Asphalion: Trustworthy Shielding Against Byzantine Faults

Vukotic, Ivana; Rahli, Vincent; Veríssimo, Paulo Jorge Esteves

Asphalion: Trustworthy Shielding Against Byzantine Faults

IVANA VUKOTIC, SnT, University of Luxembourg
VINCENT RAHLI, University of Birmingham
PAULO ESTEVES-VERÍSSIMO, SnT, University of Luxembourg

Byzantine fault-tolerant state-machine replication (BFT-SMR) is a technique for hardening systems to tolerate *arbitrary* faults. Although robust, BFT-SMR protocols are very costly in terms of the number of required replicas ($3f + 1$ to tolerate f faults) and of exchanged messages. However, with "hybrid" architectures, where "normal" components trust some "special" components to provide properties in a trustworthy manner, the cost of using BFT can be dramatically reduced. Unfortunately, even though such *hybridization* techniques decrease the message/time/space complexity of BFT protocols, they also increase their structural complexity.

Therefore, we introduce Asphalion, the first theorem prover-based framework for verifying *implementations* of hybrid systems and protocols. It relies on three novel languages: (1) HyLoE: a Hybrid Logic of Events to reason about hybrid fault models; (2) MoC: a Monadic Component language to implement systems as collections of interacting hybrid components; and (3) LoCK: a sound Logic of events-based Calculus of Knowledge to reason about both 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, Asphalion 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. As a case study, we have verified crucial safety properties (e.g., agreement) of several implementations of hybrid protocols.

CCS Concepts: • **Theory of computation → Logic and verification**.

Additional Key Words and Phrases: Formal verification, Distributed systems, Fault-tolerance, Byzantine faults, Hybrid protocols, MinBFT, Compositional reasoning, Coq, Knowledge calculus, Monad, Step-indexing

ACM Reference Format:
Ivana Vukotic, Vincent Rahli, and Paulo Esteves-Veríssimo. 2019. Asphalion: Trustworthy Shielding Against Byzantine Faults. In *Proceedings of (OOPSLA'19).* ACM, New York, NY, USA, 32 pages.

1 INTRODUCTION

Our society strongly depends on critical information infrastructures such as electrical grids, autonomous vehicles, distributed public ledgers, etc. Unfortunately, proving 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), ensuring their correct behavior becomes even more necessary. Ideally, we should ensure their correctness, relying on a minimal trusted computing base, and to the highest standards possible, e.g., using theorem provers. However, because state-of-the-art verification tools (such as theorem provers) cannot yet tackle complex production infrastructures, bugs and attacks are bound to happen in partially verified systems [Fonseca et al. 2017].

One standard technique to mitigate this problem is to use Byzantine fault-tolerant state machine replication (BFT-SMR) [Bessani et al. 2014; Castro and Liskov 1999b; Lamport et al. 1982] in addition to cheaper certification techniques. It enables correct functioning of a system even when some parts of the system are not working correctly,[1] by masking the behavior of faulty replicas behind the behavior of enough healthy replicas. Unfortunately, because these protocols are rather complex, usually come without a formal specification, and sometimes even without an implementation [Dragoi

[1]Processes and messages in transit can be corrupted arbitrarily. However, we assume perfect cryptography, i.e., a process cannot impersonate another process without the two processes being faulty.

et al. 2015], there is a non-negligible chance that they will later be found incorrect [Abraham et al. 2017a]. Adding on top of that the fact that many variants of these protocols are being developed and adopted in critical sectors (e.g., in blockchain technology [Abraham et al. 2017b; Decker et al. 2016; Kokoris-Kogias et al. 2016; Luu et al. 2016; Pass and Shi 2017; Sousa et al. 2018]), it is clear that ensuring the correctness of these protocols is extremely important.

Moreover, because traditional BFT-SMR is extremely expensive,[2] "hybrid" architectures [Correia et al. 2005, 2004, 2002; Veríssimo 2006; Veríssimo and Casimiro 2002; Veríssimo et al. 2000] have been getting increasing attention: they allow dramatically cutting the message/time/space complexity mentioned above. For example, when applied to BFT-SMR, hybrid solutions only require $2f + 1$ replicas instead of $3f + 1$, to tolerate f faults. Such hybrid architectures allow the coexistence and interaction of components with largely diverse behavior, e.g., synchronous vs. asynchronous, or crash vs. Byzantine. In such models, "normal" components *trust* "special" components that provide *trustworthy* properties. These *trusted-trustworthy* "special" components are made trustworthy through careful design and by verifying their correctness. Therefore, by relying on stronger assumptions (e.g., synchrony or crash), they can be unconditionally trusted to provide stronger properties about the entire hybrid distributed system, than what would be possible otherwise.

This generic "hybridization" paradigm has been showing great promise for BFT-SMR. Many "hybrid" solutions have been designed to reduce the message/time/space complexity of BFT protocols [Behl et al. 2017; Chun et al. 2007; Correia et al. 2004, 2013; Distler et al. 2016; Kapitza et al. 2012; Levin et al. 2009; Veronese et al. 2010, 2013], by relying on trusted-trustworthy components (e.g., message counters in MinBFT [Veronese et al. 2013]) that cannot be tampered with (they are trusted in the sense that they can only fail by crashing, and otherwise always deliver correct results). An increasing number of off-the-shelf hardware systems are now providing trusted environments [Eldefrawy et al. 2017; SecureBlue 2019; SGX 2019; TrustZone 2019], thereby enabling the further development and large-scale use of hybrid protocols.

