol{\mathcal{O}}(d)\ @\ e_2$ from $\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \boldsymbol{\mathcal{O}}(d)\ @\ e_1$, [9]

$$\dfrac{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \boldsymbol{\mathcal{O}}(d)\ @\ e_1 \quad \Pi_1}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \boldsymbol{\mathcal{O}}(d)\ @\ e_2}\ \texttt{cut}$$

where Π_1 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1, y : \boldsymbol{@}(\boldsymbol{a})\ @\ e_1 \vdash \boldsymbol{@}(\boldsymbol{a})\ @\ e_1}\ \texttt{hyp}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1, y : \boldsymbol{@}(\boldsymbol{a})\ @\ e_1 \vdash \boldsymbol{@}(\boldsymbol{a})\ @\ e_2}\ \boldsymbol{@}_{\texttt{loc}} \quad \Pi_2}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1, y : \boldsymbol{@}(\boldsymbol{a})\ @\ e_1 \vdash \boldsymbol{@}(\boldsymbol{a}) \land \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_2}\ \land_{\texttt{I}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{@}(\boldsymbol{a}) \land \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1 \vdash \boldsymbol{@}(\boldsymbol{a}) \land \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_2}\ \land_{\texttt{E}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{@}(\boldsymbol{a}) \land \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1 \vdash \exists_{\texttt{n}}(\boldsymbol{@}(\boldsymbol{a}) \land \boldsymbol{\mathcal{O}}(d, \boldsymbol{a}))\ @\ e_2}\ \exists_{\texttt{I}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{\mathcal{O}}(d)\ @\ e_1 \vdash \boldsymbol{\mathcal{O}}(d)\ @\ e_2}\ \exists_{\texttt{E}}}{}$$

and Π_2 is:

$$\dfrac{\dfrac{}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1, y : \boldsymbol{@}(\boldsymbol{a})\ @\ e_1 \vdash \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1}\ \texttt{hyp}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_1, y : \boldsymbol{@}(\boldsymbol{a})\ @\ e_1 \vdash \boldsymbol{\mathcal{O}}(d, \boldsymbol{a})\ @\ e_2}\ \texttt{change}$$

7.5.7 Id After is Id Before

In this example, we show that nodes do not “forget” identifiers, i.e., if a node knows identifier i after handling an event e_1, it will know that identifier right before handling any event, which happened directly after e_1. To prove this behavior, we show here that we can derive $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^-(i)\ @\ e_2$ from $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^+(i)\ @\ e_1$, [10]

$$\dfrac{\dfrac{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^+(i)\ @\ e_1}{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \subset\boldsymbol{\mathcal{I}}^+(i)\ @\ e_2}\ \Box_{\texttt{I}}}{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^-(i)\ @\ e_2}\ \lor_{\texttt{I1}}$$

7.5.8 Id Before is Id After

Here we show that the opposite direction also holds, by deriving $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^+(i)\ @\ e_1$ from $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^-(i)\ @\ e_2$ [11]

---

[9] See `DERIVED_RULE_owns_change_localle_true` in `code/model/CalculusSM.v`.

[10] See `DERIVED_RULE_id_after_is_id_before_true` in `code/model/CalculusSM.v`.

[11] See `DERIVED_RULE_id_before_is_id_after_true` in `code/model/CalculusSM.v`.

$$\dfrac{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^-(i) \mathbin{@} e_2 \qquad \Pi_0}{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\texttt{cut}$$

where Π_0 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle G, y : \mathrm{pred}^{=}(e_2) \equiv e_1\rangle\, H, h : \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} \mathrm{pred}^{=}(e_2) \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} \mathrm{pred}^{=}(e_2)}\,\texttt{hyp}}{\langle G, y : \mathrm{pred}^{=}(e_2) \equiv e_1\rangle\, H, h : \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} \mathrm{pred}^{=}(e_2) \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\texttt{sub}_{\texttt{C}}}{\langle G, y : e_1 \equiv \mathrm{pred}^{=}(e_2)\rangle\, H, h : \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} \mathrm{pred}^{=}(e_2) \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\equiv_{\texttt{sym}}}{\langle G, y : e_1 \subset e_2\rangle\, H, h : \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} \mathrm{pred}^{=}(e_2) \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\texttt{weak}}{\langle G, y : e_1 \subset e_2\rangle\, H, h : \subset\boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_2 \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\subset_{\texttt{E}} \qquad \Pi_1}{\langle G, y : e_1 \subset e_2\rangle\, H, h : \boldsymbol{\mathcal{I}}^-(i) \mathbin{@} e_2 \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\vee_{\texttt{E}}$$

where Π_1 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle G, y : e_1 \sqsubset e_2\rangle\, H, h : i = \mathsf{initId} \mathbin{@} e_2, x : \odot \mathbin{@} e_2 \vdash \neg\odot \mathbin{@} e_2}\,\neg\odot}{\langle G, y : e_1 \subset e_2\rangle\, H, h : i = \mathsf{initId} \mathbin{@} e_2, x : \odot \mathbin{@} e_2 \vdash \neg\odot \mathbin{@} e_2}\,\texttt{weak} \qquad \Pi_2}{\langle G, y : e_1 \subset e_2\rangle\, H, h : i = \mathsf{initId} \mathbin{@} e_2, x : \odot \mathbin{@} e_2 \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\texttt{cut}}{\langle G, y : e_1 \subset e_2\rangle\, H, h : i = \mathsf{initId} \wedge \odot \mathbin{@} e_2 \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\wedge_{\texttt{E}}$$

and Π_2 is:

$$\dfrac{\dfrac{}{\langle G, y : e_1 \subset e_2\rangle\, H, h : i = \mathsf{initId} \mathbin{@} e_2, x : \odot \mathbin{@} e_2 \vdash \odot \mathbin{@} e_2}\,\texttt{hyp} \qquad \Pi_3}{\langle G, y : e_1 \subset e_2\rangle\, H, h : i = \mathsf{initId} \mathbin{@} e_2, x : \odot \mathbin{@} e_2, n : \neg\odot \mathbin{@} e_2 \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\rightarrow_{\texttt{E}}$$

and Π_3 is:

$$\dfrac{}{\langle G, y : e_1 \subset e_2\rangle\, H, h : i = \mathsf{initId} \mathbin{@} e_2, y : (\exists_{\mathtt{n}}\boldsymbol{@}) \mathbin{@} e_2, x : \odot \mathbin{@} e_2, n : \bot \mathbin{@} e_2 \vdash \boldsymbol{\mathcal{I}}^+(i) \mathbin{@} e_1}\,\bot_{\texttt{E}}$$

7.5.9 Causal, Equal and First

We show here that we can derive $\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \sigma$ from $\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \odot \mathbin{@} e_2$ and $\langle G, y : e_1 \equiv e_2\rangle\, H \vdash \sigma$ [12]

$$\dfrac{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \odot \mathbin{@} e_2 \qquad \Pi_0}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \sigma}\,\texttt{cut}$$

where Π_0 is:

$$\dfrac{\dfrac{\langle G, y : e_1 \equiv e_2\rangle\, H \vdash \sigma}{\langle G, y : e_1 \equiv e_2\rangle\, H, w : \odot \mathbin{@} e_2 \vdash \sigma}\,\texttt{thin}_{\texttt{h}} \qquad \dfrac{\dfrac{}{\langle G, y : e_1 \sqsubset e_2\rangle\, H, w : \odot \mathbin{@} e_2 \vdash \neg\odot \mathbin{@} e_2}\,\neg\odot \qquad \Pi}{\langle G, y : e_1 \sqsubset e_2\rangle\, H, w : \odot \mathbin{@} e_2 \vdash \sigma}\,\texttt{cut}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, w : \odot \mathbin{@} e_2 \vdash \sigma}\,\mathsf{STR}_{\sqsubseteq}$$

and Π is:

$$\dfrac{\dfrac{}{\langle G, y : e_1 \sqsubset e_2\rangle\, H, w : \odot \mathbin{@} e_2 \vdash \odot \mathbin{@} e_2}\,\texttt{hyp} \qquad \dfrac{}{\langle G, y : e_1 \sqsubset e_2\rangle\, H, w : \odot \mathbin{@} e_2, z : \bot \mathbin{@} e_2 \vdash \sigma}\,\bot_{\texttt{E}}}{\langle G, y : e_1 \sqsubset e_2\rangle\, H, w : \odot \mathbin{@} e_2, z : \neg\odot \mathbin{@} e_2 \vdash \sigma}\,\rightarrow_{\texttt{E}}$$

---

[12] See `DERIVED_RULE_causalle_is_equal_if_first_true` in `code/model/CalculusSM.v`.

7.6 Typical System Assumptions and Consequences

In order to derive general results about distributed knowledge, such as in Section 7.7, let us first present some typical assumptions about knowledge, which we express here within LoCK (see the file called `code/model/CalculusSM.v` for more details). We illustrate in Section 7.8 that those assumptions can be validated and used for deriving specific properties of MinBFT.

LoCK's Assumptions. We first start by defining those assumptions, and we then explain their meaning:

$$\texttt{LID} = \forall_{\texttt{t}} \lambda t.\mathcal{L}(t) \rightarrow \prec(\mathcal{OD}(t)) \tag{7.2}$$

$$\texttt{KLD} = \forall_{\texttt{t}} \lambda t.\mathcal{K}^{+}(t) \rightarrow (\mathcal{K}^{-}(t) \vee \mathcal{L}(t) \vee \mathcal{OD}(t)) \tag{7.3}$$

$$\texttt{Mon} = (\exists_{\texttt{i}} \lambda i.\mathcal{I}^{-}(i) \wedge \mathcal{I}^{+}(i)) \vee (\exists_{\texttt{i}} \lambda i_1, i_2.i_1 < i_2 \wedge \mathcal{I}^{-}(i_1) \wedge \mathcal{I}^{+}(i_2)) \tag{7.4}$$

$$\begin{aligned} \texttt{New} = \forall_{\texttt{t}} \lambda t.\forall_{\texttt{i}} \lambda i_1, i_2.\ & (\mathcal{OD}(t) \wedge \mathcal{I}^{-}(i_1) \wedge \mathcal{I}^{+}(i_2)) \\ & \rightarrow \exists_{\texttt{i}} \lambda i.(i_1 < i \wedge i \leq i_2 \wedge \mathcal{HI}(t, i) \wedge \neg \mathcal{HI}(t, i_1)) \end{aligned} \tag{7.5}$$

$$\texttt{Uniq} = \forall_{\texttt{t}} \lambda t_1, t_2.\forall_{\texttt{i}} \lambda i.(\mathcal{OD}(t_1) \wedge \mathcal{OD}(t_2) \wedge \mathcal{HI}(t_1, i) \wedge \mathcal{HI}(t_2, i)) \rightarrow t_1 = t_2 \tag{7.6}$$

Through `LID`, we get to assume that if one learns some trusted data, it must be that it was disseminated by the corresponding trusted component that owns the data, i.e., the trusted data cannot simply appear from nowhere. Moreover, as stated by `KLD`, typically if we know some trusted information, then we either knew it before, or we just learned it, or we just disseminated it. Also, a typical property of trusted components is `Mon`, which says that the identifiers maintained by those components monotonically increase, i.e., either the recorded identifier stays the same (left disjunction), or it increases (right disjunction). In addition, as stated by `New`, if a trusted component is in charge of generating trusted identifiers, such an identifier i must be between the one recorded before and the one recorded after it generated i. Finally, trusted pieces of data disseminated by a trusted component at a given point in time are typically unique (`Uniq`).

Provenance of knowledge. From `KLD` (Equation 7.3) and using LoCK's induction on causal time rule (`ind`), we can derive:[13] $\mathcal{K}^{+}(t) \rightarrow \sqsubseteq\mathcal{L}(t) \vee \sqsubseteq\mathcal{OD}(t)$. Then,

---

[13] See the lemma called `DERIVED_RULE_trusted_KLD_implies_or_true` in the file called `code/model/CalculusSM.v`.

using `LID` (Equation 7.2), and using a similar *collapsing* result as the one presented in Section 7.5 above (to collapse $\sqsubseteq\!\prec$ into $\preceq$ here), we can further derive:[14]

$$\mathcal{K}^{+}(t) \rightarrow \preceq(\mathcal{OD}(t)) \tag{7.7}$$

Progress when disseminating. `Mon` and `New` can be combined into this following derived rule, which captures that progress is made every time a trusted piece of data is disseminated, i.e., disseminating trusted pieces of data must get the recorded identifier to increase:[15]

$$\frac{\Lambda[e']\langle G\rangle\, H \vdash \texttt{Mon} \wedge \texttt{New} \mathbin{@} e' \qquad \langle G\rangle\, H \vdash \mathcal{OD}(t) \mathbin{@} e}{\langle G\rangle\, H \vdash \exists_{\mathrm{i}}\lambda i, i_1, i_2. \mathcal{I}^{-}(i_1) \wedge \mathcal{I}^{+}(i_2) \wedge \mathcal{HI}(t,i) \wedge \neg\mathcal{HI}(t,i_1) \wedge i_1 < i \wedge i \leq i_2 \mathbin{@} e} \tag{7.8}$$

This rule is a straightforward combination of `Mon` and `New`, which is convenient to prove results such as the "uniqueness over time" derived rule 7.9 presented below. (1) Either no progress is made at e, i.e., $\mathcal{I}^{-}(i)$ and $\mathcal{I}^{+}(i)$ for some i. We then derive a contradiction, because by `New`, we must have $i < i$, and we conclude using `irrefl`. (2) Or some progress is made at e, and we conclude using `New`.

Uniqueness over time. `Uniq` can be generalized to trusted pieces of data generated at *any* point in space/time by a trusted component. Namely, we can derive the following rule within LoCK:[16]

$$\frac{\Lambda[e'] \quad \langle G\rangle\, H \vdash \texttt{Mon} \wedge \texttt{New} \wedge \texttt{Uniq} \mathbin{@} e' \qquad \begin{array}{c}\langle G\rangle\, H \vdash \mathcal{OD}(t_1) \wedge \mathcal{HI}(t_1,i) \wedge @(\boldsymbol{a}) \mathbin{@} e_1 \\ \langle G\rangle\, H \vdash \mathcal{OD}(t_2) \wedge \mathcal{HI}(t_2,i) \wedge @(\boldsymbol{a}) \mathbin{@} e_2\end{array}}{\langle G\rangle\, H \vdash t_1 = t_2 \mathbin{@} e} \tag{7.9}$$

This derived rule is critical to prove Theorem 2 in Section 7.7. It says that if two trusted pieces of data t_1 and t_2 are disseminated at e_1 and e_2, respectively, such that they have the same identifier and that e_1 and e_2 happened at the same location $\boldsymbol{a}$, then t_1 must be equal to t_2. We can derive this result using LoCK's trichotomy rule $\texttt{tri}$. If $e_1 = e_2$ then we conclude using `Uniq`. If e_1 happened locally before e_2 (and similarly if e_2 happened before e_1) then from `Mon`, and using LoCK's induction on causal time rule `ind`, we derive that the identifier i_1 recorded

---

[14] See the lemma called `DERIVED_RULE_trusted_KLD_implies_gen_true` in the file called `code/model/CalculusSM.v`.

[15] See the lemma called `DERIVED_RULE_disseminate_implies_count_ex_true` in the file called `code/model/CalculusSM.v`.

[16] See the lemma called `DERIVED_RULE_trusted_disseminate_unique_ex_true` in the file called `code/model/CalculusSM.v`.

after e_1 must be less than or equal to the one, say i_2, recorded before e_2. Moreover, from `New`, we derive that i is less than or equal to i_1 and i_2 is strictly less than i. Finally, we conclude using the `trans` and `irrefl` derivation rules.

7.7 Distributed Lifting

Using the above mentioned rules and assumptions, we derived among other things the following lemma (see below for a proof sketch):[17]

Theorem 2 (Distributed Lifting)**.** *The following rule is derivable within LoCK:*

$$\frac{\begin{array}{ll}\Lambda[e'] & \langle G\rangle\, H \vdash \texttt{LID} \land \texttt{KLD} \land \texttt{Mon} \land \texttt{New} \land \texttt{Uniq} \mathbin{@} e' \\ & \langle G\rangle\, H \vdash \mathcal{K}^{+}(t_1) \land \mathcal{O}(t_1, \boldsymbol{a}) \land \mathcal{G}(d_1, t_1) \land \mathcal{HI}(t_1, i) \mathbin{@} e_1 \\ & \langle G\rangle\, H \vdash \mathcal{K}^{+}(t_2) \land \mathcal{O}(t_2, \boldsymbol{a}) \land \mathcal{G}(d_2, t_2) \land \mathcal{HI}(t_2, i) \mathbin{@} e_2\end{array}}{\langle G\rangle\, H \vdash d_1 = d_2 \mathbin{@} e}$$

This derived rule allows lifting properties of trusted components to the level of a distributed system. It states that if all assumptions presented in Section 7.6 are satisfied at all events; and at event e_1 some node knows some trusted information t_1, owned by $\boldsymbol{a}$, with identifier i, and generated from some data d_1; and similarly at e_2 some node knows some trusted information t_2, also owned by $\boldsymbol{a}$ and with identifier i, and generated from d_2; then the two pieces of data d_1 and d_2 must be equal. This is the crux of proving the safety properties of MinBFT's normal-case operation (see Chapter 8).

Proof Sketch 2. We derive here Theorem 2 essentially from the "provenance of knowledge" formula 7.7 and the "uniqueness over time" derived rule 7.9 presented above. From $\mathcal{K}^{+}(t_1)$ (at e_1) and $\mathcal{K}^{+}(t_2)$ (at e_2), we can derive using Equation 7.7 that there must be two previous events e'_1 and e'_2 such that t_1 was disseminated at e'_1 and t_2 was disseminated at e'_2 (by their rightful owners). Because $\boldsymbol{a}$ owns both t_1 and t_2 then it must be that e'_1 and e'_2 happened at the same location. We can then derive that $t_1 = t_2$ from the derived rule 7.9. Finally, we derive that $d_1 = d_2$ using LoCK's `ldata` inference rule.

7.8 Knowledge and MinBFT

As we already mentioned above, proof of the MinBFT's agreement property is a direct consequence of the distributed lifting lemma (see Theorem. 2), presented in Section 7.7. Nevertheless, to verify properties about MinBFT using LoCK, we

---

[17] See the lemma called `DERIVED_RULE_trusted_knowledge_unique3_ex_true` in the file called `code/model/CalculusSM.v`.

had to instantiate the parameters presented in Figure 7.1.[18] We only discuss here some of the most interesting parameters. The interested reader is invited to look at our Coq implementation for more details. We instantiate Data with a type that contains both UIs and triples of the form view/request/UI, which is the canonical information contained in most MinBFT messages. Trust is instantiated with the type of UIs, and Identifier is instantiated with the type of counter values. The component name mem is instantiated with LOGcomp; while trust is instantiated with USIGcomp. The predicate know is instantiated by a predicate that states that the data is stored in the log. Finally sys is instantiated with MinBFTsys.

As opposed to the USIG-based version, to reason about the TrInc-based version, we have instantiated Identifier with the type of counter value lists, because TrInc maintains multiples counters. We then say that a UI ui, with counter id i and counter value c, has identifier l (a list of counter values) if the counter value in l corresponding to i is c (the other counters can have any values).

Because Theorem 2 relies on some assumptions (see Section 7.6), we had to prove that those are indeed true about our MinBFT implementations. `LID` differs from the others because it is not a direct consequence of MinBFT's behavior, but follows from our generic AXIOM_auth_messages_were_sent_or_byz HyLoE assumption, which is a constraint on event orderings that rules out impossible message transmissions. It states that if a node receives a valid piece of data d (in the sense that its authenticity has been checked), then either (1) a correct node sent d following the protocol; or (2) some arbitrary event happened, for which no information is available, and some node sent d either authenticating it itself or impersonating some other node; or (3) some arbitrary event happened at which a trusted component generated d. `KLD` is a straightforward consequence of the way MinBFT accumulates knowledge by logging messages: a message is logged if it is generated or received. We proved `Mon` using the local lifting Theorem 1, described in Section 6.4. It is true because USIGs (and TrIncs) indeed maintain monotonic counters. `New` and `Uniq` are straightforwardly true because USIGs always increment their counters before generating a new UI.

---

[18] See the files called `code/MinBFT/MinBFTkn0.v`, `code/MinBFT/MinBFTkn.v` and `code/MinBFT/TrInckn.v` in our implementation.

Chapter 8

Hybrid Protocol Case Studies: USIG- and TrInc-based MinBFT

As explained in Chapter 5 the main goal of this thesis is to design a general, reusable and extensible framework that can be used for proving correctness of both, homogeneous as well as hybrid fault-tolerant protocol. We already showed in Chapter 5 that our framework can be used for proving correctness of homogeneous fault-tolerant protocols. In this chapter, we show that our framework can be used for proving correctness of hybrid fault-tolerant protocols as well.

In Section 8.1, we show how we exercised our framework by implementing and verifying two versions of the seminal MinBFT hybrid protocol [Ver+13]: one based on USIGs (as in the original version), and one based on TrIncs [Lev+09]. As explained in Section 2.1.3, USIGs and TrIncs have different pros and cons that make them both interesting to use and verify correct. We proved the agreement property of both versions using Theorem 2, which we proved within LoCK (see Section 7.7). Because other hybrid protocols rely on trusted components that are similar to USIGs and TrIncs, we believe that our methodology can also be used to verify the correctness of other hybrid protocols such as [Kap+12]; [BDK17]; [Chu+07]. We finish this chapter by explaining how we obtained executable MinBFT code (see Section 8.2).

8.1 Verified MinBFT properties

Using framework presented in this thesis we proved the following Coq lemma, which is critical to prove the safety of MinBFT's normal-case operation (the → direction is the agreement property):[1]

---

[1] See the files called: `code/MinBFT/MinBFTagreement_iff.v` and `code/MinBFT/TrIncagreement_iff.v`.

```
Lemma agreement_iff :
  ∀ (eo : EventOrdering) (e1 e2 : Event) (r1 r2 : Request) (i1 i2 : nat) (l1 l2 : list name),
  AXIOM_auth_messages_were_sent_or_byz eo MinBFTsys
  → ((send_accept r1 i1 l1) ∈ MinBFTsys ⇝ e1)
  → ((send_accept r2 i2 l2) ∈ MinBFTsys ⇝ e2)
  → (i1 = i2 ↔ r1 = r2).
```

This lemma states that if a correct replica executes a request r with counter value $i1$, then no other correct replica will execute the same request with a different counter value $i2 \neq i1$; and two correct replicas cannot execute two different requests with the same counter value. In addition to the executed request, these messages include the counter value generated for the request by the primary. Actually, in our implementations, replicas send "accept" messages whenever they execute a request. As mentioned above, this lemma is a straightforward consequence of the general Theorem 2 proved within LoCK and presented in Section 7.7.

As another straightforward consequence of Theorem 2, we have also proved the following agreement property:[2]

```
Lemma agreement :
  ∀ (eo : EventOrdering) (e1 e2 : Event) (r1 r2 : Request) (i : nat) (l1 l2 : list name),
  AXIOM_auth_messages_were_sent_or_byz eo MinBFTsys
  → In (send_accept r1 i l1) (M_output_sys_on MinBFTsys e1)
  → In (send_accept r2 i l2) (M_output_sys_on MinBFTsys e2)
  → (is_replica e1 ∧ is_replica e2)
  → r1 = r2.
```

Despite the fact that we only verified the normal-case operation of MinBFT, we also had to reason about Byzantine failures, because in the normal-case operation primary is believed to be correct, but the other processes might behave arbitrarily (i.e., they might be Byzantine).

Remark 2. In our proofs we only used the fact that trusted/trustworthy counters are monotonic, possibly with gaps because of the additional restriction that replicas have to execute requests one at a time without gaps. This way, the no-gap condition does not need to be coming from trusted/trustworthy components.

Remark 3. Although properties and proofs we show here do not require reasoning about intersecting quorums, reasoning about intersecting quorums is necessary in case of a view-change. For example, what could happen is that the primary p proposes some message m to some replica r_i and dies right after that. Than if r_i applies m straightaway without gathering a quorum, it might be the only one doing so, i.e., replicas might diverge (because the other replicas might run a view-change and just ignore m).

---

[2] See the lemma called `agreement` in the file called `code/MinBFT/MinBFTagreement.v` in our implementation.

Differences from the Original Proof. As it turns out, our proof of agreement_iff is significantly simpler than the original pen-and-paper proof [Ver10, pp.151–153]. The original proof of the ← direction, which we claim here to be unnecessarily convoluted, goes as follows: given that a quorum of $f+1$ replicas have committed *(r,i1)*, and a quorum of $f+1$ replicas have committed *(r,i2)*, there must be a replica at the intersection of the two quorums that has committed both *i1* and *i2* (since there are $2f+1$ replicas in total). Then, their proof goes by cases on whether or not that replica and the primary are correct, leading to four cases. However, such a replica at the intersection of the two quorums is not required because if a replica has executed a request, it must have received at least one prepare/commit for this request containing a UI created by the primary's USIG. Therefore, we can deduce that the primary's USIG must have created UIs for the two counter values corresponding to the two quorums mentioned above. We can then trace back these two counters to the time that primary's USIG generated UIs for them, and conclude using monotonicity. Note that we do not need to go by cases on whether replicas are correct or not because trusted components of hybrid systems (USIGs here) cannot be tampered with, and the above reasoning rely solely on properties that the system inherits from the trusted components. Thanks to the operators presented in this thesis, such as $ls \rightsquigarrow e$ (described in Section 6.3), we can always reliably access these trusted components because they cannot be compromised and because in the context of such safety proofs, they must have been running at the time they outputted values (i.e., at the time they created UIs in the case of USIGs). As a matter of fact, agreement_iff holds even if the primary, except for its USIG, has been compromised. Moreover, based on the reasoning we use to prove agreement of normal-case of MinBFT, as well as the fact that in the normal-case primary is assumed to be correct, one can also conclude that agreement of MinBFT's normal-case holds for any number of replicas.

8.2 MinBFT Extraction and Evaluation

In this section, we first explain how we obtain executable OCaml code from our Coq implementation of MinBFT, such that USIGs run inside trusted environment. Next, we compare performances of our verified version of MinBFT with our verified version of PBFT. Finally, we conclude this section by presenting our trusting computing base.

Extraction. We use Coq's extraction mechanism to obtain executable OCaml code from our distributed systems implemented in MoC (see Section 6.2.1). However, because we want to run the different components of a local subsystem separately (i.e., execute the trusted ones within trusted environments such as Intel SGX), the monad structure is "erased" during extraction.[3] Instead, a separate module is created for each component, and calls to components are extracted to calls to those modules. In addition, the functional states of MoC components are turned into imperative ones within those modules. Running the components of a local subsystem separately enables executing the trusted ones within trusted environments, in our case Intel SGX enclaves.

Trusted execution. We use Graphene-SGX [TPV17] in order to run MinBFT's trusted USIG components inside Intel SGX enclaves.[4] Graphene-SGX is a library for running unmodified applications inside SGX enclaves. Because Graphene-SGX's driver closes enclaves after each call, and because only part of the extracted code is meant to run inside SGX enclaves, our SGX-based runtime environment uses a TCP interface for replicas to interact with USIGs running in Graphene-SGX enclaves. Moreover, because to the best of our knowledge, at the time of writing, Intel SGX only supports C applications, our SGX-based runtime environment includes C wrappers around the OCaml code of the USIG components, as well as OCaml wrappers around the TCP interface implemented in C (these wrappers use [19g]). Note that to support calling the interfaces of trusted components through the above mentioned TCP interface, one has to write custom serializers/deserializers.[5] We leave it for future work to generate those automatically.

Remark 4. Instead of Graphene-SGX, one could port the OCaml runtime inside Intel SGX, i.e., create OCalls' for all required OCaml runtime library calls that are not included in the SGX libraries. Beside the fact that this solution can bring security issues, because each OCall requires switching between enclave and non-enclave memory it might be very slow.

Evaluation. As for verified PBFT (see Section 5.3), we ran our experiments using a desktop with 16GB of memory and 8 i7-6700 cores running at 3.40GHz. We also used Async [19a] to handle sender/receiver threads. The experiments we report here are with one client and the replicated service is a state machine whose state is a number and whose operation is addition.

We ran a local simulation to measure the performance of our MinBFT implementation without network and signatures: when 1 client sends 1 million requests, it takes on average 1.7 microseconds for the client to receive $f+1$ ($f=1$) replies.

---

[3] The monad erasure we perform is very simple and standard (see `code/MinBFT/runtime_w_sgx/MinBFTinstance.v`).

[4] See `code/MinBFT/runtime_w_sgx/README.md` or Appx. C for further details.

[5] See for example `code/MinBFT/runtime_w_sgx/tcp_client.c` and `code/MinBFT/runtime_w_sgx/tcp_server.c`.

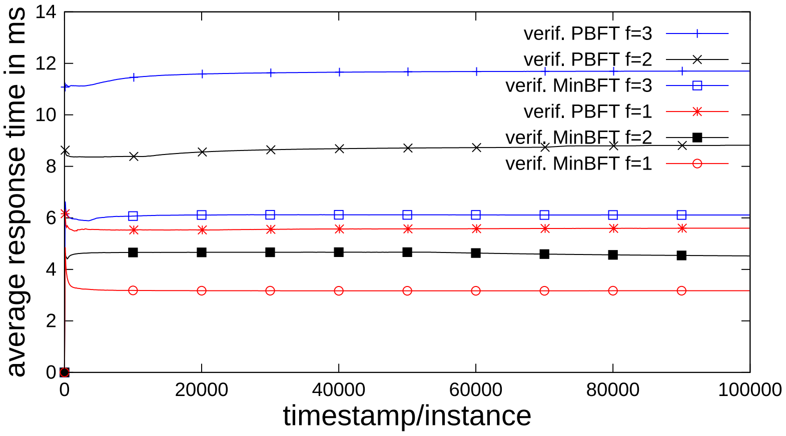


Figure 8.1: Performance of verified MinBFT ($f \in \{1, 2, 3\}$) on a single machine

We obtained exactly the same numbers when we run our verified version of PBFT (see Section 5.3).

As the Figure 8.1 shows, for $f \in \{1, 2, 3\}$ the average latency of our USIG-based version of MinBFT is lower than the average latency of the verified version of PBFT presented in Section 5.3. Although Graphine-SGX incurs some overhead, we believe that our MinBFT implementation is faster because: (1) MinBFT uses less communication steps than PBFT; and (2) our MinBFT implementation uses less expensive crypto (i.e., HMACs as opposed to RSA in Section 5.3). When we run our MinBFT implementation on several machines, connected via gigabit Ethernet network, we obtained similar results (see Figure 8.2). In order to conduct this experiment, we add a functionality during extraction that allows replicas to garbage collect from time to time.

We also compared our verified MAC version of PBFT with our MinBFT implementation (see Figure 8.3). In this case both implementations have similar performance because PBFT clients do not use PK for signing their requests (see [Cas01, Page 41]) while MinBFT clients do.

8.3 Hybrid Protocols: TCB & Proof Effort

Trusted Computing Base. The TCB of our system is composed of: (1) the fact that our HyLoE model faithfully reflects the behavior of hybrid systems (see Section 6.1); (2) the validity of the assumptions described in Section 7.6; (3) Coq's logic and implementation; (4) our runtime environment implemented in OCaml (Section 8.2); (5) and the hardware and software on which our framework is running

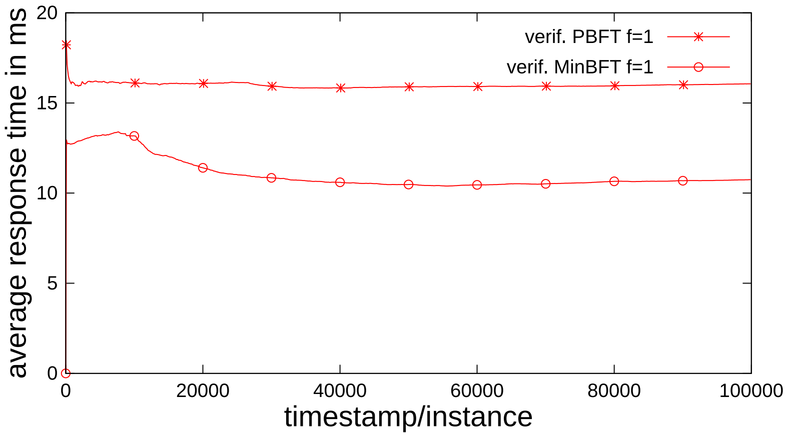


Figure 8.2: Performance of verified MinBFT ($f = 1$) on several machines

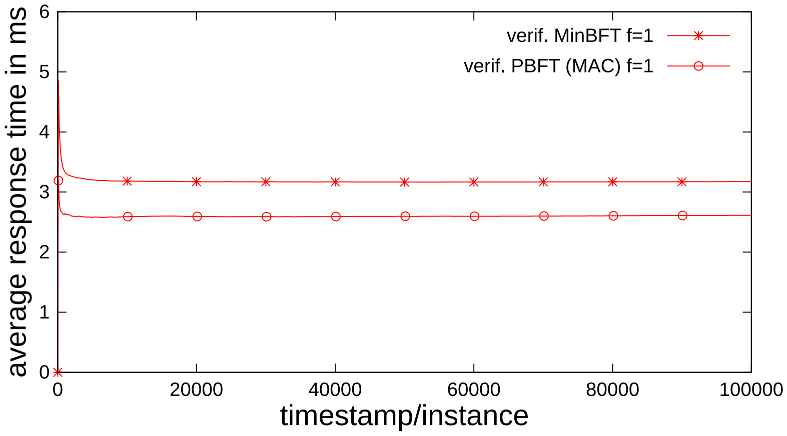


Figure 8.3: Comparison of verified PBFT based on MACs and verified MinBFT

(including the trusted environments such as SGX).

Proof Effort. Developing hybrid part of our framework and partially verifying MinBFT took us about one person year, i.e., developing the whole framework presented in this thesis took us about two and a half person years. Our hybrid part of the model is about 10000 lines of specifications and 8500 lines of proofs, i.e., the whole model presented in this thesis consists of about 1450 lines of specifications and 12500 lines of proofs. Our MinBFT proofs (USIG-based and TrInc-based) are about 7000 lines of specifications and 4500 lines of proofs.

Chapter 9

Conclusions and Future Work

This thesis presents a theorem-prover based framework that can be used for reasoning about implementations of homogeneous fault-tolerant protocols, which rely on synchrony or asynchrony, as well as about implementations of hybrid fault-tolerant protocols. Our framework comes with:

- HyLoE—a logic to model homogeneous/hybrid systems;
- MoC—a programming language to implement systems composed of interacting components;
- LoCK—a knowledge theory to reason about systems at a high-level of abstraction without having to worry about low-level implementation details.

In addition, our framework introduces novel proof techniques to lift properties about (trusted) components to the level of distributed systems. To show usability of this framework, we formally verified several BFT-SMR protocols that rely on different system and fault models: the seminal synchronous protocol called SM, the seminal practical asynchronous protocol called PBFT and the seminal hybrid protocol called MinBFT.

Although the main goal of this thesis was to design a general, reusable and extensible framework, much remains to be done in this field. In the paragraphs below, we provide more details.

Replicated service. Ideally, both the replication mechanism and the instances of the replicated service should be verified. However, we focus here on the replication mechanism, which has to be done only once, while formal verification of the instances of the replicated service needs to be done for every service and for every replica instance.

Diversity and rejuvenation. To enable correct functioning of the system during its lifetime and in order to avoid persistent and shared vulnerabilities, replicas need to be rejuvenated periodically [CL02]; [Sou07], need to be diverse enough [Jaj+11], and ideally need to be physically far apart. We leave reasoning about diversity and rejuvenation for future work.

Collision resistant assumption. Our current collision resistant assumption (see Section 3.1.7.4) is too strong because it is always possible to find two distinct messages that are hashed to the same hash. We leave it to future work to turn it into a more realistic probabilistic assumption.

Linearizability. The main goal of this thesis was not to prove linearizability, especially because in [Wil+15]; [Woo+16] the authors present a methodology for proving linearizability. Our main objective was to come up with an innovative framework to reason about typical safety properties (agreement and validity [CGR11]). We leave extending our framework with reusable patterns for proving linearizability for future work.

IC2. We demonstrate that our framework can be used to prove properties of synchronous BFT protocols by proving that both our implementations of the SM protocol satisfy the safety part of the *IC1* property, originally introduced in [LSP82]. We leave verifying *IC2*, which is also introduced in [LSP82], as well as the liveness part of *IC1* for future work.

Reason about garbage collection and view-changes at knowledge level. Although we have proved a critical safety property of PBFT, including its garbage collection and view-change procedures (which are essential in practical protocols), we have not yet developed generic abstractions to specifically reason about garbage collection and view-changes that can be reused in other protocols. We leave developing these abstractions for future work.

Different system assumptions of hybrids. Our framework supports reasoning about different failure assumptions of hybrid systems—crash vs. Byzantine. We leave adding support for reasoning about the different system assumptions of hybrid systems for future work—synchrony vs. asynchrony.

MoC to imperative code. In the future, we would like to implement a formally verified compiler from a programming language we use for implementing systems composed of interacting components (MoC) to imperative code.

Improve compositionality of MoC. Our current version of MoC is build such that a local subsystem is defined as a pair of a main component and a list of components. This means that two local subsystems can be composed either by: (1) introducing a new main component and merging lists of components into one list; or (2) decomposing one of those local subsystems into a list of components (this list would also include its main component) and then by appending that list to the list of components of the other system. In the future, we plan to develop a version of MoC which will be more compositional.

Local lifting lemmas for higher-level components. As explained in Section 6.4, our local lifting lemma (see Theorem 1) assumes that trusted components need to be at level 1. We leave developing local lifting lemmas for higher-level components for future work.