Anticipating the impact and widespread use of such systems, and to support the development of correct hybrid systems, we present Asphalion,[3] the first theorem prover-based framework that can guarantee the correctness of implementations of hybrid fault tolerant distributed systems communicating via message passing. Asphalion is inspired by Velisarios [Rahli et al. 2018], a framework for verifying the correctness of homogeneous BFT protocols (see Sec. 2 for a comparison). As opposed to Velisarios, Asphalion allows reasoning about hybrid systems by modeling replicas as collections of multiple components that can have different failure assumptions, e.g., some can fail arbitrarily, while others can only crash on failure.[4] In addition, Asphalion allows modular reasoning by lifting properties proved about sub-components of a local system to the level of that local system (see Sec. 5.4). As part of Asphalion, we developed LoCK: a sound knowledge calculus to reason about both homogeneous and hybrid systems, at a high level of abstraction. LoCK enables lifting properties proved about (trusted) sub-components to the level of a distributed system (see Sec. 6.7). As for any such abstract language, a benefit of using LoCK is also that it allows reusing proofs of high-level properties for multiple implementations. As a case study, we verified, among other things, critical safety properties (e.g., agreement) of several versions of the seminal MinBFT hybrid protocol [Veronese et al. 2013],[5] and managed to simplify some of the original proofs of those properties [Veronese 2010] (see Sec. 7). Verifying MinBFT-like protocols is important because:

[2]Seminal BFT protocols such as [Castro 2001; Castro and Liskov 1999a,b] are expensive in terms of the messages exchanged, and the required number of replicas, which in addition have to be diverse enough to enforce independence of failures.
[3]Asphalion was one of king Menelaus' squires, and is associated with *trustworthiness*.
[4]We focus here on the different failure assumptions aspect (crash vs. Byzantine) and leave the different system assumptions aspect (synchronous vs. asynchronous) for future work.
[5]MinBFT [Veronese et al. 2013] is part of the Hyperledger Fabric umbrella [Hyperledger 2019].

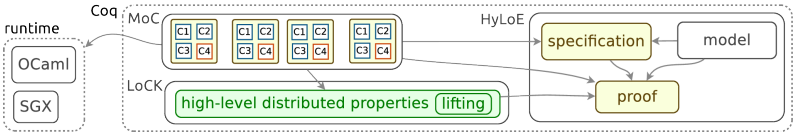


Fig. 1. Overview of Asphalion

(1) MinBFT is part of other protocols, such as [Distler et al. 2016; Kapitza et al. 2012]; (2) many protocols such as [Behl et al. 2017; Kapitza et al. 2012; Veronese et al. 2010, 2013] rely on the same kind of trusted components as MinBFT; and (3) to the best of our knowledge MinBFT's trusted components (called USIGs) have the smallest trusted computing base (TCB) compared to other trusted components used in contemporary hybrid protocols.

Contributions. To summarize, our contributions are as follows: (1) We introduce Asphalion, a generic and extensible Coq-based [Bertot and Casteran 2004; Coq 2019] framework for verifying implementations of hybrid fault tolerant distributed systems communicating via message passing. (2) As part of Asphalion, we developed a Hybrid Logic of Events (Sec. 4) to reason about programs composed of multiple components that can have different failure assumptions (Sec. 5). (3) We developed LoCK, a sound knowledge calculus to reason about hybrid systems at a high-level of abstraction (Sec. 6). (4) We proved several reasoning patterns within LoCK, which we used to prove properties of both homogeneous and hybrid systems. (5) We developed methods to *lift* properties of (trusted) sub-components of a local system to the level of that local system (Sec. 5.4), and to further lift those properties to the level of a distributed system (Sec. 6.7). (6) We implemented the normal case operation of two versions of the seminal MinBFT protocol: one based on USIGs (as in the original version) and one based on TrIncs [Levin et al. 2009] (Sec. 7.1). (7) We proved critical safety properties, such as agreement, of these versions of MinBFT, and simplified some of the original pen-and-paper proofs (Sec. 7.2). (8) We implemented a runtime environment to execute OCaml code extracted from Coq, such that trusted components run inside Intel SGX enclaves (Sec. 8).

2 OVERVIEW

Before diving into the details of our framework in Sec. 4, 5, and 6, we provide here a high-level overview of Asphalion (available at: https://github.com/vrahli/Asphalion/tree/v1.0). In addition, Sec. 3 illustrates how it can be used to verify the correctness of fault-tolerant distributed systems.