Automation of LoCK. In the future, we would like to extend our knowledge theory (LoCK) so that some proofs about distributed knowledge could be automated. We have started developing proof tactics similar to Coq's *intro* and *destruct* (see Chapter B). In addition, we would like to develop both simple "brute-force" proof search engines, and decision procedures for fragments of LoCK.

LoCK to running code. We would also like to investigate whether our knowledge theory (LoCK) specifications could be compiled to running code.

MinBFT's garbage collection and view-change. We leave implementing and formally verifying MinBFT's garbage collection and view-change mechanisms for future work, because the normal phase operation provides the necessary and sufficient context to address the challenges of reasoning about hybrid systems.

Liveness/timeliness. In the future, we plan to also tackle liveness/timeliness. Indeed, proving the safety of a distributed system is far from being enough: a protocol that does not run (which is not live) is useless. Following the same line of reasoning, we want to tackle timeliness because, for real world systems, it is not enough to prove that a system will *eventually reply*. One often desires that the system replies in a timely fashion.

Appendix A

Opening the `LID`

A.1 Primitive Principles Behind `LID`

Section 7.6 presents typical assumptions about knowledge, expressed within LoCK. In particular, it presents the following `LID` assumption in Equation 7.2, which states that if one learns about a trusted piece of data, then this trusted piece of data must have been disseminated by its owner in the past:

$$\lambda t.\boldsymbol{\mathcal{L}}(t) \rightarrow \prec(\boldsymbol{\mathcal{OD}}(t))$$

As mentioned in Section 7.8, `LID` essentially follows from our generic communication assumption called AXIOM_auth_messages_were_sent_or_byz.[1] Given a distributed system such as MinBFT, it is not complicated to prove that `LID` (its HyLoE interpretation) holds assuming AXIOM_auth_messages_were_sent_or_byz. However, it requires using induction in HyLoE, which we are trying to avoid: we are aiming at having all the inductive reasoning done in LoCK in order to keep the reasoning done in HyLoE as simple as possible. The reason for the inductive nature of this proof is that `LID` allows going back directly to the owner of the learned trusted piece of data, while AXIOM_auth_messages_were_sent_or_byz only allows getting back to *some* point in space/time, where the trusted piece of data was disseminated: it does not have to be disseminated by the owner at that point because the data might have been relayed by an intermediary node. As it turns out, `LID` can be derived within LoCK from more primitive principles, which we present next.[2]

Let `Com` be the following LoCK expression:

$$\forall_{\mathtt{t}}\lambda t.\boldsymbol{\mathcal{L}}(t) \rightarrow (\exists_{\mathtt{d}}\lambda d.\prec(\boldsymbol{\mathcal{ND}}(d) \wedge t{\in}d \wedge \boldsymbol{\mathcal{C}})) \vee \prec\boldsymbol{\mathcal{OD}}(t)$$

---

[1] see `code/model/ComponentAxiom.v` for more details

[2] See `code/model/CalculusSM_derived4.v` for more details.

As for $\boldsymbol{\mathcal{C}}$, $t{\in}d$ is not discussed in the main body of this thesis because it is scarcely used. It expresses that the trusted piece of data t occurs in the piece of data d. $\boldsymbol{\mathcal{N}\mathcal{D}}(d)$ is defined as $\boldsymbol{\mathcal{N}} \wedge \boldsymbol{\mathcal{D}}(d)$, where $\boldsymbol{\mathcal{N}} = \exists_{\texttt{n}}\lambda \boldsymbol{a}.@(\boldsymbol{a})$. We have proved that `Com` is a straightforward consequence of the communication axiom AXIOM_auth_messages_were_sent_or_byz, i.e., we have proved (assuming a few simple properties that relate HyLoE parameters and LoCK parameters):[3]

$$\begin{aligned} &\forall eo \in \mathsf{EO}.\ \text{AXIOM\_auth\_messages\_were\_sent\_or\_byz}\ eo\ \mathsf{sys} \\ &\qquad \rightarrow \forall e \in \mathsf{Event}(eo).[\![\texttt{Com}]\!]_{e} \end{aligned} \tag{A.1}$$

We can then derive the following derived rule:

$$\dfrac{\Lambda[e'] \quad \langle G\rangle\ H \vdash \texttt{Com} \wedge \texttt{KLD} \wedge \texttt{NKD} \wedge \texttt{KIK} @ e'}{\langle G\rangle\ H \vdash \texttt{LID} @ e}\ \texttt{LID} \tag{A.2}$$

where `KLD` is defined in Equation 7.3 in Section 7.6, and

$$\begin{aligned} \texttt{NKD} &= \forall_{\texttt{d}}\lambda d.\boldsymbol{\mathcal{N}\mathcal{D}}(d) \rightarrow \boldsymbol{\mathcal{C}} \rightarrow \boldsymbol{\mathcal{K}}^{+}(d) \\ \texttt{KIK} &= \forall_{\texttt{d}}\lambda d.\forall_{\texttt{t}}\lambda t.\boldsymbol{\mathcal{K}}^{+}(d) \rightarrow t{\in}d \rightarrow \boldsymbol{\mathcal{K}}^{+}(t) \end{aligned}$$

`NKD` says that nodes must know about the pieces of data they disseminate; while `KIK` says that if a node know a piece of data, then it must know about all the trusted pieces of data contained in that piece of data.

A.2 A Proof of the `LID` Derived Rule

Let us now discuss the proof of the `LID` derived rule.[4] First of all, we show that we can derive $\boldsymbol{\mathcal{K}}^{+}(t)$ from `NKD`, `KIK`, $\boldsymbol{\mathcal{N}\mathcal{D}}(d)$, $\boldsymbol{\mathcal{C}}$, and $t{\in}d$ (we combine some steps for readability):

$$\dfrac{\langle G\rangle\ H \vdash \texttt{NKD} @ e \qquad \dfrac{\langle G\rangle\ H \vdash \boldsymbol{\mathcal{N}\mathcal{D}}(d) @ e \quad \langle G\rangle\ H \vdash \boldsymbol{\mathcal{C}} @ e \quad \dfrac{\langle G\rangle\ H \vdash \texttt{KIK} @ e \quad \Pi}{\langle G\rangle\ H, x : \boldsymbol{\mathcal{K}}^{+}(d) @ e \vdash \boldsymbol{\mathcal{K}}^{+}(t) @ e}\ \texttt{cut} + \texttt{thin}_{\texttt{h}}}{\langle G\rangle\ H, x : \texttt{NKD} @ e \vdash \boldsymbol{\mathcal{K}}^{+}(t) @ e}\ \forall_{\texttt{E}} + \rightarrow_{\texttt{E}} + \texttt{thin}_{\texttt{h}}}{\langle G\rangle\ H \vdash \boldsymbol{\mathcal{K}}^{+}(t) @ e}\ \texttt{cut}$$

where Π is

$$\dfrac{\dfrac{}{\langle G\rangle\ H, x : \boldsymbol{\mathcal{K}}^{+}(d) @ e \vdash \boldsymbol{\mathcal{K}}^{+}(d) @ e}\ \texttt{hyp} \qquad \langle G\rangle\ H \vdash t{\in}d @ e \quad \Pi_1}{\langle G\rangle\ H, x : \boldsymbol{\mathcal{K}}^{+}(d) @ e, y : \texttt{KIK} @ e \vdash \boldsymbol{\mathcal{K}}^{+}(t) @ e}\ \forall_{\texttt{E}} + \rightarrow_{\texttt{E}} + \texttt{thin}_{\texttt{h}}$$

and Π_1 is

$$\dfrac{}{\langle G\rangle\ H, x : \boldsymbol{\mathcal{K}}^{+}(d) @ e, y : \boldsymbol{\mathcal{K}}^{+}(t) @ e \vdash \boldsymbol{\mathcal{K}}^{+}(t) @ e}\ \texttt{hyp}$$

---

[3] See `ASSUMPTION_authenticated_messages_were_sent_or_byz_true` in `code/model/CalculusSM_derived4.v`.

[4] See `DERIVED_RULE_implies_all_trusted_learns_if_gen2_true` in `code/model/CalculusSM_derived4.v` for more details.

The rule we just derived is then:

$$\dfrac{\langle G\rangle\ H \vdash \mathtt{NKD} \mathbin{@} e \quad \langle G\rangle\ H \vdash \mathtt{KIK} \mathbin{@} e \quad \langle G\rangle\ H \vdash \boldsymbol{\mathcal{ND}}(d) \mathbin{@} e \quad \langle G\rangle\ H \vdash \boldsymbol{\mathcal{C}} \mathbin{@} e \quad \langle G\rangle\ H \vdash t{\in}d \mathbin{@} e}{\langle G\rangle\ H \vdash \boldsymbol{\mathcal{K}}^{+}(t) \mathbin{@} e}\ \mathtt{DITK}$$

Let us now go back to Equation A.2. We proved the validity of this derived rule in LoCK by induction. As it turns out, we used a different rule than `ind`, which allows us to go by induction on the *happened before* relation, as opposed to `ind`, which goes by induction on the *direct predecessor* relation (from now on we will call both rules `ind` for simplicity):[5]

$$\dfrac{\Lambda[e] \quad \langle G\rangle\ H \vdash (\forall_{\prec}\tau) \rightarrow \tau \mathbin{@} e}{\langle G\rangle\ H \vdash \tau \mathbin{@} e}\ \mathtt{ind}$$

Note the use of the $\forall_{\prec}\tau$ operator. This (primitive) operator is also not discussed in the main body of this thesis for space reasons and because it is only used scarcely. Its semantics is:

$$[\![\forall_{\prec}\tau]\!]_{e} = \forall e' \prec e. [\![\forall_{\prec}\tau]\!]_{e'}$$

In our proof of Equation A.2, we will also use the following derived rule, which is similar to Equation 7.7, where $\forall_{\preceq}\tau = \forall_{\prec}\tau \vee \tau$:[6]

$$\dfrac{\Lambda[e'] \quad \langle G\rangle\ H \vdash \mathtt{KLD} \mathbin{@} e' \quad \langle G\rangle\ H \vdash \forall_{\preceq}\mathtt{LID} \mathbin{@} e}{\langle G\rangle\ H \vdash \boldsymbol{\mathcal{K}}^{+}(d) \rightarrow \preceq(\boldsymbol{\mathcal{OD}}(d)) \mathbin{@} e}\ \mathtt{KID}$$

In addition, we will also use the following derived rule, which strengthens a $\forall_{\preceq}$ to a $\forall_{\prec}$ by navigating to a later point in space/time (from e' to e below):[7]

$$\dfrac{\langle G, u : e'{\prec}e\rangle\ H \vdash \forall_{\prec}\tau \mathbin{@} e}{\langle G, u : e'{\prec}e\rangle\ H \vdash \forall_{\preceq}\tau \mathbin{@} e'}\ \mathtt{STR}\forall_{\preceq}$$

Finally, we will also use the following derived rule, which allows weakening $\prec$ to $\preceq$ by navigating to an earlier point in space/time, i.e., from e to e' below:[8]

$$\dfrac{\langle G, u : e'{\prec}e\rangle\ H \vdash \preceq\tau \mathbin{@} e'}{\langle G, u : e'{\prec}e\rangle\ H \vdash \prec\tau \mathbin{@} e}\ \mathtt{WEAK}{\prec}$$

Let us now derive Equation A.2:

$$\dfrac{\dfrac{\dfrac{\langle G\rangle\ H \vdash \mathtt{Com} \mathbin{@} e \quad \dfrac{\Pi_1 \quad \dfrac{}{\langle G\rangle\ H, x : \forall_{\prec}\mathtt{LID} \mathbin{@} e, y : \boldsymbol{\mathcal{L}}(t) \mathbin{@} e, z : \prec\boldsymbol{\mathcal{OD}}(t) \mathbin{@} e \vdash \prec(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\ \mathtt{hyp}}{\langle G\rangle\ H, x : \forall_{\prec}\mathtt{LID} \mathbin{@} e, y : \boldsymbol{\mathcal{L}}(t) \mathbin{@} e, z : \mathtt{Com} \mathbin{@} e \vdash \prec(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\ \forall_{\mathtt{E}}{+}\rightarrow_{\mathtt{E}}{+}\vee_{\mathtt{E}}}{\langle G\rangle\ H, x : \forall_{\prec}\mathtt{LID} \mathbin{@} e, y : \boldsymbol{\mathcal{L}}(t) \mathbin{@} e \vdash \prec(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\ \mathtt{cut} + \mathtt{thin}_{\mathtt{h}}}{\langle G\rangle\ H \vdash \forall_{\prec}\mathtt{LID} \rightarrow \mathtt{LID} \mathbin{@} e}\ \rightarrow_{\mathtt{I}} + \forall_{\mathtt{I}}}{\langle G\rangle\ H \vdash \mathtt{LID} \mathbin{@} e}\ \mathtt{ind}$$

---

[5] See `code/model/PRIMITIVE_RULE_induction_true` for a proof of the validity of this rule.

[6] See `DERIVED_RULE_KLD_implies_gen2_true` in `code/model/CalculusSM_derived4.v`.

[7] See `DERIVED_RULE_forall_node_before_eq_trans_true` in `code/model/CalculusSM_derived4.v`.

[8] See `DERIVED_RULE_unhappened_before_if_causal_trans` in `code/model/CalculusSM.v`.

where Π_1 is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle G, u : e' {\prec} e\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e \vdash \forall_{\preceq}\texttt{LID} \mathbin{@} e'}\;\texttt{STR}\forall_{\preceq} + \texttt{hyp} \qquad \Pi_2}{\langle G, u : e' {\prec} e\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e, k : \boldsymbol{\mathcal{K}}^{+}(t) \mathbin{@} e' \vdash {\prec}(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\;\texttt{cut} + \texttt{KID} + \texttt{thin}_{\texttt{h}} + \rightarrow_{\texttt{E}}}{\langle G, u : e' {\prec} e\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e, z : \boldsymbol{\mathcal{ND}}(d) \mathbin{@} e', i : t {\in} d \mathbin{@} e', c : \boldsymbol{\mathcal{C}} \mathbin{@} e', k : \boldsymbol{\mathcal{K}}^{+}(t) \mathbin{@} e' \vdash {\prec}(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\;\texttt{thin}_{\texttt{h}}}{\langle G, u : e' {\prec} e\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e, z : \boldsymbol{\mathcal{ND}}(d) \mathbin{@} e', i : t {\in} d \mathbin{@} e', c : \boldsymbol{\mathcal{C}} \mathbin{@} e' \vdash {\prec}(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\;\texttt{combo}}{\langle G\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e, z : (\exists_{\texttt{d}} \lambda d. {\prec}(\boldsymbol{\mathcal{ND}}(d) \wedge t {\in} d \wedge \boldsymbol{\mathcal{C}})) \mathbin{@} e \vdash {\prec}(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\;\exists_{\texttt{E}} + {\prec}_{\texttt{E}} + \wedge_{\texttt{E}}}{\langle G\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e, y : \boldsymbol{\mathcal{L}}(t) \mathbin{@} e, z : (\exists_{\texttt{d}} \lambda d. {\prec}(\boldsymbol{\mathcal{ND}}(d) \wedge t {\in} d \wedge \boldsymbol{\mathcal{C}})) \mathbin{@} e \vdash {\prec}(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\;\texttt{thin}_{\texttt{h}}$$

and where Π_2 is:

$$\dfrac{}{\langle G, u : e' {\prec} e\rangle\, H, x : \forall_{\prec}\texttt{LID} \mathbin{@} e, k : \boldsymbol{\mathcal{K}}^{+}(t) \mathbin{@} e', d : {\preceq}(\boldsymbol{\mathcal{OD}}(d)) \mathbin{@} e' \vdash {\prec}(\boldsymbol{\mathcal{OD}}(t)) \mathbin{@} e}\;\texttt{WEAK}{\prec} + \texttt{hyp}$$

and `combo` is actually a sequence of following rules: `cut`, `DITK`, $\texttt{thin}_{\texttt{h}}$ and `hyp`.

Appendix B

LoCK Tutorial

We briefly explain here how to use LoCK to prove lemmas. We provide several examples in the following files: `code/model/CalculusSM.v`, `code/model/CalculusSM2.v`, `code/model/CalculusSM_derived.v`, `code/model/CalculusSM_derived2.v`, `code/model/CalculusSM_derived3.v`, and `code/model/CalculusSM_derived3.v`. The names of the lemmas in those files either start with `PRIMITIVE` for primitive rules, or by `DERIVED` for derived rules. The example we use here is `DERIVED_RULE_unlocal_before_eq_hyp_true`, which we proved in `code/model/CalculusSM.v`, and which we discuss in Section 7.5:

```
Definition DERIVED_RULE_unlocal_before_eq_hyp
u x (eo : EventOrdering) e R H K a b :=
  MkRule1
    (fun e' ⇒ [⟨(u : e' ⊑ e), R⟩ H, (x : a @ e'), K ⊢ b])
    ((⟨R⟩ H, x : ⊑ a @ e), K ⊢ b).

Lemma DERIVED_RULE_unlocal_before_eq_hyp_true :
  ∀ u x (eo : EventOrdering) e R H K a b,
     rule_true (DERIVED_RULE_unlocal_before_eq_hyp u x e R H K a
b).
Proof.
  start_proving_derived st.
  LOCKelim x.

  { LOCKapply (PRIMITIVE_RULE_unlocal_before_hyp_true u).
    LOCKapply@ u PRIMITIVE_RULE_local_if_localle_true.
    inst_hyp e0 st. }

  { LOCKapply (DERIVED_RULE_add_localle_refl_true u e).
    inst_hyp e st. }
Qed.
```

First we start the proof with the tactic: `start_proving_derived st`, which allows us to focus on the conclusion of the rule, and moves the hypotheses of the rule to the hypotheses in Coq (those hypotheses are called *st*). We can then start applying rules.

Every time we apply a rule, we use the proof that the rule is true. For example, `LOCKapply` (PRIMITIVE_RULE_unlocal_before_hyp_true u), applies the PRIMITIVE_RULE_unlocal_before_hyp rule, which we have proved to be valid in the lemma PRIMITIVE_RULE_unlocal_before_hyp_true. Incidentally, this rule is the $\square_{\mathtt{E}}$ elimination rule presented in Figure 7.9. The `LOCKelim` tactic automatically tries to apply the appropriate (elimination) rule. Here because the hypothesis x is of the form $\sqsubseteq\tau$, which is defined as $\sqsubset\tau \vee \tau$ (see Section 7.2), `LOCKelim` automatically applies the *or* elimination rule. From this, we get two branches, one for each branch of the *or*, which is why we have two blocks below that tactic: the first one is the proof of the left branch, and the second one is the proof of the right branch.