2.1 High-Level Architecture of Asphalion

Fig. 1 depicts Asphalion's architecture, where the yellow parts must be provided by the user, while the green parts are optional but convenient to use as we explain below. One starts by *implementing* a distributed system *Sys* within MoC, our component-based programming language shallowly embedded into Coq, where components interact through a monad.[6] A distributed system is a collection of local sub-systems, which are themselves collections of trusted/non-trusted components. Fig. 1 depicts a system composed of 4 local sub-systems, each composed of 3 non-trusted blue components and a trusted one in orange. Then, one provides a specification *Spec* (e.g. agreement) for *Sys* within our model of distributed systems HyLoE—a hybrid logic of events based on Lamport's *happened before relation* [Lamport 1978] (one of the two main models of distributed systems, along with *distributed snapshots* [Chandy and Lamport 1985]). Finally, one proves that *Sys* satisfies *Spec* within HyLoE by proving that *Spec* holds for all possible runs of *Sys* (see Sec. 4). This can be done:

[6]See the file called `model/ComponentSM.v` in our implementation for a definition of MoC, as well as the two files called `model/ComponentSMExample1.v` and `model/ComponentSMExample2.v` for examples.

(1) using the general high-level distributed properties proved within our knowledge calculus LoCK (as discussed in Sec. 2.2, knowledge calculi provide convenient abstraction layers to reason about distributed systems without concern for low-level details), and (2) by directly proving the properties specific to *Sys* using the automation provided by Asphalion in the form of Coq tactics.

One can then generate executable OCaml code from the distributed system *Sys* implemented in MoC, using Coq's extraction mechanism. In addition, Asphalion provides support to execute trusted components (the orange C4 components in the case of *Sys*) within Intel SGX enclaves.[7] Note that MoC implementations are Coq programs that can be as abstract or concrete as one wants. For example, one could choose to abstract away some data structures using parameters. However, these data structures ultimately need to be instantiated in order to extract executable OCaml code.

2.2 High-Level Reasoning

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 Sec. 5.4 (local lifting) and Sec. 6.7 (distributed lifting).

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 Sec. 6.1); and (2) by proving that the assumptions made within the abstract model are satisfied by the concrete implementations (see Sec. 6.6 and Sec. 7.2 for examples of such assumptions). The high-level results we present here (such as the lifting property presented in Sec. 6.7) can be instantiated for many implementations of hybrid systems. We already used those results to prove the safety of the Micro system discussed in Sec. 3, as well as two versions of MinBFT that rely on two different trusted components (see Sec. 7).

We chose to rely on a knowledge calculus because such calculi provide a convenient way to reason about distributed systems at a high-level of abstraction, as it has been demonstrated in the extensive literature on the subject. Many knowledge based systems have been developed to, e.g.: analyze distributed systems [Dwork and Moses 1990; Fagin et al. 1997; Halpern 1987; Halpern and Moses 1990; Panangaden and Taylor 1992]; reason about synchronous systems [Ben-Zvi 2011; Ben-Zvi and Moses 2014; Castañeda et al. 2014, 2016; Dan et al. 2017; Goren and Moses 2018]; derive protocols [Halpern and Zuck 1992]; synthesize systems [Bickford et al. 2004]; and reason about blockchain protocols [Halpern and Pass 2017]. However, as opposed to "standard" knowledge theories that consider an external and logical notion of knowledge (that cannot necessarily be computed), Asphalion relies on a syntactic and explicit representation of knowledge [Fagin et al. 2003], which is more pragmatic and computational, in the sense that pieces of knowledge are concrete pieces of data stored locally and exchanged through messages (allowing processes to gain knowledge [Chandy and Misra 1986; Halpern 1987]).

2.3 Rationale for Designing Asphalion

As it turns out, Asphalion is not an extension of Velisarios, but is inspired by and uses part of it. Starting from the foundations of Velisarios (its logic of events), we designed an entirely new

[7] We explain how to obtain running code such that trusted components are executed inside Intel SGX enclaves, in the file called `MinBFT/runtime_w_sgx/README.md` in our implementation.

framework in order to handle hybrid systems, and reason about such systems in a principled way (Sec. 8 describes our proof effort). Let us now elaborate on the four main reasons that led us to design a new framework and not simply extend Velisarios.

(1) Velisarios does not provide full support for compositional programming and reasoning in the sense that, there, a local system is a single component. To add axioms about trusted components to it, we would first need the notion of interacting components, which is why we developed MoC (see Sec. 5). In MoC distributed systems are implemented as collections of local systems, which are themselves collections of components, some of them being marked 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 (see Sec. 5.4).

(2) Moreover, to capture the behavior of these trusted components, we had to modify Velisarios's logic of events, to allow non-trusted components to misbehave, while the trusted ones keep following their specifications. We captured this by changing the semantics of events (i.e., the trigger function described in Sec. 4.3) to also handle events where trusted components of compromised nodes are called (see Sec. 4 for details on events and their semantics in Velisarios and Asphalion). This led us to developing the HyLoE logic described in Sec. 4.[8]

(3) Inspired by Velisarios's knowledge library, we equipped Asphalion with LoCK, a sound (hybrid) knowledge sequent calculus, which differs and goes well beyond Velisarios's library. First of all, as opposed to Velisarios's knowledge library (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) and principles (as derivation rules) of the theory. Moreover, LoCK enforces an abstraction barrier (being deeply embedded in Coq), which does not exist in Velisarios's library. Also, LoCK allows reasoning at a high-level of abstraction about trusted and non-trusted knowledge, while Velisarios's library does not distinguish between trusted and non-trusted knowledge. Other advantages of LoCK that we plan to explore in the future are that: such a sequent calculus opens the door to some automation; and while its semantics is currently expressed in terms of HyLoE, other backends could be used.