We use a couple more useful tactics in this proof, which we describe next. To prove the left branch, we use the `LOCKapply@` *u rule* tactic, which is similar to `LOCKapply`, but in addition gets either the guards or the hypotheses (depending on whether the name u is a guard name or an hypothesis name) in the right shape whenever a rule mentions a guard or an hypothesis. For example DERIVED_RULE_add_localle_refl_true, is the validity proof of one of our weakening rule (see the rule called **weak** in Figure 7.9 in Section 7.4). The guards in the conclusion of that rule are of the form $G_1, y : e' \sqsubset e, G_2$. The tactic `LOCKapply@` helps turn the guards in the current sequent into that precise shape by pointing to the guard name y (u in our proof above).

Finally `inst_hyp` *e st*, instantiates the hypotheses of our rule, namely the function (`fun` *e'* $\Rightarrow$ $[\langle(u : e' \sqsubseteq e), R\rangle\ H, (x : a\ @\ e'), K \vdash b]$) with the event variable e, and call the instances *st*.

Let us now end this section with a summary of the tactics we provide as part of LoCK:

- `start_proving_derived` *st*: to start proving a derived rule.
- `start_proving_primitive` *st ct ht*: to start proving a primitive rule.
- `inst_hyp` *v st*: to instantiate the hypotheses of a rule with v, which must either be an event, or a node name, or a trusted piece of data, or a non-trusted piece of data, or an identifier.
- `LOCKapply`: to apply a rule.
- `LOCKapply@`: to apply a rule on a given guard or hypothesis.
- `LOCKintro`: an "introduction" tactic, which can be extended at will.

- `LOCKelim`: an “elimination” tactic, which can be extended at will.
- `LOCKauto`: an “auto” tactic, which can be extended at will, and which currently tries to apply a few simple rules, such as the “hypothesis” rule.
- `LOCKclear`: to clear an hypothesis or a guard.
- `simseqs` j: to get the sequents in the right shape after having applied a rule (one should not need to use this tactic, because it is done by `LOCKapply` and `LOCKapply@`).
- `causal_norm_with` u: to focus on a particular guard (one should not need to use this tactic, because it is done by `LOCKapply@`).
- `norm_with` x: to focus on a particular hypothesis (one should not need to use this tactic, because it is done by `LOCKapply@`).

Appendix C

OCaml Runtime Environments

We implemented two runtime environments to execute MoC distributed systems. One of them relies on SGX to execute trusted components, while the other simpler one runs trusted components as the other "normal" components. We discuss both environments below.

SGX-free runtime. Beside the runtime environment discussed in Section 8.2 and below, that uses Intel SGX, we developed an additional runtime environment (located in `code/MinBFT/runtime_wo_sgx`) that does not depend on any trusted environment for two reasons: it enables testing our framework on platforms that do not contain any trusted execution environment; and it can be very useful for debugging.

SGX-based runtime. As mentioned in Section 8.2, using framework presented in this thesis, one can extract OCaml code from distributed systems implemented using MoC, such that trusted components execute inside Intel SGX enclaves. We chose to rely on Graphene-SGX [TPV17] to do this because, to the best of our knowledge, one cannot directly run OCaml code inside SGX enclaves. Instead of using Graphene-SGX, one could run OCaml's runtime environment inside SGX enclaves, which would require creating OCALLs for all system calls made by OCaml's runtime environment that are not included in the libraries provided by SGX. Besides the fact that this solution could lead to security issues, it might be very slow.

Here, using a concrete example, i.e., a *createUI* call, we explain the interaction between a replica's main component and the Graphene-SGX enclave that runs this replica's USIG component. As mentioned above, because Graphene-SGX closes enclaves after each call, we implemented a loop around the USIG service to keep it running forever, as well as a TCP interface to access this loop. Also, because

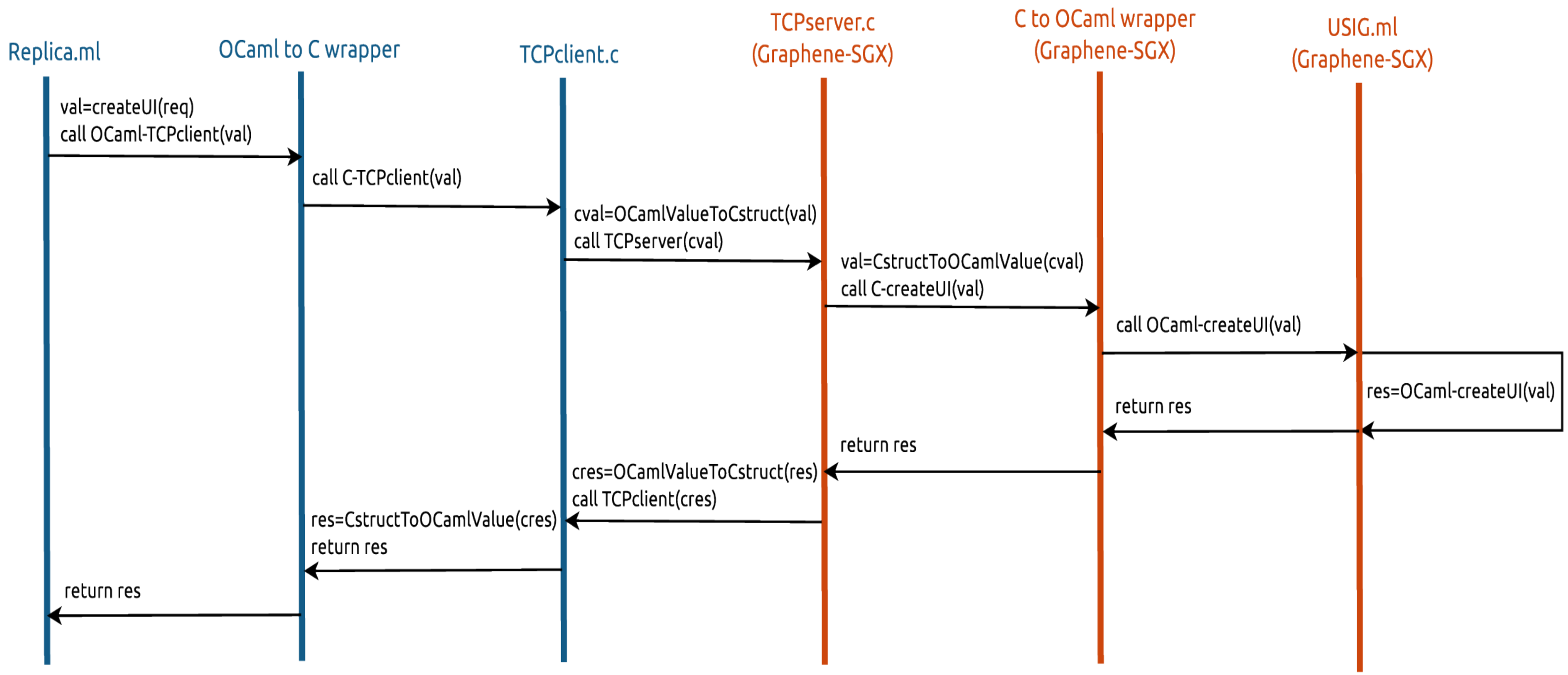


Table C.1: OCaml/SGX interaction

Graphene-SGX, to the best of our knowledge, provides only a C interface, we implemented this loop and this TCP interface in C. As shown in Fig. C.1, when the main component of a replica calls the *createUI* function of its USIG, this call is forwarded to the client of this TCP interface. Moreover, because we extract MoC code to OCaml, we had to implement an OCaml/C wrapper around our TCP interface implemented C. Next, the TCP client forwards the value it received through this call to *createUI* to the TCP server, which runs inside a Graphene-SGX enclave. To transfer this OCaml value across the TCP connection, we had to implement a custom serializer to convert that value to a C structure. Finally, when the TCP server running a Graphene-SGX enclave receives this C structure, it uses a custom deserializer to convert it back to an OCaml value, which the server uses to call the OCaml *createUI* function (again using a C/OCaml wrapper around the USIG code). Note that similar steps have to executed to deliver the value computed by the USIG, back to the main component.

Appendix D

Summary of Notation

To help readers relate this thesis with its implementation, we provide in Table D.1—D.5 a summary of the notation we use throughout this thesis. Table D.1 summarizes the ByLoE and HyLoE notation; Table D.2 summarizes the MoC notation; and Table D.3 summarizes the ByK notation; and Table D.4 summarizes the LoCK notation. In addition, Table D.5 provides pointers to the rules in our implementations.

HyLoE Notation	Meaning & File
Event	a set of events see Event field in the EventOrdering class (`code/model/EventOrdering.v`)
AuthData	a set of authenticated pieces of data see the `AuthenticatedData` record (`code/model/Crypto.v`)
Keys	a set of keys see class Keys (`code/model/Crypto.v`)
$\prec$	a causal ordering relation see happenedBefore field in the EventOrdering class (`code/model/EventOrdering.v`)
$\mathrm{loc}(e)$	the location where the event e happens see loc field in the EventOrdering class (`code/model/EventOrdering.v`)
$\mathrm{trigger}(e)$	explains why event e happened see trigger field in the EventOrdering class (`code/model/EventOrdering.v`)
$\mathtt{TImsg}(msg)$	an event happened at a correct node that followed the given protocol see constructor trigger_info_data in the `trigger_info` (`code/model/EventOrdering.v`)

$\mathtt{TItrust}(it)$	an event happened at a compromised node and the trusted component was called see constructor trigger_info_trusted in the `trigger_info` (`code/model/EventOrdering.v`)
$\mathtt{TIarbitrary}$	an event happened at a compromised node and the trusted component was not called see constructor trigger_info_arbitrary in the `trigger_info` (`code/model/EventOrdering.v`)
$\mathrm{pred}(e)$	local direct predecessor of e see direct_pred field in the EventOrdering class (`code/model/EventOrdering.v`)
$\mathrm{keys}(e)$	the keys available at e see keys field in the EventOrdering class (`code/model/EventOrdering.v`)
$\mathrm{nfo2auth}(nfo)$	a list of the authenticated pieces of data included in nfo see bind_op_list, get_contained_authenticated_data and trigger_op (`code/model/EventOrdering.v`)
$\mathrm{first?}(e) = \mathtt{true}$	$\mathrm{pred}(e) = \mathtt{None}$ see definition isFirst (`code/model/EventOrdering.v`)
$e_1 \subset e_2$	$\mathrm{pred}(e_2) = \mathtt{Some}(e_1)$ see direct_pred field in the EventOrdering class (`code/model/EventOrdering.v`)
$\mathrm{pred}^{=}(e)$	e' if $e' \subset e$, and e otherwise see definition local_pred (`code/model/EventOrdering.v`)
$e_1 \preceq e_2$	$e_1 \prec e_2 \vee e_1 = e_2$ see definition happenedBeforeLe (`code/model/EventOrdering.v`)
$e_1 \sqsubset e_2$	$e_1 \prec e_2 \wedge \mathrm{loc}(e_1) = \mathrm{loc}(e_2)$ see definition localHappenedBefore (`code/model/EventOrdering.v`)
$e_1 \sqsubseteq e_2$	$e_1 \preceq e_2 \wedge \mathrm{loc}(e_1) = \mathrm{loc}(e_2)$ see definition localHappenedBeforeLe (`code/model/EventOrdering.v`)

Table D.1: Summary of our ByLoE and HyLoE notation

MoC Notation	Meaning & File
$\mathcal{S}(cn)$	the type of the state of component cn see stateFun class (`code/model/ComponentSM.v`)
$\mathcal{I}(cn)$	the type of inputs of component cn see cio_I field in the ComponentIO record (`code/model/ComponentSM.v`)
$\mathcal{O}(cn)$	the type of output of component cn see cio_O field in the ComponentIO record (`code/model/ComponentSM.v`)
$\mathsf{Component}^{n}$	the collection of components at level n see definition n_proc (`code/model/ComponentSM.v`)

$\mathsf{M}^{n}(T)$	level n component monad of type T see definition M_n (`code/model/ComponentSM.v`)
$\mathsf{Upd}^{n}(cn)$	type of the update function of the component called cn see definition M_Update (`code/model/ComponentSM.v`)
$\mathsf{ret}(a)$	*return* operator of our component monad see definition ret (`code/model/ComponentSM.v`)
$m \gg= f$	*bind* operator of our component monad see definition bind (`code/model/ComponentSM.v`)
call	*call* operator of our component monad see definition call_proc (`code/model/ComponentSM.v`)
$ls@^{-}e$	local subsystem ls after it has executed the list of events locally preceding e, excluding e see definition M_run_ls_before_event (`code/model/ComponentSM.v`)
$ls@^{+}e$	local subsystem ls after it has executed the list of events locally preceding e, including e see definition M_run_ls_on_event (`code/model/ComponentSM.v`)
$ls\downharpoonleft_{cn}$	accesses the state of a component named cn of a local subsystem ls see definition state_of_component (`code/model/ComponentSM.v`)
$comp\downharpoonleft_{cn}$	returns the component $comp$ if its name is cn, otherwise it is undefined see definition on_state_of_component (`code/model/ComponentSM.v`)
$ls@^{-}e\downharpoonleft_{cn}$	returns the state of ls's component called cn before the event e see definition M_byz_state_ls_before_event_of_trusted (`code/model/ComponentSM.v`)
$ls@^{+}e\downharpoonleft_{cn}$	returns the state of ls's component called cn after the event e see definition M_byz_state_ls_on_event_of_trusted (`code/model/ComponentSM.v`)
$S@^{-}e\downharpoonleft_{cn}$	computes the state of a component cn of a system S before a given event e see definition M_byz_state_sys_before_event (`code/model/ComponentSM.v`)
$S@^{+}e\downharpoonleft_{cn}$	computes the state of a component cn of a system S after a given event e see definition M_byz_state_sys_on_event (`code/model/ComponentSM.v`)
$ls \rightsquigarrow e$	returns the outputs of ls's main component at e when all the events preceding e are non-Byzantine, and returns the outputs of the trusted component otherwise see definition M_byz_output_ls_on_event (`code/model/ComponentSM.v`)
$S \rightsquigarrow e$	$S(\mathrm{loc}(e)) \rightsquigarrow e$ see definition M_byz_output_sys_on_event (`code/model/ComponentSM.v`)
$d \in ls \rightsquigarrow e$	the d occurs within the outputs computed by $ls \rightsquigarrow e$ this is simply the membership relation as one can see in (`code/model/ComponentSM.v`)

RET(a)	*return* operator of our simple deep embedding (see Section 6.4) see constructor PROC_RET in the Proc (code/model/ComponentSM2.v)
BIND(p_1, p_2)	*bind* operator of our simple deep embedding (see Section 6.4) see constructor PROC_BIND in the Proc (code/model/ComponentSM2.v)
CALL(cn, i)	*call* operator of our simple deep embedding (see Section 6.4) see constructor PROC_CALL in the Proc (code/model/ComponentSM2.v)

Table D.2: Summary of our MoC notation

ByK Notation	Meaning & File
byk_data	the type of "raw" data that nodes have knowledge of see lak_data in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_info	the type of information that might be shared by different pieces of data see lak_info in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_mem	the type of objects used to store one's knowledge see lak_memory in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_data2info	extracts the information contained in some piece of data see lak_data2info in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_knows	what it means to know some piece of data see lak_knows in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_sys	the system that one wants to reason about see lak_system in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_data2owner	extracts the "owner" of some piece of data see lak_data2owner in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_data2msg	converts a piece of data to a message see dis_data2msg in Disseminate class (code/model/Disseminate.v)
byk_data2auth	extracts some piece of authenticated data from some piece of raw data see lak_data2auth in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_data2auth_list	extracts all pieces of authenticated data from some piece of raw data see lak_data2auth_list in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_verify	verifies the correctness of authenticity of a piece of data see lak_verify in LearnAndKnows class (code/model/LearnAndKnows.v)
byk_max_sign	the total number of processes that can sign messages in the system see dis_max_sign in AuthKnowledge class (code/model/Disseminate.v)

byk_ext_info	creates a piece of data from some piece of information and a signature see dis_extend_info in AuthKnowledge class (`code/model/Disseminate.v`)
byk_ext_data	creates a piece of data from a piece of data and a signature see dis_extend_data in AuthKnowledge class (`code/model/Disseminate.v`)
byk_data2data	allows casting a pieces of knowledge byk_data into a pieces of data see dis_data2data in AuthKnowledge class (`code/model/Disseminate.v`)
byk_data2sign	extracts a signature from a piece of data see dis_data2sign in AuthKnowledge class (`code/model/Disseminate.v`)
byk_data2can	converts a piece of data into its canonical form, which is a list of pairs of (1) a piece of data/information and (2) a signature see dis_data2can in AuthKnowledge class (`code/model/Disseminate.v`)

Table D.3: Summary of our ByK notation

LoCK Notation	Meaning
Data	a set of pieces of data see kc_data field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
Trust	a set of trusted pieces of data see kc_trust field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
Identifier	a set of data identifiers see kc_id field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
trustHasId	relates trusted pieces of data and identifiers see kc_trust_has_id field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
sys	the distributed system one wants to reason about see kc_sys field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
mem	the name of the component holding the knowledge see kc_mem field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
trust	the name of the trusted component see IOTrusted class (`code/model/EventOrdering.v`) and see trustedStateFun class (`code/model/ComponentSM.v`)
owner	identifies the node that generated a given piece of data see kc_data_owner field in the KnowledgeComponents class (`code/model/CalculusSM.v`)

verify(e, $auth$)	returns `true` if the authenticated piece of data $auth$ can indeed be authenticated at e, and `false` otherwise see definition kc_verify (`code/model/CalculusSM.v`)
genFor	relates trusted pieces of data and non-trusted pieces of data see kc_generated_for field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
know	expresses what it means to hold some information see kc_knows field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
trusted2id	returns the trusted identifier maintained by the trusted component see kc_trust_has_id field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
initId	initial value of the identifier maintained by the trusted component see kc_init_id field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
auth2data	extracts the pieces of data contained within an authenticated piece of data see kc_auth2data field in the KnowledgeComponents class (`code/model/CalculusSM.v`)
$\top$, $\bot$, $\wedge$, $\vee$, $\rightarrow$, $\exists$, $\forall$	standard first-order logic operators see constructors: KE_TRUE, KE_FALSE, KE_AND, KE_OR, KE_IMPLIES, KE_EX, KE_ALL, respectively (`code/model/CalculusSM.v`)
$\subset$, $\prec$, $\sqsubset$	HyLoE-specific operators to state properties relating different points in space/time see constructors: KE_RIGHT_BEFORE, KE_HAPPENED_BEFORE, KE_LOCAL_BEFORE, respectively (`code/model/CalculusSM.v`)
$\odot$	the HyLoE-specific operator to talk about initial event see constructor KE_FIRST (`code/model/CalculusSM.v`)
@	the HyLoE-specific operators to relate space/time coordinates see constructor KE_AT (`code/model/CalculusSM.v`)
$\mathcal{K}^+$	knows see constructor KE_KNOWS (`code/model/CalculusSM.v`)
$\mathcal{L}$	learns see constructor KE_LEARNS (`code/model/CalculusSM.v`)
$\mathcal{O}$	owns see constructor KE_HAS_OWNER (`code/model/CalculusSM.v`)
$\mathcal{D}$	disseminate see constructor KE_DISS (`code/model/CalculusSM.v`)
$\mathcal{I}^+$	knows identifier see constructor KE_ID_AFTER (`code/model/CalculusSM.v`)
$\mathcal{HI}$	has identifier see constructor KE_HAS_ID (`code/model/CalculusSM.v`)
$\mathcal{G}$	generated see constructor KE_GEN_FOR (`code/model/CalculusSM.v`)
$\mathcal{O}(d)$	"we" own the data d see definition KE_OWNS (`code/model/CalculusSM.v`)

$\mathcal{OD}(d)$	“we” disseminated the data d see definition KE_DISS_OWN (`code/model/CalculusSM.v`)
`LID`	if one learns some trusted piece data, it must have been disseminated by the corresponding trusted component see definition ASSUMPTION_learns_if_gen (`code/model/CalculusSM.v`)
`KLD`	if we know some trusted information, then we either knew it before, or we just learned it, or we just disseminated it see definition ASSUMPTION_learns_or_gen (`code/model/CalculusSM.v`)
`Mon`	the identifiers maintained by trusted components monotonically increase see definition ASSUMPTION_monotonicity (`code/model/CalculusSM.v`)
`New`	an identifier generated by a trusted component i must be between the one it recorded before and the one it recorded after it generated i see definition ASSUMPTION_generates_new (`code/model/CalculusSM.v`)
`Uniq`	a trusted pieces of data disseminated by a trusted component at a given point in space/time must be unique see definition ASSUMPTION_disseminates_unique (`code/model/CalculusSM.v`)
$\exists_{\mathtt{i}} f$, $\exists_{\mathtt{d}} f$, $\exists_{\mathtt{t}} f$, $\exists_{\mathtt{n}} f$	$\exists\langle\mathtt{KTi}, f\rangle$, $\exists\langle\mathtt{KTd}, f\rangle$, $\exists\langle\mathtt{KTt}, f\rangle$, and $\exists\langle\mathtt{KTn}, f\rangle$, respectively i.e., existential quantifier for our different kinds of values see KE_EX_ID, KE_EX_DATA, KE_EX_TRUST, and KE_EX_NODE, respectively (`code/model/CalculusSM.v`)
$\forall_{\mathtt{i}} f$, $\forall_{\mathtt{d}} f$, $\forall_{\mathtt{t}} f$, $\forall_{\mathtt{n}} f$	$\forall\langle\mathtt{KTi}, f\rangle$, $\forall\langle\mathtt{KTd}, f\rangle$, $\forall\langle\mathtt{KTt}, f\rangle$, and $\forall\langle\mathtt{KTn}, f\rangle$, respectively i.e., universal quantifier for our different kinds of values see KE_ALL_ID, KE_ALL_DATA, KE_ALL_TRUST, and KE_ALL_NODE, respectively (`code/model/CalculusSM.v`)
$\exists_{\mathtt{i}}\lambda i_1,\dots,i_n.\tau$	$\exists_{\mathtt{i}}\lambda i.\dots\exists_{\mathtt{i}}\lambda i_n.\tau$, i.e., universal multi-quantifier for node (and similarly for the other values) see KE_EX_IDs (`code/model/CalculusSM.v`)
$\forall_{\mathtt{i}}\lambda i_1,\dots,i_n.\tau$	$\forall_{\mathtt{i}}\lambda i.\dots\forall_{\mathtt{i}}\lambda i_n.\tau$, i.e., universal multi-quantifier for node (and similarly for the other values) see KE_ALL_IDs (`code/model/CalculusSM.v`)
$\neg\tau$	negation see KE_NOT (`code/model/CalculusSM.v`)
$\preceq\tau$	happened before or equal, i.e., $\prec\tau \vee \tau$ see KE_HAPPENED_BEFORE_EQ (`code/model/CalculusSM.v`)
$\sqsubseteq\tau$	happened locally before or equal, i.e., $\sqsubset\tau \vee \tau$ see KE_LOCAL_BEFORE_EQ (`code/model/CalculusSM.v`)
$\subseteq\tau$	direct predecessor or equal, i.e., $\subset\tau \vee (\tau \wedge \odot)$ see KE_RIGHT_BEFORE_EQ (`code/model/CalculusSM.v`)
$i_1 \leq i_2$	identifier is less than or equal to, i.e., $i_1 < i_2 \vee i_1 = i_2$ see KE_ID_LE (`code/model/CalculusSM.v`)

$[\![\tau]\!]_e$	interpretation of LoCK expressions see interpret (`code/model/CalculusSM.v`)
$\langle G \rangle\ H \vdash \sigma$	syntax of sequents see MkSeq (`code/model/CalculusSM.v`)
H_1, H_2	append operation on sequent hypotheses this is simply the append operation on lists (`code/model/CalculusSM.v`)

Table D.4: Summary of our LoCK notation

Thesis name	Implementation name
$\square_{\mathtt{E}}$ for $(\prec)$	see `PRIMITIVE_RULE_unhappened_before_hyp` in `code/model/CalculusSM.v`
$\square_{\mathtt{I}}$ for $(\prec)$	see `PRIMITIVE_RULE_unhappened_before_if_causal` in `code/model/CalculusSM.v`
$\square_{\mathtt{E}}$ for $(\sqsubset)$	see `DERIVED_RULE_unlocal_before_hyp` in `code/model/CalculusSM.v`[1]
$\square_{\mathtt{I}}$ for $(\sqsubset)$	see `DERIVED_RULE_unlocal_before_if_causal` in `code/model/CalculusSM.v`
$\square_{\mathtt{It}}$	see `PRIMITIVE_RULE_unhappened_before_if_causal_trans_eq` in `code/model/CalculusSM.v`
`if`$\neg\odot$	see `PRIMITIVE_RULE_introduce_direct_pred` in `code/model/CalculusSM.v`
`if`$\odot$	see `PRIMITIVE_RULE_introduce_direct_pred_eq` in `code/model/CalculusSM.v`
`weak` for $(\prec, \preceq)$	see `PRIMITIVE_RULE_causal_if_causalle_true` in `code/model/CalculusSM.v`
`weak` for $(\sqsubset, \sqsubseteq)$	see `PRIMITIVE_RULE_local_if_localle` in `code/model/CalculusSM.v`
`weak` for $(\sqsubset, \prec)$	see `PRIMITIVE_RULE_local_if_causal` in `code/model/CalculusSM.v`
`weak` for $(\sqsubseteq, \preceq)$	see `PRIMITIVE_RULE_localle_if_causalle` in `code/model/CalculusSM.v`
`weak` for $(\subset, \sqsubset)$	see `PRIMITIVE_RULE_direct_pred_if_local_pred` in `code/model/CalculusSM.v`
`weak` for $(\equiv, \sqsubseteq)$	see `PRIMITIVE_RULE_localle_if_eq` in `code/model/CalculusSM.v`
`sub`$_{\mathtt{H}}$	see `PRIMITIVE_RULE_subst_causal_eq_hyp` in `code/model/CalculusSM.v`
`sub`$_{\mathtt{C}}$	see `PRIMITIVE_RULE_subst_causal_eq_concl` in `code/model/CalculusSM.v`
$\equiv_{\mathtt{refl}}$	see `PRIMITIVE_RULE_add_eq_refl` in `code/model/CalculusSM.v`
$\neg\odot$	see `PRIMITIVE_RULE_not_first` in `code/model/CalculusSM.v`
$\odot_{\mathtt{dec}}$	see `PRIMITIVE_RULE_first_dec` in `code/model/CalculusSM.v`

---

[1] This rule, as well as $\square_{\mathtt{I}}$, are not primitive anymore because $\sqsubset$ is not a primitive operator of LoCK anymore. However, we still present it as such for simplicity (see `KE_LOCAL_BEFORE` in `code/model/CalculusSM.v`).

`ind`	see `PRIMITIVE_RULE_pred_induction` in `code/model/CalculusSM.v`
`tr`i	see `PRIMITIVE_RULE_tri_if_same_loc` in `code/model/CalculusSM.v`
`sym`	see `PRIMITIVE_RULE_id_eq_sym` in `code/model/CalculusSM.v`
`trans` for $(=,<,<)$	see `PRIMITIVE_RULE_id_lt_trans_eq_lt` in `code/model/CalculusSM.v`
`trans` for $(<,=,<)$	see `PRIMITIVE_RULE_id_lt_trans_lt_eq` in `code/model/CalculusSM.v`
`trans` for $(<,<,<)$	see `PRIMITIVE_RULE_id_lt_trans_lt_lt` in `code/model/CalculusSM.v`
`trans` for $(=,=,=)$	see `PRIMITIVE_RULE_id_eq_trans_true` in `code/model/CalculusSM.v`
$\mathtt{K}_{\mathtt{dec}}$	see `PRIMITIVE_RULE_decidable_knows` in `code/model/CalculusSM.v`
`irrefl`	see `PRIMITIVE_RULE_id_lt_elim` in `code/model/CalculusSM.v`
`lowner`	see `PRIMITIVE_RULE_has_owner_implies_eq` in `code/model/CalculusSM.v`
`ldata`	see `PRIMITIVE_RULE_collision_resistant` in `code/model/CalculusSM.v`
`lid`	see `PRIMITIVE_RULE_ids_after_imply_eq_ids` in `code/model/CalculusSM.v`
$\top_{\mathtt{I}}$	see `PRIMITIVE_RULE_true` in `code/model/CalculusSM.v`
$\bot_{\mathtt{E}}$	see `PRIMITIVE_RULE_false_elim` in `code/model/CalculusSM.v`
$\rightarrow_{\mathtt{E}}$	see `PRIMITIVE_RULE_implies_elim` in `code/model/CalculusSM.v`
$\rightarrow_{\mathtt{I}}$	see `PRIMITIVE_RULE_implies_intro` in `code/model/CalculusSM.v`
$\vee_{\mathtt{E}}$	see `PRIMITIVE_RULE_or_elim` in `code/model/CalculusSM.v`
$\vee_{\mathtt{Il}}$	see `PRIMITIVE_RULE_or_intro_left` in `code/model/CalculusSM.v`
$\vee_{\mathtt{Ir}}$	see `PRIMITIVE_RULE_or_intro_right` in `code/model/CalculusSM.v`
$\wedge_{\mathtt{E}}$	see `PRIMITIVE_RULE_and_elim` in `code/model/CalculusSM.v`
$\wedge_{\mathtt{I}}$	see `PRIMITIVE_RULE_and_intro` in `code/model/CalculusSM.v`
$\exists_{\mathtt{E}}$ (for `KTi`)	see `PRIMITIVE_RULE_exists_id_elim` in `code/model/CalculusSM.v`
$\exists_{\mathtt{E}}$ (for `KTd`)	see `PRIMITIVE_RULE_exists_data_elim` in `code/model/CalculusSM.v`
$\exists_{\mathtt{E}}$ (for `KTt`)	see `PRIMITIVE_RULE_exists_trust_elim` in `code/model/CalculusSM.v`
$\exists_{\mathtt{E}}$ (for `KTn`)	see `PRIMITIVE_RULE_exists_node_elim` in `code/model/CalculusSM.v`
$\exists_{\mathtt{I}}$ (for `KTi`)	see `PRIMITIVE_RULE_id_count_intro` in `code/model/CalculusSM.v`
$\exists_{\mathtt{I}}$ (for `KTd`)	see `PRIMITIVE_RULE_exists_data_intro` in `code/model/CalculusSM.v`
$\exists_{\mathtt{I}}$ (for `KTt`)	see `PRIMITIVE_RULE_exists_trust_intro` in `code/model/CalculusSM.v`
$\exists_{\mathtt{I}}$ (for `KTn`)	see `PRIMITIVE_RULE_exists_node_intro` in `code/model/CalculusSM.v`
$\forall_{\mathtt{E}}$ (for `KTi`)	see `PRIMITIVE_RULE_all_id_elim` in `code/model/CalculusSM.v`
$\forall_{\mathtt{E}}$ (for `KTd`)	see `PRIMITIVE_RULE_all_data_elim` in `code/model/CalculusSM.v`
$\forall_{\mathtt{E}}$ (for `KTt`)	see `PRIMITIVE_RULE_all_trust_elim` in `code/model/CalculusSM.v`

$\forall_{\mathtt{E}}$ (for KTn)	see PRIMITIVE_RULE_all_node_elim in code/model/CalculusSM.v
$\forall_{\mathtt{I}}$ (for KTi)	see PRIMITIVE_RULE_all_id_intro in code/model/CalculusSM.v
$\forall_{\mathtt{I}}$ (for KTd)	see PRIMITIVE_RULE_all_data_intro in code/model/CalculusSM.v
$\forall_{\mathtt{I}}$ (for KTt)	see PRIMITIVE_RULE_all_trust_intro in code/model/CalculusSM.v
$\forall_{\mathtt{I}}$ (for KTn)	see PRIMITIVE_RULE_all_node_intro in code/model/CalculusSM.v
$\subset_{\mathtt{E}}$	see PRIMITIVE_RULE_unright_before_hyp in code/model/CalculusSM.v
$\square_{\mathtt{E}}$ for $\subset$	see PRIMITIVE_RULE_unright_before_hyp_if_causal in code/model/CalculusSM.v
$\square_{\mathtt{I}}$ for $\subset$	see PRIMITIVE_RULE_unright_before_if_causal in code/model/CalculusSM.v
$\mathtt{STR}_{\sqsubseteq}$	see PRIMITIVE_RULE_split_local_before_eq2 in code/model/CalculusSM.v
$\mathtt{STR}_{\preceq}$	see PRIMITIVE_RULE_split_happened_before_eq2 in code/model/CalculusSM.v
$\mathtt{STRl}_{\preceq}$	see PRIMITIVE_RULE_at_implies_localle in code/model/CalculusSM.v
$\mathtt{STRl}_{\prec}$	see PRIMITIVE_RULE_at_implies_local in code/model/CalculusSM.v
$\mathtt{split}_{\sqsubset}$	see PRIMITIVE_RULE_split_local_before in code/model/CalculusSM.v
$\equiv_{\mathtt{sym}}$	see PRIMITIVE_RULE_causal_eq_sym in code/model/CalculusSM.v
$\equiv_{\mathrm{pred}=}$	see PRIMITIVE_RULE_weaken_direct_pred_to_local_pred in code/model/CalculusSM.v
$@_{\mathtt{loc}}$	see PRIMITIVE_RULE_at_change_localle in code/model/CalculusSM.v
loc	see PRIMITIVE_RULE_at_implies_same_node in code/model/CalculusSM.v
change for $i_1 = i_2$	see PRIMITIVE_RULE_id_eq_change_event in code/model/CalculusSM.v
change for $d_1 = d_2$	see PRIMITIVE_RULE_data_eq_change_event in code/model/CalculusSM2.v
change for $t_1 = t_2$	see PRIMITIVE_RULE_trust_eq_change_event in code/model/CalculusSM.v
change for $\boldsymbol{a}_1 = \boldsymbol{a}_2$	see PRIMITIVE_RULE_node_eq_change_event in code/model/CalculusSM2.v
change for $i_1 < i_2$	see PRIMITIVE_RULE_id_lt_change_event in code/model/CalculusSM.v
change for $\mathcal{HI}(t, i)$	see PRIMITIVE_RULE_has_id_change_event in code/model/CalculusSM.v
change for $\mathcal{O}(d, \boldsymbol{a})$	see PRIMITIVE_RULE_has_owner_change_event in code/model/CalculusSM.v
change for $\mathcal{G}(d, t)$	see PRIMITIVE_RULE_gen_for_change_event in code/model/CalculusSM.v
valSub for ($\mathcal{HI}(t, i)$)	see PRIMITIVE_RULE_trust_has_id_subst in code/model/CalculusSM.v

`valSub` for $(\mathcal{O}(d, \boldsymbol{a}))$	see `PRIMITIVE_RULE_subst_node_in_has_owner` in `code/model/CalculusSM.v`

Table D.5: Pointers to our rules

Bibliography

[17a] *BitcoinUnlimited*. 2017. URL: `https://www.coindesk.com/code-bug-exploit-bitcoin-unlimited-nodes`.

[17b] *CloudBleed*. 2017. URL: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1139`.

[18] *Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm*. 2018. URL: `http://lamport.azurewebsites.net/tla/byzpaxos.html`.

[19a] *Async*. 2019. URL: `https://janestreet.github.io/guide-async.html`.

[19b] *Boing737*. 2019. URL: `https://www.nytimes.com/interactive/2019/business/boeing-737-crashes.html`.

[19c] *CBC Casper FAQ*. 2019. URL: `https://github.com/ethereum/cbc-casper/wiki/FAQ`.

[19d] *The Coq Proof Assistant*. 2019. URL: `http://coq.inria.fr/`.

[19e] *Hyperledger*. 2019. URL: `https://github.com/hyperledger-labs`.

[19f] *nocrypto*. 2019. URL: `https://github.com/mirleft/ocaml-nocrypto`.

[19g] *Interfacing C with OCaml*. 2019. URL: `https://caml.inria.fr/pub/docs/manual-ocaml/intfc.html`.

[19h] *Secure Blue*. 2019. URL: `https://researcher.watson.ibm.com/researcher/view_page.php?id=6904`.

[19i] *SGX*. 2019. URL: `https://software.intel.com/en-us/sgx`.

[19j] *(Almost) Everything you wanted to know about NEO*. 2019. URL: `https://steemit.com/cryptocurrency/@basiccrypto/almost-everything-you-wanted-to-know-about-neo-part-1-of-2`.

[19k] *Tendermint*. 2019. URL: `https://tendermint.com/`.

[19l] *TPM*. 2019. URL: `https://trustedcomputinggroup.org/work-groups/trusted-platform-module/`.

[19m] *ARM TrustZone.* 2019. URL: `https://www.arm.com/products/security-on-arm/trustzone`.

[Aba+18] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. “When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise”. In: *CCS*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1351–1368. DOI: `10.1145/3243734.3243745`. URL: `http://doi.acm.org/10.1145/3243734.3243745`.

[Abr+10] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. “Rodin: an open toolset for modelling and reasoning in Event-B”. In: *STTT* 12.6 (2010), pp. 447–466. DOI: `10.1007/s10009-010-0145-y`. URL: `http://dx.doi.org/10.1007/s10009-010-0145-y`.

[Abr+16] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. “Solidus: An Incentive-compatible Cryptocurrency Based on Permissionless Byzantine Consensus”. In: *CoRR* abs/1612.02916 (2016). arXiv: `1612.02916`. URL: `http://arxiv.org/abs/1612.02916`.

[Abr+18] Ittai Abraham, Dahli Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. “Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus”. 2018.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010. URL: `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569`.

[AK15] Abhishek Anand and Ross A. Knepper. “ROSCoq: Robots Powered by Constructive Reals”. In: *ITP-6.* Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. LNCS. Springer, 2015, pp. 34–50. DOI: `10.1007/978-3-319-22102-1_3`. URL: `http://dx.doi.org/10.1007/978-3-319-22102-1_3`.