(4) We developed, within LoCK, a general technique to *lift* properties of trusted components to the global level of an entire distributed system. A great advantage of such high-level results is that they are abstract and can be reused for several implementations. Moreover, the result we proved in Sec. 6.7 captures a key aspect of the logic of hybrid systems.

2.4 Benefits and Limitations

As hinted at above, in addition to reasoning about hybrid systems,[9] using Asphalion one can also reason about homogeneous BFT systems by not using trusted components, and about crash fault tolerant systems by assuming that there are no Byzantine events (see Sec. 4.2). Moreover, as explained in this paper, and as illustrated in Sec. 3 and 7, Asphalion supports verifying safety properties of such systems, while providing support for liveness is left for future work. Asphalion's support comes in the form of three novel languages. (1) MoC, presented in Sec. 5, is a programming language shallowly embedded in Coq. In order to automatically derive properties of components, Asphalion allows defining deep embeddings of sub-languages (for which the desired properties hold) that are interpreted to MoC expressions. We so far provide two such deep embeddings, which are prototypical, and which we expect will be reusable for other protocols. In case additional features that are not supported by these two embeddings are required, one can simply implement additional deep embeddings following the two examples we provide. (2) HyLoE, presented in Sec. 4, is a logic

[8]Note that Asphalion reuses only these logical foundations of Velisarios's foundations, i.e., part of its logic of events.
[9]To the best of our knowledge, Asphalion is the only framework that supports reasoning about hybrid systems.

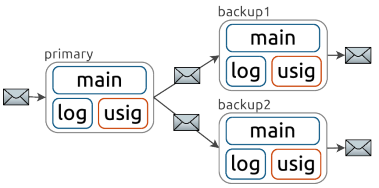
of events shallowly embedded in Coq (i.e., one must use Coq's logic to state and derive properties from HyLoE's axioms). Therefore, when specifying and proving properties of distributed systems in Asphalion, one is constrained by: (a) the expressiveness of HyLoE's operators, (b) HyLoE's axioms, and (c) Coq's logic. Finally, (3) LoCK, presented in Sec. 6, is a knowledge calculus deeply embedded in Coq, whose expressiveness is constrained by its inference rules. We leave studying LoCK's proof-theoretic strength for future work. LoCK is optional but recommended because: (a) it allows stating system properties at a high-level of abstraction, without concern for how knowledge is computed (it is more abstract and less verbose than HyLoE); and (b) it allows reusing those properties to prove the correctness of multiple protocols.

2.5 Notation

Before illustrating how Asphalion works through a simple example in Sec. 3, let us finish here by presenting some notation used throughout the paper. The type $A \longrightarrow B$ 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: $\texttt{let}\ x, y = p\ \texttt{in}\ f$. We write $p.\mathbf{1}$ and $p.\mathbf{2}$ 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).

3 RUNNING EXAMPLE

Let us now explain the workflow in Asphalion, by going through the simple example depicted on the right, which we refer to as Micro (a simplified version of MinBFT), and which we use throughout the paper. We start by implementing Micro within MoC. Next, we specify its agreement property within HyLoE. Finally, we verify this property primarily using LoCK.



Micro's implementation in MoC. Micro is composed of three nodes, i.e. three local sub-systems: a primary called primary, and two backups called backup1 and backup2. More precisely, let the Micro distributed system be a function that, for every node name $a \in \{\text{primary}, \text{backup1}, \text{backup2}\}$, returns a local sub-system (a's code). Each local sub-system is composed of three components (state machines), namely, a main component called main and two sub-components: a message log called log; and a trusted message counter, called usig, similar to the one used in MinBFT (Sec. 7.1).

Each node's main component is in charge of receiving messages; calling the log and usig sub-components to handle messages appropriately as discussed below; and finally possibly sending further messages. A message is either of the form: (1) request(r)—sent from clients to the primary; or (2) commit(r, ui)—sent from the primary to the backups; or (3) accept(r, i)—sent from the backups to themselves. The log components receive inputs of the form log(c) (to log commits) and produce outputs of the form logged; while usig components receive inputs of the form createUI(r) or verifyUI(r, ui) and produce outputs of the form createdUI(ui), goodUI, or badUI.

On every input request(r), the primary (its main component) first calls its trusted usig component to assign a unique trusted sequence number i to the request r (i.e., the usig component increments its local counter and signs r along with the new counter value i using a confidential key). It then stores the signed request in its log. Finally, it broadcasts commit(r, $\langle i, \vartheta \rangle$) to both backups, where ϑ is the usig-generated signature of the pair $\langle r, i \rangle$. The pair $\langle i, \vartheta \rangle$ is called a UI as it allows Uniquely Identifying the request r in a reliable manner (thanks to the signature). Upon receipt of such a message c = commit(r, $\langle i, \vartheta \rangle$) from the primary, each backup b (its main component) first

checks whether c has a *valid* trusted sequence number i, i.e., the signature ϑ is correct and whether $i = j + 1$, where j is the highest sequence number received so far by b from the primary. If c is valid, then b stores it in its log, and sends a message to acknowledge the fact that c has been accepted.