[Alu+05] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. “Synthesis of interface specifications for Java classes”. In: *POPL 2005.* Ed. by Jens Palsberg and Martín Abadi. ACM, 2005, pp. 98–109. DOI: `10.1145/1040305.1040314`. URL: `http://doi.acm.org/10.1145/1040305.1040314`.

[Ami+11] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. “Prime: Byzantine Replication under Attack”. In: *IEEE Trans. Dependable Sec. Comput.* 8.4 (2011), pp. 564–577. DOI: `10.1109/TDSC.2010.70`. URL: `https://doi.org/10.1109/TDSC.2010.70`.

[AMS14] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. “Analysis of Self-$\star$ and P2P Systems Using Refinement”. In: *ABZ 2014*. Ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. LNCS. Springer, 2014, pp. 117–123. DOI: `10.1007/978-3-662-43652-3_9`. URL: `http://dx.doi.org/10.1007/978-3-662-43652-3_9`.

[Avg14] Athanasios (Thanassis) Avgerinos. “Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs”. PhD thesis. Carnegie Mellon University, 2014.

[ÅW17] Thomas Ågotnes and Yì N. Wáng. “Resolving distributed knowledge”. In: *Artif. Intell.* 252 (2017), pp. 1–21. DOI: `10.1016/j.artint.2017.07.002`. URL: `https://doi.org/10.1016/j.artint.2017.07.002`.

[Bai+14] Guangdong Bai, Jianan Hao, Jianliang Wu, Yang Liu, Zhenkai Liang, and Andrew P. Martin. “TrustFound: Towards a Formal Foundation for Model Checking Trusted Computing Platforms”. In: *FM 2014*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. LNCS. Springer, 2014, pp. 110–126. DOI: `10.1007/978-3-319-06410-9\_8`. URL: `https://doi.org/10.1007/978-3-319-06410-9%5C_8`.

[Bar+92] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. “Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement”. In: *Inf. Comput.* 97.2 (1992), pp. 205–233. DOI: `10.1016/0890-5401(92)90035-E`. URL: `https://doi.org/10.1016/0890-5401(92)90035-E`.

[Bar10] Bruno Barras. “Sets in Coq, Coq in Sets”. In: *Journal of Formalized Reasoning* 3.1 (2010), pp. 29–48.

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* `http://www.labri.fr/perso/casteran/CoqArt`. SpringerVerlag, 2004.

[BCR12] Mark Bickford, Robert L. Constable, and Vincent Rahli. “Logic of Events, a framework to reason about distributed systems”. In: *Languages for Distributed Algorithms Workshop*. 2012. URL: `http://www.nuprl.org/documents/Bickford/LOE-LADA2012.html`.

[BDH07] William J. Bolosky, John R. Douceur, and Jon Howell. “The Farsite project: a retrospective”. In: *Operating Systems Review* 41.2 (2007), pp. 17–26. DOI: `10.1145/1243418.1243422`. URL: `http://doi.acm.org/10.1145/1243418.1243422`.

[BDK15] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. “Consensus-Oriented Parallelization: How to Earn Your First Million”. In: *16th Annual Middleware Conference*. Ed. by Rodger Lea, Sathish Gopalakrishnan, Eli Tilevich, Amy L. Murphy, and Michael Blackstock. ACM, 2015, pp. 173–184. DOI: `10.1145/2814576.2814800`. URL: `http://doi.acm.org/10.1145/2814576.2814800`.

[BDK17] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. “Hybrids on Steroids: SGX-Based High Performance BFT”. In: *EUROSYS 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 222–237. DOI: `10.1145/3064176.3064213`. URL: `http://doi.acm.org/10.1145/3064176.3064213`.

[Ben11] Ido Ben-Zvi. “Causality, Knowledge and Coordinaltion in Distributed Systems”. PhD thesis. Technion – Computer Science Department, Sept. 2011.

[Ben83] Michael Ben-Or. “Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract)”. In: *PODC 1983*. Ed. by Robert L. Probert, Nancy A. Lynch, and Nicola Santoro. ACM, 1983, pp. 27–30. DOI: `10.1145/800221.806707`. URL: `https://doi.org/10.1145/800221.806707`.

[Ber+19] Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. “Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics”. 2019.

[Ber06] Yves Bertot. “Coq in a Hurry”. In: *CoRR* abs/cs/0603118 (2006). arXiv: `cs/0603118`. URL: `http://arxiv.org/abs/cs/0603118`.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. “Towards Optimal Distributed Consensus (Extended Abstract)”. In: *FOCS 1989*. IEEE Computer Society, 1989, pp. 410–415. DOI: `10.1109/SFCS.1989.63511`. URL: `https://doi.org/10.1109/SFCS.1989.63511`.

[Bic+04] Mark Bickford, Robert L. Constable, Joseph Y. Halpern, and Sabina Petride. “Knowledge-Based Synthesis of Distributed Systems Using Event Structures”. In: *LPAR 2004*. Ed. by Franz Baader and Andrei Voronkov. Vol. 3452. LNCS. Springer, 2004, pp. 449–465. DOI: `10.1007/978-3-540-32275-7_30`. URL: `https://doi.org/10.1007/978-3-540-32275-7_30`.

[Bic09] Mark Bickford. “Component Specification Using Event Classes”. In: *CBSE 2009*. Ed. by Grace A. Lewis, Iman Poernomo, and Christine Hofmeister. Vol. 5582. LNCS. Springer, 2009, pp. 140–155.

[Bie+07] Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. “Tolerating corrupted communication”. In: *PODC 2007*. Ed. by Indranil Gupta and Roger Wattenhofer. ACM, 2007, pp. 244–253. DOI: `10.1145/1281100.1281136`. URL: `http://doi.acm.org/10.1145/1281100.1281136`.

[BJX17] Sergiu Bursuc, Christian Johansen, and Shiwei Xu. “Automated Verification of Dynamic Root of Trust Protocols”. In: *POST 2017*. Ed. by Matteo Maffei and Mark Ryan. Vol. 10204. LNCS. Springer, 2017, pp. 95–116. DOI: `10.1007/978-3-662-54455-6\_5`. URL: `https://doi.org/10.1007/978-3-662-54455-6%5C_5`.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. “A decade of software model checking with SLAM”. In: *Commun. ACM* 54.7 (2011), pp. 68–76. DOI: `10.1145/1965724.1965743`. URL: `http://doi.acm.org/10.1145/1965724.1965743`.

[BM10] Ido Ben-Zvi and Yoram Moses. “Beyond Lamport’s Happened-Before: On the Role of Time Bounds in Synchronous Systems”. In: *DISC 2010*. Ed. by Nancy A. Lynch and Alexander A. Shvartsman. Vol. 6343. LNCS. Springer, 2010, pp. 421–436. DOI: `10.1007/978-3-642-15763-9\_42`. URL: `https://doi.org/10.1007/978-3-642-15763-9%5C_42`.

[BM14] Ido Ben-Zvi and Yoram Moses. “Beyond Lamport’s Happened-before: On Time Bounds and the Ordering of Events in Distributed Systems”. In: *J. ACM* 61.2 (2014), 13:1–13:26. DOI: `10.1145/2542181`. URL: `https://doi.org/10.1145/2542181`.

[BPS98] Bettina Buth, Jan Peleska, and Hui Shi. “Combining Methods for the Livelock Analysis of a Fault-Tolerant System”. In: *AMAST 1998*. Ed. by Armando Martin Haeberer. Vol. 1548. Lecture Notes in Computer Science. Springer, 1998, pp. 124–139. DOI: `10.1007/3-540-49253-4\_11`. URL: `https://doi.org/10.1007/3-540-49253-4%5C_11`.

[Bry11] Jeremy W. Bryans. “Developing a Consensus Algorithm Using Stepwise Refinement”. In: *ICFEM 2011*. Ed. by Shengchao Qin and Zongyan Qiu. Vol. 6991. LNCS. Springer, 2011, pp. 553–568. DOI: `10.1007/978-3-642-24559-6_37`. URL: `http://dx.doi.org/10.1007/978-3-642-24559-6_37`.

[BSA14] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. “State Machine Replication for the Masses with BFT-SMART”. In: *DSN 2014*. IEEE, 2014, pp. 355–362. DOI: `10.1109/DSN.2014.43`. URL: `http://dx.doi.org/10.1109/DSN.2014.43`.

[BSW11] Martin Biely, Ulrich Schmid, and Bettina Weiss. “Synchronous consensus under hybrid process and link failures”. In: *Theor. Comput. Sci.* 412.40 (2011), pp. 5602–5630. DOI: `10.1016/j.tcs.2010.09.032`. URL: `https://doi.org/10.1016/j.tcs.2010.09.032`.

[BTZ17] Aaron Bembenek, Lily Tsai, and Ezra Zigmond. “Better Trust Zone : Verifying Security of Enclave-Aware Calculi”. In: 2017.

[But+97] Bettina Buth, Michel Kouvaras, Jan Peleska, and Hui Shi. “Deadlock Analysis for a Fault-Tolerant System”. In: *AMAST 1997*. Ed. by Michael Johnson. Vol. 1349. Lecture Notes in Computer Science. Springer, 1997, pp. 60–74. DOI: `10.1007/BFb0000463`. URL: `https://doi.org/10.1007/BFb0000463`.

[Cas01] Miguel Castro. “Practical Byzantine Fault Tolerance”. Also as Technical Report MIT-LCS-TR-817. Ph.D. MIT, 2001.

[CCM09] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. “A Reduction Theorem for the Verification of Round-Based Distributed Algorithms”. In: *RP 2009*. Ed. by Olivier Bournez and Igor Potapov. Vol. 5797. LNCS. Springer, 2009, pp. 93–106. DOI: `10.1007/978-3-642-04420-5_10`. URL: `https://doi.org/10.1007/978-3-642-04420-5_10`.

[CDM11] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. “Formal Verification of Consensus Algorithms Tolerating Malicious Faults”. In: *SSS 2011*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Vol. 6976. LNCS. Springer, 2011, pp. 120–134. DOI: `10.1007/978-3-642-24550-3_11`. URL: `https://doi.org/10.1007/978-3-642-24550-3_11`.

[CGM14] Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. “Unbeatable Consensus”. In: *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*. Ed. by Fabian Kuhn. Vol. 8784. LNCS. Springer, 2014, pp. 91–106. DOI: `10.1007/978-3-662-45174-8\_7`. URL: `https://doi.org/10.1007/978-3-662-45174-8%5C_7`.

[CGM16] Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. "Unbeatable Set Consensus via Topological and Combinatorial Reasoning". In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*. Ed. by George Giakkoupis. ACM, 2016, pp. 107–116. DOI: `10.1145/2933057.2933120`. URL: `https://doi.org/10.1145/2933057.2933120`.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. DOI: `10.1007/978-3-642-15260-3`. URL: `http://dx.doi.org/10.1007/978-3-642-15260-3`.

[Cha+00] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. "Tight bounds for k-set agreement". In: *J. ACM* 47.5 (2000), pp. 912–943. DOI: `10.1145/355483.355489`. URL: `https://doi.org/10.1145/355483.355489`.

[Cha+10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. "Verifying Safety Properties with the TLA+ Proof System". In: *IJCAR 2010*. Ed. by Jürgen Giesl and Reiner Hähnle. Vol. 6173. LNCS. Springer, 2010, pp. 142–148. DOI: `10.1007/978-3-642-14203-1_12`. URL: `https://doi.org/10.1007/978-3-642-14203-1_12`.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: `http://mitpress.mit.edu/books/certified-programming-dependent-types`.

[Chu+07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. "Attested append-only memory: making adversaries stick to their word". In: *SOSP*. Ed. by Thomas C. Bressoud and M. Frans Kaashoek. ACM, 2007, pp. 189–204. DOI: `10.1145/1294261.1294280`. URL: `http://doi.acm.org/10.1145/1294261.1294280`.

[CL02] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. DOI: `10.1145/571637.571640`. URL: `http://doi.acm.org/10.1145/571637.571640`.

[CL85] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (1985), pp. 63–75. DOI: `10.1145/214451.214456`. URL: `http://doi.acm.org/10.1145/214451.214456`.

[CL99a] Miguel Castro and Barbara Liskov. *A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm.* Technical Memo MIT-LCS-TM-590. MIT, 1999.

[CL99b] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI 1999.* Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 173–186. DOI: `10.1145/296806.296824`. URL: `http://doi.acm.org/10.1145/296806.296824`.

[CLS16] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. "Formal Verification of Multi-Paxos for Distributed Consensus". In: *FM 2016.* Ed. by John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou. Vol. 9995. LNCS. 2016, pp. 119–136. DOI: `10.1007/978-3-319-48989-6_8`. URL: `https://doi.org/10.1007/978-3-319-48989-6_8`.

[CM06] Judicaël Courant and Jean-François Monin. "Defending the bank with a proof assistant". In: *WITS 2006.* In WITS proceedings. Vienna, Mar. 2006.

[CM86] K. Mani Chandy and Jayadev Misra. "How Processes Learn". In: *Distributed Computing* 1.1 (1986), pp. 40–52. DOI: `10.1007/BF01843569`. URL: `https://doi.org/10.1007/BF01843569`.

[CNV04] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems". In: *SRDS 2004.* IEEE Computer Society, 2004, pp. 174–183. DOI: `10.1109/RELDIS.2004.1353018`. URL: `http://dx.doi.org/10.1109/RELDIS.2004.1353018`.

[CNV13] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "BFT-TO: Intrusion Tolerance with Less Replicas". In: *Comput. J.* 56.6 (2013), pp. 693–715. DOI: `10.1093/comjnl/bxs148`. URL: `http://dx.doi.org/10.1093/comjnl/bxs148`.

[Coa86] Brian A. Coan. "A Communication-Efficient Canonical Form for Fault-Tolerant Distributed Protocols". In: *PODC 1986.* Ed. by Joseph Y. Halpern. ACM, 1986, pp. 63–72. DOI: `10.1145/10590.10596`. URL: `https://doi.org/10.1145/10590.10596`.

[Con+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P.Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing mathematics with the Nuprl proof development system.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[Cor+05] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. "Low complexity Byzantine-resilient consensus". In: *Distributed Computing* 17.3 (2005), pp. 237–249. DOI: `10.1007/s00446-004-0110-7`. URL: `https://doi.org/10.1007/s00446-004-0110-7`.

[CS09] Bernadette Charron-Bost and André Schiper. "The Heard-Of model: computing in distributed systems with benign faults". In: *Distributed Computing* 22.1 (2009), pp. 49–71. DOI: `10.1007/s00446-009-0084-6`. URL: `https://doi.org/10.1007/s00446-009-0084-6`.

[CVN02] Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. "The Design of a COTSReal-Time Distributed Security Kernel". In: *EDCC-4*. Ed. by Fabrizio Grandoni and Pascale Thévenod-Fosse. Vol. 2485. LNCS. Springer, 2002, pp. 234–252. DOI: `10.1007/3-540-36080-8_21`. URL: `https://doi.org/10.1007/3-540-36080-8_21`.

[CW02] Hao Chen and David A. Wagner. "MOPS: an infrastructure for examining security properties of software". In: *CCS*. Ed. by Vijayalakshmi Atluri. ACM, 2002, pp. 235–244. DOI: `10.1145/586110.586142`. URL: `http://doi.acm.org/10.1145/586110.586142`.

[DAB11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. "Logical Step-Indexed Logical Relations". In: *Logical Methods in Computer Science* 7.2 (2011). DOI: `10.2168/LMCS-7(2:16)2011`. URL: `https://doi.org/10.2168/LMCS-7(2:16)2011`.

[Dat+09] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kirli Kaynar. "A Logic of Secure Systems and its Application to Trusted Computing". In: *SP 2009*. IEEE Computer Society, 2009, pp. 221–236. DOI: `10.1109/SP.2009.16`. URL: `https://doi.org/10.1109/SP.2009.16`.

[DCK16] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. "Resource-Efficient Byzantine Fault Tolerance". In: *IEEE Trans. Computers* 65.9 (2016), pp. 2807–2819. DOI: `10.1109/TC.2015.2495213`. URL: `https://doi.org/10.1109/TC.2015.2495213`.

[Del+10] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. "A Formal Analysis of Authentication in the TPM". In: *FAST*. Ed. by Pierpaolo Degano, Sandro Etalle, and Joshua D. Guttman. Vol. 6561. LNCS. Springer, 2010, pp. 111–125. DOI: `10.1007/978-3-642-19751-2\_8`. URL: `https://doi.org/10.1007/978-3-642-19751-2%5C_8`.

[Del+11] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. “Formal Analysis of Protocols Based on TPM State Registers”. In: *CSF*. IEEE Computer Society, 2011, pp. 66–80. DOI: `10.1109/CSF.2011.12`. URL: `https://doi.org/10.1109/CSF.2011.12`.

[Des+18] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. “Compositional programming and testing of dynamic distributed systems”. In: *PACMPL* 2.OOPSLA (2018), 159:1–159:30. DOI: `10.1145/3276529`. URL: `https://doi.org/10.1145/3276529`.

[DHZ15] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. “The Need for Language Support for Fault-Tolerant Distributed Systems”. In: *SNAPL 2015*. Ed. by Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 90–102. DOI: `10.4230/LIPIcs.SNAPL.2015.90`. URL: `https://doi.org/10.4230/LIPIcs.SNAPL.2015.90`.

[DHZ16] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. “PSync: a partially synchronous language for fault-tolerant distributed algorithms”. In: *POPL 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 400–415. DOI: `10.1145/2837614.2837650`. URL: `http://doi.acm.org/10.1145/2837614.2837650`.

[DJN18] Bruno Dutertre, Dejan Jovanovic, and Jorge A. Navas. “Verification of Fault-Tolerant Protocols with Sally”. In: *NFM 2018*. Ed. by Aaron Dutle, César A. Muñoz, and Anthony Narkawicz. Vol. 10811. LNCS. Springer, 2018, pp. 113–120. DOI: `10.1007/978-3-319-77935-5\_8`. URL: `https://doi.org/10.1007/978-3-319-77935-5%5C_8`.

[DM90] Cynthia Dwork and Yoram Moses. “Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures”. In: *Inf. Comput.* 88.2 (1990), pp. 156–186. DOI: `10.1016/0890-5401(90)90014-9`. URL: `https://doi.org/10.1016/0890-5401(90)90014-9`.

[DMM17] Asa Dan, Rajit Manohar, and Yoram Moses. “On Using Time Without Clocks via Zigzag Causality”. In: *PODC 2017*. Ed. by Elad Michael Schiller and Alexander A. Schwarzmann. ACM, 2017, pp. 241–250. DOI: `10.1145/3087801.3087839`. URL: `https://doi.org/10.1145/3087801.3087839`.

[Dra+14] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. “A Logic-Based Framework for Verifying Consensus Algorithms”. In: *VMCAI 2014*. Ed. by Kenneth L. McMillan and Xavier Rival. Vol. 8318. LNCS. Springer, 2014, pp. 161–181. DOI:

10.1007/978-3-642-54013-4_10. URL: https://doi.org/10.1007/978-3-642-54013-4_10.