Each main component maintains a state composed of: (1) the service state (a number, such as the balance of a bank account for example), which is updated every time a request is executed; and (2) the highest sequence number received from the primary (this is only used by backups). The initial state of the main component of each node a is simply the pair $\langle 0, 0 \rangle$, and its update function is depicted above on the right. Given a state s and an input message m, main pattern matches on m, and runs the appropriate handler. Note the $\mathbb{I}(\_)$ operator. Let us explain what it does. As discussed in Sec. 5.4, the three handlers are expressed in a deep embedding of a simple language, which is more amenable to automation than our general monadic programming language shallowly embedded in Coq (and therefore rather unwieldy).[10] $\mathbb{I}(\_)$ lifts processes from the deep embedding to the general shallow embedding. This simple deep embedding provides three constructors, namely: RET(_) to create a process out of a Coq term, BIND(_, _) to compose processes (sometimes written as _ BIND _), and CALL(_, _) to call sub-processes.

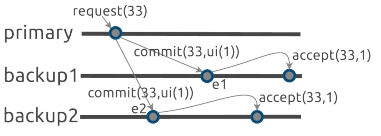
$$\lambda s, m. \mathbb{I}\left(\begin{array}{l}\texttt{match } m \texttt{ with}\\ \mid \texttt{request}(\_) \Rightarrow \texttt{handleRequest}(a, s, m)\\ \mid \texttt{commit}(\_, \_) \Rightarrow \texttt{handleCommit}(a, s, m)\\ \mid \texttt{accept}(\_, \_) \Rightarrow \texttt{handleAccept}(a, s, m)\end{array}\right)$$

Let us now define handleCommit, which handles commits sent by the primary to the backups—we elude some details for readability. The other handlers and components are defined in a similar fashion, and are therefore omitted here (see `MinBFT/MicroBFT.v` for more details). A commit message c contains a request value and a UI, which we access using c.val and c.ui, respectively. The validCommit function checks that: c was sent by the primary, a is a backup, and a received the counter values less than the one in c.ui (this information is stored in s). If c is invalid, handleCommit returns RET(s, []), meaning that main's state remains the same (i.e., s), and it does not output any message ([] is the empty list). If c is valid, main verifies the validity of c.ui by calling its usig sub-component using CALL. If c.ui is valid, main updates its state using update, which computes the highest counter between the one in c.ui (i.e., c.ui.counter) and the one recorded so far in s. Finally, it logs the commit by calling its log sub-component, and returns its updated state s' and an accept message, which is meant to be sent to itself.

```
def handleCommit(a, s, c)
= if ¬validCommit(a, s, c) then RET(s, []) else
   CALL(usig, verifyUI(c.val, c.ui)) BIND λo.
    match o with
    | goodUI ⇒
        let s' = update(c, s) in
        CALL(log, log(c)) BIND λ_.
        RET(s', [accept(c.val, c.ui.counter)])
    | _ ⇒ RET(s, [])
```

Micro's specification using HyLoE. We then specify Micro's agreement property within HyLoE (our hybrid logic of events shallowly embedded in Coq). It states that if the backups accept two requests r_1 and r_2, both with sequence number i, then $r_1 = r_2$. The formula on the left formally states this property (we omit some details for readability—see `MinBFT/MicroBFTagreement.v` for more details), while the diagram on the right depicts a simple run of Micro:

```
Lemma micro_agreement :
  ∀(eo : EO)(e1, e2 : Event(eo))(r1, r2 : Request)(i : ℕ).
    accept(r1, i) ∈ Micro ⇝ e1
    → accept(r2, i) ∈ Micro ⇝ e2
    → r1 = r2
```



This property is stated directly in Coq (using Coq's logical constructors), and involves HyLoE constructs. The type EO is the type of event orderings, which are abstract representations of system runs (e.g., as depicted on the right above), and which are discussed further in Sec. 4.3. Event(eo)

[10] The monad of this general language takes care of threading the sub-components that a local system's components are allowed to use/call throughout the execution of that system.

is the type of events happening within the event ordering eo.[11] We simply write Event when the corresponding event ordering is clear from the context. In micro_agreement, the events e_1 and e_2 are therefore events happening within the event ordering eo, i.e., during the run of the system captured by eo. Therefore, this property states that *in each possible run of* Micro, if it outputs two messages of the form accept(r_1, i) and accept(r_2, i) at e_1 and e_2, respectively, where i is the trusted sequence number associated with both r_1 and r_2, then it must be that $r_1 = r_2$.

HyLoE is essentially the definition of event orderings, along with the axioms that govern them (see Sec. 4). As discussed in Sec. 5.2, on top of that, Asphalion provides constructs to reason about the behavior of processes at given events, thereby allowing one to reason about runs of MoC systems. In particular, it provides three constructs to reason about: (1) the inputs of processes at given events; (2) the states of processes before and after given events; and (3) the outputs of processes at given events. For example, in micro_agreement, accept(r_1, i) $\in$ Micro $\rightsquigarrow$ e_1 states that accept(r_1, i) belongs to Micro's outputs at e_1.