[DRS90] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. "Early Stopping in Byzantine Agreement". In: *J. ACM* 37.4 (1990), pp. 720–741. DOI: 10.1145/96559.96565. URL: https://doi.org/10.1145/96559.96565.

[DSW16] Christian Decker, Jochen Seidel, and Roger Wattenhofer. "Bitcoin meets strong consistency". In: *ICDCN*. ACM, 2016, 13:1–13:10. DOI: 10.1145/2833312.2833321. URL: http://doi.acm.org/10.1145/2833312.2833321.

[Edm+12] Andrew Edmunds, Michael J. Butler, Issam Maamria, Renato Silva, and Chris Lovell. "Event-B Code Generation: Type Extension with Theories". In: *ABZ*. Ed. by John Derrick, John S. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene. Vol. 7316. Lecture Notes in Computer Science. Springer, 2012, pp. 365–368. DOI: 10.1007/978-3-642-30885-7_33. URL: https://doi.org/10.1007/978-3-642-30885-7%5C_33.

[ERT17] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)". In: *WiSec 2017*. Ed. by Guevara Noubir, Mauro Conti, and Sneha Kumar Kasera. ACM, 2017, pp. 99–110. DOI: 10.1145/3098243.3098261. URL: https://doi.org/10.1145/3098243.3098261.

[Fag+03] Ronald Fagin, Joseph Halpern, Yoram Moses, and Moshe Vardi. *Reasoning About Knowledge*. Jan. 2003. DOI: 10.7551/mitpress/5803.001.0001.

[Fag+97] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. "Knowledge-Based Programs". In: *Distributed Computing* 10.4 (1997), pp. 199–225. DOI: 10.1007/s004460050038. URL: https://doi.org/10.1007/s004460050038.

[Fer+13a] Matthew Fernandez, Peter Gammie, June Andronick, Gerwin Klein, and Ihor Kuz. *CAmkES Glue Code Semantics*. Tech. rep. Australia: NICTA and UNSW, 2013.

[Fer+13b] Matthew Fernandez, Gerwin Klein, Ihor Kuz, and Toby Murray. *CAmkES Formalisation of a Component Platform*. Tech. rep. Australia: NICTA and UNSW, 2013.

[Fer+17a] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using verification to disentangle secure-enclave hardware from software”. In: *SOSP*. ACM, 2017, pp. 287–305. DOI: `10.1145/3132747.3132782`. URL: `http://doi.acm.org/10.1145/3132747.3132782`.

[Fer+17b] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. “Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis”. In: *ASPLOS 2017*. Ed. by Yunji Chen, Olivier Temam, and John Carter. ACM, 2017, pp. 555–568. DOI: `10.1145/3037697.3037739`. URL: `http://doi.acm.org/10.1145/3037697.3037739`.

[Fer16] Matthew Fernandez. “Formal Verification of a Component Platform”. PhD thesis. Sydney, Australia: UNSW Computer Science & Engineering, 2016.

[Fet03] Christof Fetzer. “Perfect Failure Detection in Timed Asynchronous Systems”. In: *IEEE Trans. Computers* 52.2 (2003), pp. 99–112. DOI: `10.1109/TC.2003.1176979`. URL: `https://doi.org/10.1109/TC.2003.1176979`.

[Fon+17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. “An Empirical Study on the Correctness of Formally Verified Distributed Systems”. In: *EUROSYS 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 328–343. DOI: `10.1145/3064176.3064183`. URL: `http://doi.acm.org/10.1145/3064176.3064183`.

[Für+14a] Andreas Fürst, Thai Son Hoang, David A. Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. “Code Generation for Event-B”. In: *IFM 2014*. Ed. by Elvira Albert and Emil Sekerinski. Vol. 8739. LNCS. Springer, 2014, pp. 323–338. DOI: `10.1007/978-3-319-10181-1_20`. URL: `http://dx.doi.org/10.1007/978-3-319-10181-1_20`.

[Für+14b] Andreas Fürst, Thai Son Hoang, David A. Basin, Naoto Sato, and Kunihiko Miyazaki. “Formal System Modelling Using Abstract Data Types in Event-B”. In: *ABZ 2014*. Ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. LNCS. Springer, 2014, pp. 222–237. DOI: `10.1007/978-3-662-43652-3\_20`. URL: `https://doi.org/10.1007/978-3-662-43652-3%5C_20`.

[GA19] Haroldas Giedra and Romas Alonderis. “Automated proof search system for logic of correlated knowledge”. In: *CoRR* abs/1902.08847 (2019). arXiv: `1902.08847`. URL: `http://arxiv.org/abs/1902.08847`.

[Gan+05] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. “Automatic discovery of API-level exploits”. In: *ICSE 2005*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 312–321. DOI: `10.1145/1062455.1062518`. URL: `http://doi.acm.org/10.1145/1062455.1062518`.

[Gar+04] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. *IOA user guide and reference manual*. Tech. rep. MIT/LCS/TR-961. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[Geo+09] Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. “Automated implementation of complex distributed algorithms specified in the IOA language”. In: *Int. J. Softw. Tools Technol. Transf.* 11 (2 Feb. 2009), pp. 153–171. DOI: `10.1007/s10009-008-0097-7`. URL: `http://dl.acm.org/citation.cfm?id=1529842.1529844`.

[Gie14] Haroldas Giedra. “Proof System for Logic of Correlated Knowledge”. PhD thesis. Vilnius University, 2014.

[Gif79] David K. Gifford. “Weighted Voting for Replicated Data”. In: *SOSP*. Ed. by Michael D. Schroeder and Anita K. Jones. ACM, 1979, pp. 150–162. DOI: `10.1145/800215.806583`. URL: `http://doi.acm.org/10.1145/800215.806583`.

[GL00] Stephen J. Garland and Nancy Lynch. “Using I/O automata for developing distributed systems”. In: *Foundations of componentbased systems*. Ed. by Gary T. Leavens and Murali Sitaraman. New York, NY, USA: Cambridge University Press, 2000, pp. 285–312. URL: `http://dl.acm.org/citation.cfm?id=336431.336455`.

[GLR95] Li Gong, Patrick Lincoln, and John Rushby. “Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults”. In: *Dependable Computing for Critical Applications—5*. Ed. by Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor. Vol. 10. Dependable Computing and Fault Tolerant Systems. Champaign, IL: IEEE Computer Society, Sept. 1995, pp. 139–157. URL: `http://www.csl.sri.com/papers/dcca95/`.

[GM18] Guy Goren and Yoram Moses. “Silence”. In: *PODC 2018*. Ed. by Calvin Newport and Idit Keidar. ACM, 2018, pp. 285–294. DOI: `10.1145/3212734`. URL: `https://dl.acm.org/citation.cfm?id=3212768`.

[Gua15] Bai Guangdong. "Formally Analyzing and Verifying Secure System Design and Implementation". PhD thesis. National Univeristy of Singapore, 2015.

[Gue+10] Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. "The next 700 BFT protocols". In: *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. Ed. by Christine Morin and Gilles Muller. ACM, 2010, pp. 363–376. DOI: `10.1145/1755913.1755950`. URL: `http://doi.acm.org/10.1145/1755913.1755950`.

[Hal08] Stefan Hallerstede. "Incremental System Modelling in Event-B". In: *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine. Vol. 5751. Lecture Notes in Computer Science. Springer, 2008, pp. 139–158. DOI: `10.1007/978-3-642-04167-9\_8`. URL: `https://doi.org/10.1007/978-3-642-04167-9%5C_8`.

[Hal87] Joseph Y. Halpern. "Using Reasoning About Knowledge to Analyze Distributed Systems". In: *Annual Review of Computer Science* 2.1 (1987), pp. 37–68. DOI: `10.1146/annurev.cs.02.060187.000345`. eprint: `https://doi.org/10.1146/annurev.cs.02.060187.000345`. URL: `https://doi.org/10.1146/annurev.cs.02.060187.000345`.

[Haw+14] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-End Security via Automated Full-System Verification". In: *OSDI '14*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 165–181. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel`.

[Haw+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct". In: *SOSP*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 1–17. DOI: `10.1145/2815400.2815428`. URL: `http://doi.acm.org/10.1145/2815400.2815428`.

[Haw+17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving safety and liveness of practical distributed systems". In:

Commun. ACM 60.7 (2017), pp. 83–92. DOI: `10.1145/3068608`. URL: `http://doi.acm.org/10.1145/3068608`.

[HM90] Joseph Y. Halpern and Yoram Moses. "Knowledge and Common Knowledge in a Distributed Environment". In: *J. ACM* 37.3 (1990), pp. 549–587. DOI: `10.1145/79147.79161`. URL: `http://doi.acm.org/10.1145/79147.79161`.

[HN07] Raul Hakli and Sara Negri. "Proof Theory for Distributed Knowledge". In: *CLIMA 2007*. Ed. by Fariba Sadri and Ken Satoh. Vol. 5056. LNCS. Springer, 2007, pp. 100–116. DOI: `10.1007/978-3-540-88833-8\_6`. URL: `https://doi.org/10.1007/978-3-540-88833-8%5C_6`.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[HP17] Joseph Y. Halpern and Rafael Pass. "A Knowledge-Based Analysis of the Blockchain Protocol". In: *TARK 2017*. Ed. by Jérôme Lang. Vol. 251. EPTCS. 2017, pp. 324–335. DOI: `10.4204/EPTCS.251.22`. URL: `https://doi.org/10.4204/EPTCS.251.22`.

[HW87] Maurice Herlihy and Jeannette M. Wing. "Axioms for Concurrent Objects". In: *POPL 1987*. ACM Press, 1987, pp. 13–26. DOI: `10.1145/41625.41627`. URL: `http://doi.acm.org/10.1145/41625.41627`.

[HZ92] Joseph Y. Halpern and Lenore D. Zuck. "A Little Knowledge Goes a Long Way: Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols". In: *J. ACM* 39.3 (1992), pp. 449–478. DOI: `10.1145/146637.146638`. URL: `http://doi.acm.org/10.1145/146637.146638`.

[Jaj+11] Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and Xiaoyang Sean Wang, eds. *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*. Vol. 54. Advances in Information Security. Springer, 2011. DOI: `10.1007/978-1-4614-0977-9`. URL: `https://doi.org/10.1007/978-1-4614-0977-9`.

[JD16] Dejan Jovanovic and Bruno Dutertre. "Property-directed k-induction". In: *FMCAD 2016*. Ed. by Ruzica Piskac and Muralidhar Talupur. IEEE, 2016, pp. 85–92. DOI: `10.1109/FMCAD.2016.7886665`. URL: `https://doi.org/10.1109/FMCAD.2016.7886665`.

[Jia+15] Limin Jia, Shayak Sen, Deepak Garg, and Anupam Datta. "A Logic of Programs with Interface-Confined Code". In: *CSF*. Ed. by Cédric Fournet, Michael W. Hicks, and Luca Viganò. IEEE Computer Society, 2015, pp. 512–525. DOI: `10.1109/CSF.2015.38`. URL: `https://doi.org/10.1109/CSF.2015.38`.

[Joh+12] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. "Starting a Dialog between Model Checking and Fault-tolerant Distributed Algorithms". In: *CoRR* abs/1210.3839 (2012). arXiv: `1210.3839`. URL: `http://arxiv.org/abs/1210.3839`.

[Jos+03] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. "Checking Cache-Coherence Protocols with TLA^{+}". In: *Formal Methods in System Design* 22.2 (2003), pp. 125–131. DOI: `10.1023/A:1022969405325`. URL: `http://dx.doi.org/10.1023/A:1022969405325`.

[Jug+16] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. "Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation". In: *CSF*. IEEE Computer Society, 2016, pp. 45–60. DOI: `10.1109/CSF.2016.11`. URL: `https://doi.org/10.1109/CSF.2016.11`.

[Kap+12] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. "CheapBFT: resource-efficient byzantine fault tolerance". In: *EuroSys '12*. Ed. by Pascal Felber, Frank Bellosa, and Herbert Bos. ACM, 2012, pp. 295–308. DOI: `10.1145/2168836.2168866`. URL: `http://doi.acm.org/10.1145/2168836.2168866`.

[KB07] William J. Knottenbelt and Jeremy T. Bradley. "Tackling Large State Spaces in Performance Modelling". In: *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*. Ed. by Marco Bernardo and Jane Hillston. Vol. 4486. Lecture Notes in Computer Science. Springer, 2007, pp. 318–370. DOI: `10.1007/978-3-540-72522-0\_8`. URL: `https://doi.org/10.1007/978-3-540-72522-0%5C_8`.

[Kei06] Gavin Keighren. "Model Checking Security APIs". PhD thesis. School of informatics, Univeristy of Edinburg, 2006.

[Kle+09] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *SOSP*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: `10.1145/1629575.1629596`. URL: `http://doi.acm.org/10.1145/1629575.1629596`.

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive formal verification of an OS microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (2014), 2:1–2:70. DOI: `10.1145/2560537`. URL: `http://doi.acm.org/10.1145/2560537`.

[Kle+18] Gerwin Klein, June Andronick, Matthew Frenandez, Ihor Kuz, Toby Murray, and Gernot Heiser. "Formally Verified Software in the Real World". Communicatons of the ACM. 2018. URL: `http://ssrg.nicta.com/publications/csiro_full_text/Klein_AKMHF_toappear.pdf`.

[Kok+16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 279–296. URL: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias`.

[Kon+17] Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. "A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms". In: *POPL 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 719–734. DOI: `10.1145/3009837`. URL: `http://dl.acm.org/citation.cfm?id=3009860`.

[Kro+18] Moren Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, and Lars Birkedal. "Aneris: A Logic for Node-Local, Modular Reasoning ofDistributed Systems". 2018. URL: `https://iris-project.org/pdfs/2019-aneris-submission.pdf`.

[Kro18] Morten Krogh-Jespersen. "Towards Modular Reasoning for Stateful and ConcurrentPrograms". PhD thesis. Aarhus University, 2018.

[KU10] Roman Krenický and Mattias Ulbrich. *Deductive Verification of a Byzantine Agreement Protocol.* Tech. rep. 2010-7. Karlsruhe Institute of Technology, Department of Computer Science, 2010. URL: `https://lfm.iti.kit.edu/english/769.php`.

[Kus13] David Kushner. "The real story of stuxnet". In: *Spectrum, IEEE* 50 (Mar. 2013), pp. 48–53. DOI: `10.1109/MSPEC.2013.6471059`.

[Kuz+07] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. "CAmkES: A component model for secure microkernel-based embedded systems". In: *Journal of Systems and Software* 80.5 (2007), pp. 687–699. DOI: `10.1016/j.jss.2006.08.039`. URL: `https://doi.org/10.1016/j.jss.2006.08.039`.

[KVW15] Igor Konnov, Helmut Veith, and Josef Widder. "SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms". In: *CAV*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. LNCS. Springer, 2015, pp. 85–102. DOI: `10.1007/978-3-319-21690-4_6`. URL: `https://doi.org/10.1007/978-3-319-21690-4_6`.

[KVW17] Igor V. Konnov, Helmut Veith, and Josef Widder. "On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability". In: *Inf. Comput.* 252 (2017), pp. 95–109. DOI: `10.1016/j.ic.2016.03.006`. URL: `https://doi.org/10.1016/j.ic.2016.03.006`.

[Lam04] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2004.

[Lam11] Leslie Lamport. "Byzantizing Paxos by Refinement". In: *DISC 2011*. Ed. by David Peleg. Vol. 6950. LNCS. Springer, 2011, pp. 211–224. DOI: `10.1007/978-3-642-24100-0_22`. URL: `http://dx.doi.org/10.1007/978-3-642-24100-0_22`.

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: `10.1145/359545.359563`. URL: `http://doi.acm.org/10.1145/359545.359563`.

[Lam94] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923. DOI: `10.1145/177492.177726`. URL: `http://doi.acm.org/10.1145/177492.177726`.

[Lam95] Leslie Lamport. "How to Write a Proof". In: *American Mathematical Monthly. Also appeared in Global Analysis in Modern Mathematics, Karen Uhlenbeck, editor. Publish or Perish Press, Houston. Also appeared as SRC Research Report 94.* 102.7 (Aug. 1995), pp. 600–608. URL: `https://www.microsoft.com/en-us/research/publication/how-to-write-a-proof/`.

[Lam98] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. URL: `http://dl.acm.org/citation.cfm?id=279229`.

[Laz+17] Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. “Synthesis of Distributed Algorithms with Parameterized Threshold Guards”. In: *OPODIS 2017*. Ed. by James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão. Vol. 95. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 32:1–32:20. DOI: `10.4230/LIPIcs.OPODIS.2017.32`. URL: `https://doi.org/10.4230/LIPIcs.OPODIS.2017.32`.

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. “Chapar: certified causally consistent distributed key-value stores”. In: *POPL 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 357–370. DOI: `10.1145/2837614.2837622`. URL: `http://doi.acm.org/10.1145/2837614.2837622`.

[LCF15] Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. “Verifying Linearizability of Intel® Software Guard Extensions”. In: *CAV*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. LNCS. Springer, 2015, pp. 144–160. DOI: `10.1007/978-3-319-21668-3_9`. URL: `https://doi.org/10.1007/978-3-319-21668-3_9`.

[Lei10] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *LPAR-16*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. LNCS. Springer, 2010, pp. 348–370. DOI: `10.1007/978-3-642-17511-4_20`. URL: `http://dx.doi.org/10.1007/978-3-642-17511-4_20`.

[Lev+09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. “TrInc: Small Trusted Hardware for Large Distributed Systems”. In: *NSDI*. Ed. by Jennifer Rexford and Emin Gün Sirer. USENIX Association, 2009, pp. 1–14. URL: `http://www.usenix.org/events/nsdi09/tech/full_papers/levin/levin.pdf`.

[LM94] Leslie Lamport and Stephan Merz. “Specifying and Verifying Fault-Tolerant Systems”. In: *FTRTFTS 1994*. Ed. by Hans Langmaack, Willem P. de Roever, and Jan Vytopil. Vol. 863. LNCS. Springer, 1994, pp. 41–76. DOI: `10.1007/3-540-58468-4_159`. URL: `https://doi.org/10.1007/3-540-58468-4_159`.

[LMW11] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. “Towards Verification of the Pastry Protocol Using TLA $^+$ ”. In: *FORTE 2011*. Ed. by Roberto Bruni and Jürgen Dingel. Vol. 6722. LNCS. Springer, 2011, pp. 244–258. DOI: `10.1007/978-3-642-21461-5_16`. URL: `http://dx.doi.org/10.1007/978-3-642-21461-5_16`.

[Lok+19] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. "Fast and secure global payments with Stellar". In: *SOSP 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 80–96. DOI: `10.1145/3341301.3359636`. URL: `https://doi.org/10.1145/3341301.3359636`.

[LR93a] Patrick Lincoln and John Rushby. *A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model.* Tech. rep. SRI-CSL-93-2. Also available as NASA Contractor Report 4527, July 1993. Computer Science Laboratory, SRI International, Mar. 1993. URL: `http://www.csl.sri.com/papers/csl-93-2/`.