As explained in Sec. 4, one feature of HyLoE is that it allows reasoning about the behavior of trusted components running at compromised nodes. Here, it allows reasoning about usig components even if the main and log components have been compromised. In general, to prove a system property, one has to prove that it holds for all event orderings, even those where events happen at nodes where only the trusted components are running correctly. As it turns out, micro_agreement holds even for the runs where the primary, except for its usig component, has been compromised.

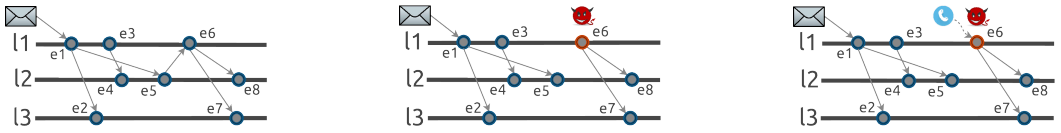
Micro's verification using LoCK. One could prove Micro's agreement property using only HyLoE, i.e., using only its axioms and properties of the above mentioned constructs to reason about systems' inputs, states and outputs. Instead, we recommend to use LoCK for two main reasons. (1) As mentioned above, one advantage of using LoCK is that it allows one to *reuse* the results proved there for several protocols. (2) Moreover, LoCK is a convenient language to reason about systems because it is more *abstract* and less verbose than HyLoE, as LoCK expressions do not mention events and event orderings. Note that even though expressions do not mention events, sequents do, and LoCK provides a highly convenient way to navigate through events, through what we call *guards* (see Sec. 6.4). Let us provide an example. LoCK is a sequent calculus, where a sequent is of form $\langle G \rangle\ H \vdash \sigma$, where G is a list of guards, H is a list of hypotheses, and σ is the conclusion. In the following sequent (LoCK's syntax and semantics are presented in See 6.2 and 6.3):

$$\langle y : e_1 \prec e_2 \rangle\ x_1 : \mathcal{K}^+(d_1)\ @\ e_1, x_2 : \mathcal{K}^+(d_2)\ @\ e_2 \vdash \sigma$$

the expressions $\mathcal{K}^+(d_1)$ (i.e., *we know* d_1) and $\mathcal{K}^+(d_2)$ (i.e., *we know* d_2) are event-free. The x_1 hypothesis states that $\mathcal{K}^+(d_1)$ holds at some event e_1, and similarly for x_2, while the y guard states that e_1 happened before e_2. Through guards, one can then conveniently relate the knowledge available at different points in space/time in a system run (which is captured by the hypothesis list).

Now, back to Micro, we derived micro_agreement (see Sec. 6.8 for further details) using Thm. 6.1, a general abstract lemma proved within LoCK (i.e., using LoCK's inference rules). As it turns out, Thm. 6.1 captures part of the logic used by hybrid systems, and can be reused for several such systems (we show two other examples in this paper: USIG- and TrInc-based versions of MinBFT). Essentially, Thm. 6.1 allows one to derive that if two nodes know two pieces of information for which the same trusted sequence number has been generated, then those pieces of information must be the same. It relies on a number of protocol-dependent assumptions, described in Sec. 6.6, regarding, for example, the way knowledge gets propagated, and the way trusted sequence numbers are maintained. Because we have proved LoCK's soundness, i.e., its inference rules are valid w.r.t. its HyLoE semantics, once we have proved a lemma within LoCK, we can immediately extract its HyLoE interpretation.

[11] The event ordering depicted on the right above is composed of 5 events: one triggered by the receipt of a request by the primary; two triggered by the receipt of commits by the backups; and two triggered by the receipt of accepts by the backups.



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

Fig. 2. Examples of message sequence diagrams

We use this to prove micro_agreement, which is expressed in HyLoE, i.e., we instantiate Thm. 6.1 appropriately, and compute its HyLoE interpretation. It then remains to prove, within HyLoE, that the corresponding instances of its protocol-dependent assumptions hold about Micro (see Sec. 5.3 for an example of such an HyLoE proof). One interesting fact about these properties is that they are not proved by induction, as the inductive reasoning is all done within LoCK. For example, one of those assumptions, called **KLD** in Sec. 6.6, is: $\forall_{\mathbf{t}} \lambda t. \mathcal{K}^{+}(t) \rightarrow (\mathcal{K}^{-}(t) \vee \mathcal{L}(t) \vee \mathcal{OD}(t))$. Intuitively, it states that if we know a trusted piece of information then either (1) we already knew about it; or (2) we just learned about it; or (3) we came up with this piece of information (and disseminated it). This is straightforwardly true about Micro because backups get to know about UIs by learning about them from the primary. As it turns out, the protocol-dependent assumptions that Thm. 6.1 relies on, are all straightforward to prove, allowing us to straightforwardly derive micro_agreement.

4 HYLOE: A HYBRID LOGIC OF EVENTS

We now present HyLoE, a new hybrid variant of the Logic of Events (LoE) that was originally introduced in [Bickford 2009] to reason about crash fault tolerant protocols [Bickford et al. 2012; Rahli et al. 2017; Schiper et al. 2014], and later used to reason about cyber-physical systems [Anand and Knepper 2015]. LoE was then extended in [Rahli et al. 2018] to handle BFT systems. We now extend LoE further to enable reasoning about hybrid fault models and hybrid protocols (which contain components with different failure assumptions—some can be compromised, while others can only crash on failure), and explain the main differences with previous versions. First, we start by introducing basic concepts such as names, messages, etc., which we use to define HyLoE.

4.1 Basic HyLoE Concepts

To model the behavior of a distributed protocol one has to reason about its nodes (also called processes, locations, or local sub-systems), and the messages they exchange. In order to make our model as general as possible, these concepts are introduced as parameters of HyLoE, and have to be instantiated for each implemented protocol. One of HyLoE's parameters is a type Node of node names, ranged over by a. Because nodes communicate via message passing, another parameter is Msg, a type of messages ranged over by msg. The nodes of a system receive messages and produce *directed messages*, which are pairs of a message and a list of destinations denoting the locations to which the message has to be delivered. In Asphalion, nodes are collections of components, some of which are trusted (i.e., they cannot be compromised—see Sec. 5). We assume that those trusted components only receive inputs of some abstract type InputTrusted, ranged over by it.

4.2 Accounting for Trusted Components in HyLoE Through Hybrid Events

HyLoE is a logic of events to model hybrid fault tolerant distributed systems. One of the most fundamental concepts to reason about distributed systems in LoE, is the concept of an *event*, which can be seen as a point in space/time [Lamport 1978] at which something happened. In EventML [Bickford et al. 2012; Rahli et al. 2017; Schiper et al. 2014] events are abstract objects that only correspond to the handling of a message by a node that follows its specification (kind 1—see

Types: Event (ranged over by e) AuthData (ranged over by $auth$) Keys (ranged over by ks)

Functions: $\prec \in \text{Event} \to \text{Event} \to \mathbb{P}$ $\quad \text{trigger} \in \text{Event} \to \text{TriggerInfo}$ $\quad \text{keys} \in \text{Event} \to \text{Keys}$
$\text{loc} \in \text{Event} \to \text{Node}$ $\quad \text{pred} \in \text{Event} \to \text{option(Event)}$ $\quad \text{nfo2auth} \in \text{TriggerInfo} \to \text{list(AuthData)}$

Axioms:
(1) $\prec$ is transitive and well-founded
(2) Equality on events is decidable
(3) $\forall e_1, e_2.\text{pred}(e_1) = \text{Some}(e_2) \to \text{loc}(e_1) = \text{loc}(e_2)$
(4) $\forall e_1, e_2.\text{pred}(e_1) = \text{Some}(e_2) \to e_2 \prec e_1$
(5) $\forall e_1, e_2.\text{pred}(e_1) = \text{None} \to \text{loc}(e_1) = \text{loc}(e_2) \to e_1 \neq e_2 \to e_1 \prec e_2$
(6) $\forall e_1, e_2.\text{pred}(e_1) = \text{pred}(e_2) \to \text{loc}(e_1) = \text{loc}(e_2) \to e_1 = e_2$
(7) $\forall e, e_1, e_2.e_1 \prec e_2 \to \text{loc}(e_1) = \text{loc}(e_2) \to \text{pred}(e_2) = \text{Some}(e) \to e = e_1 \vee e_1 \prec e$

Fig. 3. HyLoE parameters

Fig. 2a). As opposed to EventML, in Velisarios [Rahli et al. 2018], an event is either of kind 1, or it corresponds to some arbitrary behavior, in which case no further information regarding this event is available/provided (kind 2—see Fig. 2b). HyLoE further extends LoE by providing means to reason about three kinds of events. As in EventML and Velisarios, Asphalion supports events of kind 1 (see the constructor `TImsg` below). Furthermore, the kind 2 events of Velisarios, 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 Fig. 2b and `TIarbitrary` below); and (2) those that called a trusted component (kind 2b—see Fig. 2c and `TItrust` below). Correspondingly, we introduce the type (msg and it are introduced in Sec. 4.1):

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

4.3 Hybrid Event Orderings

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 LoE, 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* (*event orderings* formalize the *message sequence diagrams* used by system designers to describe the behavior of systems). 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).[12]

Fig. 2 provides examples of message sequence diagrams. Fig. 2a, 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 Fig.2b, e_6 is a Byzantine event for which no information is available and at which no trusted component was called; and in Fig. 2c, e_6 is a hybrid event at a Byzantine location and at which a trusted component was called.