[LR93b] Patrick Lincoln and John M. Rushby. "The Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model". In: *CAV*. Ed. by Costas Courcoubetis. Vol. 697. LNCS. Springer, 1993, pp. 292–304. DOI: `10.1007/3-540-56922-7_24`. URL: `https://doi.org/10.1007/3-540-56922-7_24`.

[LSL17] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. "From Clarity to Efficiency for Distributed Algorithms". In: *ACM Trans. Program. Lang. Syst.* 39.3 (2017), 12:1–12:41. DOI: `10.1145/2994595`. URL: `https://doi.org/10.1145/2994595`.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. DOI: `10.1145/357172.357176`. URL: `http://doi.acm.org/10.1145/357172.357176`.

[LT87] Nancy A. Lynch and Mark R. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In: *PODC 1987*. Ed. by Fred B. Schneider. ACM, 1987, pp. 137–151. DOI: `10.1145/41840.41852`. URL: `http://doi.acm.org/10.1145/41840.41852`.

[Luu+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. "A Secure Sharding Protocol For Open Blockchains". In: *CCS*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 17–30. DOI: `10.1145/2976749.2978389`. URL: `http://doi.acm.org/10.1145/2976749.2978389`.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[Mat89] Friedemann Mattern. “Virtual Time and Global States of Distributed Systems”. In: *Proc. Workshop on Parallel and Distributed Algorithms.* Ed. by Cosnard M. et al. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.) North-Holland / Elsevier, 1989, pp. 215–226.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS-14*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. LNCS. Springer, 2008, pp. 337–340. DOI: `10.1007/978-3-540-78800-3_24`. URL: `http://dx.doi.org/10.1007/978-3-540-78800-3_24`.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Vol. 92. Lecture Notes in Computer Science. Springer, 1980. DOI: `10.1007/3-540-10235-3`. URL: `https://doi.org/10.1007/3-540-10235-3`.

[Mog89] Eugenio Moggi. “Computational Lambda-Calculus and Monads”. In: *LICS*. IEEE Computer Society, 1989, pp. 14–23.

[MR97] Dahlia Malkhi and Michael K. Reiter. “Byzantine Quorum Systems”. In: *STOC 1997*. Ed. by Frank Thomson Leighton and Peter W. Shor. ACM, 1997, pp. 569–578. DOI: `10.1145/258533.258650`. URL: `http://doi.acm.org/10.1145/258533.258650`.

[MS11] Dominique Méry and Neeraj Kumar Singh. “Automatic code generation from event-B models”. In: *Symposium on Information and Communication Technology, SoICT 2011.* Ed. by Huynh Quyet Thang and Dinh Khang Tran. ACM, 2011, pp. 179–188. DOI: `10.1145/2069216.2069252`. URL: `http://doi.acm.org/10.1145/2069216.2069252`.

[MSB15] Ognjen Maric, Christoph Sprenger, and David A. Basin. “Consensus Refined”. In: *DSN 2015*. IEEE Computer Society, 2015, pp. 391–402. DOI: `10.1109/DSN.2015.38`. URL: `https://doi.org/10.1109/DSN.2015.38`.

[MSB17] Ognjen Maric, Christoph Sprenger, and David A. Basin. “Cutoff Bounds for Consensus Algorithms”. In: *CAV*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. LNCS. Springer, 2017, pp. 217–237. DOI: `10.1007/978-3-319-63390-9\_12`. URL: `https://doi.org/10.1007/978-3-319-63390-9%5C_12`.

[New+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. “How Amazon web services uses formal methods”. In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: `10.1145/2699417`. URL: `http://doi.acm.org/10.1145/2699417`.

[New14] Chris Newcombe. “Why Amazon Chose TLA +”. In: *ABZ 2014*. Ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. LNCS. Springer, 2014, pp. 25–39. DOI: `10.1007/978-3-662-43652-3_3`. URL: `http://dx.doi.org/10.1007/978-3-662-43652-3_3`.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. “Petri Nets, Event Structures and Domains, Part I”. In: *Theor. Comput. Sci.* 13 (1981), pp. 85–108. DOI: `10.1016/0304-3975(81)90112-2`. URL: `https://doi.org/10.1016/0304-3975(81)90112-2`.

[OO14] Diego Ongaro and John K. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.* Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[Owr+95] Sam Owre, John M. Rushby, Natarajan Shankar, and Friedrich W. von Henke. “Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS”. In: *IEEE Trans. Software Eng.* 21.2 (1995), pp. 107–125. DOI: `10.1109/32.345827`. URL: `https://doi.org/10.1109/32.345827`.

[P B89] K.J. Perry P. Berman J.A. Garay. *Asymptotically optimal distributed consensus.* Tech. rep. Bell Labs, 1989. URL: `plan9.bell-labs.co/who/garay/asopt.ps`.

[Pad+16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. “Ivy: safety verification by interactive generalization”. In: *PLDI 2016.* Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 614–630. DOI: `10.1145/2908080.2908118`. URL: `http://doi.acm.org/10.1145/2908080.2908118`.

[Pad+17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. “Paxos made EPR: decidable reasoning about distributed protocols”. In: *PACMPL* 1.OOPSLA (2017), 108:1–108:31. DOI: `10.1145/3140568`. URL: `http://doi.acm.org/10.1145/3140568`.

[Pad+18] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. “Reducing liveness to safety in first-order logic”. In: *PACMPL* 2.POPL (2018), 26:1–26:33. DOI: `10.1145/3158114`. URL: `http://doi.acm.org/10.1145/3158114`.

[Pie+18] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, 2018.

[PS17] Rafael Pass and Elaine Shi. "Hybrid Consensus: Efficient Consensus in the Permissionless Model". In: *DISC*. Ed. by Andréa W. Richa. Vol. 91. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 39:1–39:16. DOI: `10.4230/LIPIcs.DISC.2017.39`. URL: `https://doi.org/10.4230/LIPIcs.DISC.2017.39`.

[PT12] Giuseppe Primiero and Mariarosaria Taddeo. "A modal type theory for formalizing trusted communications". In: *J. Applied Logic* 10.1 (2012), pp. 92–114. DOI: `10.1016/j.jal.2011.12.002`. URL: `https://doi.org/10.1016/j.jal.2011.12.002`.

[PT92] Prakash Panangaden and Kim Taylor. "Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems". In: *Distributed Computing* 6.2 (1992), pp. 73–93. DOI: `10.1007/BF02252679`. URL: `https://doi.org/10.1007/BF02252679`.

[RA15] Robbert van Renesse and Deniz Altinbuken. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* 47.3 (2015), 5:1–5:36. DOI: `10.1145/2673577`. URL: `http://dx.doi.org/10.1145/2673577`.

[Rah+12] Vincent Rahli, Nicolas Schiper, Robbert van Renesse, Mark Bickford, and Robert L. Constable. "A diversified and correct-by-construction broadcast service". In: *ICNP 2012*. IEEE Computer Society, 2012, pp. 1–6. DOI: `10.1109/ICNP.2012.6459943`. URL: `https://doi.org/10.1109/ICNP.2012.6459943`.

[Rah+15] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. "Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML". In: *ECEASST* 72 (2015). URL: `http://journal.ub.tu-berlin.de/eceasst/article/view/1013`.

[Rah+17] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. "EventML: Specification, Verification, and Implementation of Crash-Tolerant State Machine Replication Systems". In: *SCP* (2017).

[Ray10] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010. DOI: `10.2200/S00294ED1V01Y201009DCT003`. URL: `https://doi.org/10.2200/S00294ED1V01Y201009DCT003`.

[RBA13] Vincent Rahli, Mark Bickford, and Abhishek Anand. “Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types”. In: *ITP 2013*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. LNCS. Springer, 2013, pp. 261–278. DOI: `10.1007/978-3-642-39634-2_20`. URL: `https://doi.org/10.1007/978-3-642-39634-2_20`.

[Rey02] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *LICS*. IEEE Computer Society, 2002, pp. 55–74. DOI: `10.1109/LICS.2002.1029817`. URL: `http://dx.doi.org/10.1109/LICS.2002.1029817`.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[RHO91] John Rushby, Friedrich von Henke, and Sam Owre. *An Introduction to Formal Specification and Verification Using* EHDM. Tech. rep. SRI-CSL-91-2. Menlo Park, CA: Computer Science Laboratory, SRI International, 1991.

[Roe07] Floris Roelofsen. “Distributed knowledge”. In: *Journal of Applied Non-Classical Logics* 17.2 (2007), pp. 255–273. DOI: `10.3166/jancl.17.255-273`. URL: `https://doi.org/10.3166/jancl.17.255-273`.

[RS04] Robbert van Renesse and Fred B. Schneider. “Chain Replication for Supporting High Throughput and Availability”. In: *OSDI 2004*. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 2004, pp. 91–104. URL: `http://www.usenix.org/events/osdi04/tech/renesse.html`.

[Rus01] John Rushby. *Formal Verification of Hybrid Byzantine Agreement Under Link Faults*. Tech. rep. Oct. 2001.

[Rus03] John Rushby. *Formal Verification of an Oral Messages Algorithm for Interactive Consistency*. Tech. rep. 2003.

[SBV18] João Sousa, Alysson Bessani, and Marko Vukolic. “A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform”. In: *DSN 2018*. IEEE Computer Society, 2018, pp. 51–58. DOI: `10.1109/DSN.2018.00018`. URL: `https://doi.org/10.1109/DSN.2018.00018`.

[Sch+14] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. "Developing Correctly Replicated Databases Using Formal Tools". In: *DSN 2014*. IEEE, 2014, pp. 395–406. DOI: `10.1109/DSN.2014.45`. URL: `http://dx.doi.org/10.1109/DSN.2014.45`.

[Sha+15] Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. "Formal Analysis of Enhanced Authorization in the TPM 2.0". In: *ASIA CCS*. Ed. by Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn. ACM, 2015, pp. 273–284. DOI: `10.1145/2714576.2714610`. URL: `http://doi.acm.org/10.1145/2714576.2714610`.

[Sin+15] Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. "Moat: Verifying Confidentiality of Enclave Programs". In: *CCS*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 1169–1184. DOI: `10.1145/2810103.2813608`. URL: `http://doi.acm.org/10.1145/2810103.2813608`.

[SM16] Alexander J. Summers and Peter Müller. "Actor Services - Modular Verification of Message Passing Programs". In: *ESOP 2016*. Ed. by Peter Thiemann. Vol. 9632. LNCS. Springer, 2016, pp. 699–726. DOI: `10.1007/978-3-662-49498-1_27`. URL: `https://doi.org/10.1007/978-3-662-49498-1_27`.

[Sou07] Paulo Sousa. "Proactive Resilience". PhD thesis. Lisbon, Portugal: Faculty of Sciences, University of Lisbon, 2007.

[SQF18] Jianxiong Shao, Yu Qin, and Dengguo Feng. "Formal analysis of HMAC authorisation in the TPM2.0 specification". In: *IET Information Security* 12.2 (2018), pp. 133–140. DOI: `10.1049/iet-ifs.2016.0005`. URL: `https://doi.org/10.1049/iet-ifs.2016.0005`.

[ST87] T. K. Srikanth and Sam Toueg. "Simulating Authenticated Broadcasts to Derive Simple Fault-Tolerant Algorithms". In: *Distributed Computing* 2.2 (1987), pp. 80–94. DOI: `10.1007/BF01667080`. URL: `https://doi.org/10.1007/BF01667080`.

[Sto+19] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. "Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking". In: *TACAS 2019*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 357–374. DOI: `10.1007/978-3-030-17465-1\_20`. URL: `https://doi.org/10.1007/978-3-030-17465-1%5C_20`.

[Sub+17] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. “A Formal Foundation for Secure Remote Execution of Enclaves”. In: *CCS*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 2435–2450. DOI: `10.1145/3133956.3134098`. URL: `http://doi.acm.org/10.1145/3133956.3134098`.

[SWR02] Ulrich Schmid, Bettina Weiss, and John M. Rushby. “Formally Verified Byzantine Agreement in Presence of Link Faults”. In: *ICDCS*. 2002, pp. 608–616. DOI: `10.1109/ICDCS.2002.1022311`. URL: `https://doi.org/10.1109/ICDCS.2002.1022311`.

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. “Programming and Proving with Distributed Protocols”. In: *POPL 2018*. 2018.

[Tau+18] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. “Modularity for decidability of deductive verification with applications to distributed systems”. In: *PLDI 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 662–677. DOI: `10.1145/3192366.3192414`. URL: `http://doi.acm.org/10.1145/3192366.3192414`.

[Tau04] Joshua A. Tauber. “Verifiable Compilation of I/O Automata without Global Synchronization”. PhD thesis. Departement of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[TG98] Ashis Tarafdar and Vijay K. Garg. “Addressing False Causality while Detecting Predicates in Distributed Programs”. In: *ICDCS 1998*. IEEE Computer Society, 1998, pp. 94–101. DOI: `10.1109/ICDCS.1998.679491`. URL: `https://doi.org/10.1109/ICDCS.1998.679491`.

[Tho79] Robert H. Thomas. “A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases”. In: *ACM Trans. Database Syst.* 4.2 (1979), pp. 180–209. DOI: `10.1145/320071.320076`. URL: `http://doi.acm.org/10.1145/320071.320076`.

[Tog13] Ronald Togl. “On Trusted Computing Interfaces”. PhD thesis. Faculty of Computer Science, Graz University of Technology, 2013.

[TPV17] Chia-che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *USENIX ATC*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 2017, pp. 645–658. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai`.

[TS07] Tatsuhiro Tsuchiya and André Schiper. "Model Checking of Consensus Algorithm". In: *SRDS 2007*. IEEE Computer Society, 2007, pp. 137–148. DOI: `10.1109/SRDS.2007.20`. URL: `https://doi.org/10.1109/SRDS.2007.20`.

[TS08] Tatsuhiro Tsuchiya and André Schiper. "Using Bounded Model Checking to Verify Consensus Algorithms". In: *DISC 2008*. Ed. by Gadi Taubenfeld. Vol. 5218. LNCS. Springer, 2008, pp. 466–480. DOI: `10.1007/978-3-540-87779-0_32`. URL: `https://doi.org/10.1007/978-3-540-87779-0_32`.

[VC02] Paulo Veríssimo and Antonio Casimiro. "The Timely Computing Base Model and Architecture". In: *IEEE Trans. Computers* 51.8 (2002), pp. 916–930. DOI: `10.1109/TC.2002.1024739`. URL: `https://doi.org/10.1109/TC.2002.1024739`.

[VCF00] Paulo Veríssimo, Antonio Casimiro, and Christof Fetzer. "The timely computing base: Timely actions in the presence of uncertain timeliness". In: *DSN 2000*. IEEE Computer Society, 2000, pp. 533–542. DOI: `10.1109/ICDSN.2000.857587`. URL: `https://doi.org/10.1109/ICDSN.2000.857587`.

[Ver+09] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary". In: *SRDS 2009*. IEEE Computer Society, 2009, pp. 135–144. DOI: `10.1109/SRDS.2009.36`. URL: `https://doi.org/10.1109/SRDS.2009.36`.

[Ver+10] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. "EBAWA: Efficient Byzantine Agreement for Wide-Area Networks". In: *HASE 2010*. IEEE Computer Society, 2010, pp. 10–19. DOI: `10.1109/HASE.2010.19`. URL: `https://doi.org/10.1109/HASE.2010.19`.

[Ver+13] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. "Efficient Byzantine Fault-Tolerance". In: *IEEE Trans. Computers* 62.1 (2013), pp. 16–30. DOI: `10.1109/TC.2011.221`. URL: `http://doi.ieeecomputersociety.org/10.1109/TC.2011.221`.

[Ver03] Paulo Veríssimo. "Uncertainty and Predictability: Can They Be Reconciled?" In: *FDDC 2003*. Ed. by André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao. Vol. 2584. LNCS. Springer, 2003, pp. 108–113. DOI: `10.1007/3-540-37795-6_22`. URL: `https://doi.org/10.1007/3-540-37795-6_22`.

[Ver06] Paulo Veríssimo. "Travelling through wormholes: a new look at distributed systems models". In: *SIGACT News* 37.1 (2006), pp. 66–81. DOI: `10.1145/1122480.1122497`. URL: `http://doi.acm.org/10.1145/1122480.1122497`.

[Ver10] Guiliana Santos Veronese. "Intrusion Tolerance in Large Scale Networks". PhD thesis. Universidade de Lisboa, 2010.

[VRC97] Paulo Veríssimo, Luís E. T. Rodrigues, and António Casimiro. "CesiumSpray: a Precise and Accurate Global Time Service for Large-scale Systems". In: *Real-Time Systems* 12.3 (1997), pp. 243–294. DOI: `10.1023/A:1007949113722`. URL: `https://doi.org/10.1023/A:1007949113722`.

[Vuk10] Marko Vukolic. "The Origin of Quorum Systems". In: *Bulletin of the EATCS* 101 (2010), pp. 125–147. URL: `http://albcom.lsi.upc.edu/ojs/index.php/beatcs/article/view/5`.

[Wil+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *PLDI 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 357–368. DOI: `10.1145/2737924.2737958`. URL: `http://doi.acm.org/10.1145/2737924.2737958`.

[Woo+11] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant J. Shenoy, and Emmanuel Cecchet. "ZZ and the art of practical BFT execution". In: *EuroSys 2011*. Ed. by Christoph M. Kirsch and Gernot Heiser. ACM, 2011, pp. 123–138. DOI: `10.1145/1966445.1966457`. URL: `http://doi.acm.org/10.1145/1966445.1966457`.

[Woo+16] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. "Planning for change in a formal verification of the raft consensus protocol". In: *CPP 2016*. Ed. by Jeremy Avigad and Adam Chlipala. ACM, 2016, pp. 154–165. DOI: `10.1145/2854065.2854081`. URL: `http://doi.acm.org/10.1145/2854065.2854081`.

[WST17] James R. Wilcox, Ilya Sergey, and Zachary Tatlock. "Programming Language Abstractions for Modularly Verified Distributed Systems". In: *SNAPL 2017*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 19:1–19:12. DOI: `10.4230/LIPIcs.SNAPL.2017.19`. URL: `https://doi.org/10.4230/LIPIcs.SNAPL.2017.19`.

[XBM13] Shiwei Xu, Sergiu Bursuc, and Julian P. Murphy. "New abstractions in applied pi-calculus and automated verification of protected executions". In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 686. URL: `http://eprint.iacr.org/2013/686`.

[You+05] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. *Robbing the bank with a theorem prover*. Tech. rep. UCAM-CL-TR-644. University of Cambridge, Computer Laboratory, 2005. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-644.pdf`.

[You+07] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross J. Anderson. "Robbing the Bank with a Theorem Prover - (Abstract)". In: *SP 2007*. Ed. by Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe. Vol. 5964. LNCS. Springer, 2007, p. 171. DOI: `10.1007/978-3-642-17773-6_21`. URL: `https://doi.org/10.1007/978-3-642-17773-6_21`.

[ZSR02] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. "COCA: A secure distributed online certification authority". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 329–368. DOI: `10.1145/571637.571638`. URL: `https://doi.org/10.1145/571637.571638`.