Formally, an event ordering eo of type EO is a record (see Fig. 3) that consists of a set of abstract events Event ordered by a well-founded and transitive causal ordering relation $\prec$ (see Axiom (1)).[13] The function loc returns the location where each event e happens, and trigger explains why it happened by associating an element of TriggerInfo with e. Events are totally ordered at a given location: pred(e) returns e's local direct predecessor, if it exists. As in Velisarios [Rahli et al. 2018, Sec.3.3], our model relies on an abstract concept of keys (of type Keys) to implement and reason about authenticated communication. Even though for the purpose of this paper the type AuthData,

[12]Note that event orderings are used to model systems and prove properties about them, and cannot be accessed by the systems themselves, i.e., faulty nodes identified in the model through TriggerInfo, are not identified by programs.
[13]Our model is based on Lamport's *happened before* relation [Lamport 1978], as opposed to the "global state" semantics [Chandy and Lamport 1985].

of authenticated pieces of data, is left abstract, let us mention that an authenticated piece of data (e.g., an authenticated message) can be seen as the pair of a piece of data and an authentication token (also an abstract entity, that can be instantiated for example with RSA signatures) that has been generated using keys (that can be instantiated for example with RSA keys). Keys are associated with nodes as follows: keys(e) returns the keys available at e. Finally, nfo2auth(nfo) lists all the authenticated pieces of data included in nfo. Axioms (3) to (7) provide an axiomatization of pred. For example, Axiom (4) says that if e_2 is e_1's direct predecessor, then e_2 happened before e_1; and Axiom (5) says that if e_1 has no direct predecessor and e_2 happened at the same location as e_1, then e_2 happened after e_1 if e_2 is not e_1 (e_1 is the initial event at that location). Thanks to these axioms, one can see an event ordering as a collection of local traces, where a local trace is a collection of events happening at the same location and ordered in time (through pred), and such that some events of different local traces are causally ordered (through $\prec$). Typically, some runs/event orderings are not possible and therefore excluded through assumptions in specifications (e.g., for fault-tolerant systems, we typically exclude event orderings with more than f faulty nodes).

HyLoE Notation. Even though some operators are parameterized by event orderings, we often omit those for readability. We now define some useful notation. 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; let $e_1 \preceq e_2$ be ($e_1 \prec e_2 \vee e_1 = e_2$); let $e_1 \sqsubset e_2$ be $e_1 \prec e_2 \wedge$ loc(e_1) = loc(e_2); and let $e_1 \sqsubseteq e_2$ be $e_1 \preceq e_2 \wedge$ loc(e_1) = loc(e_2).

5 MOC: COMPONENT-BASED PROGRAMMING

Asphalion enables reasoning about distributed systems, where local sub-systems are composed of multiple components that can have different failure assumptions. Components are referred to by their names. Let CompName be the set of component names, ranged over by cn. A component name includes a tag (a Boolean) describing whether the component is trusted (trusted components are constrained to only react to inputs of type InputTrusted—see Sec. 4.1). 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 $\mathcal{O}(cn)$. Sec. 5.1 introduces components and explains how they interact through a monad. It then explains how to build local/distributed systems as collections of components. Sec. 5.2 explains how to relate the execution of systems with event orderings. Finally, Sec. 5.4 explains how to reason about systems compositionally by lifting properties of sub-components of a local system to the level of that system.

5.1 Components as State Machines, and Local and Distributed Systems

Components. A component is a named state machine, which essentially consists of an update function and the current state of the machine. To allow components calling each other, we define state machines using a state monad [Moggi 1989]. 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 use step indexing [Dreyer et al. 2011] to define them, requiring that machines at level n can only use machines of lower levels. Let 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):

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

Fig. 4. An execution of a local system

Going back to state machines, a machine at level $n+1$ (of type 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 $\mathcal{S}(cn)$.[14]

Monad operators. 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 $\mathsf{M}^n(B)$. We also introduce a call operator to call other components from within a component at level $n+1$. It 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 sub-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 sub-components that $comp$ can use because of its level). This computation produces an output o and a list of updated sub-components $subs_2$. Finally, call returns the output o, as well as the list of sub-components $subs$, where $subs_1$ is replaced by $subs_2$.[15]

Local & Distributed Systems. A *local system* of type LocalSystem is a pair of a main component at level n and a list of sub-components at lower levels. We enforce that main components send and receive messages. A (distributed) *system* of type System is a function from node names to local systems, i.e., of type Node → LocalSystem (see, e.g., the Micro system presented in Sec. 3).

5.2 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 in Sec. 5.3 showing how to combine these definitions to prove systems' properties in a compositional manner.

Computing systems' states. First, $ls@^{-}e$ runs the local system ls by applying its main component to its sub-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 events locally preceding e, including e). It either (1) returns a local system ls' if all those events have been triggered by information of the form $\mathtt{TImsg}(msg)$, i.e., non-Byzantine events; or (2) it returns a trusted component in case at least one of those events was triggered by some information of the form $\mathtt{TItrust}(it)$ (in case the trusted component[16] is called) or $\mathtt{TIarbitrary}$ (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 system is in; or (3) it is undefined if one of those events is a Byzantine event and ls does not include a trusted component. For example, Fig. 4 shows the status of the components of a local system (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.

[14] State machines also have the ability to halt on their own. However, we do not discuss this feature here for simplicity.
[15] See [Vukotic et al. 2019, Appx.A] for an example of a local system and of how call works.
[16] For simplicity, we currently only support systems with at most one trusted component per local sub-system—the typical case in the literature on hybrid systems. This can easily be extended to systems with multiple trusted components if needed.

As Fig. 4 illustrates, if one of those preceding events is Byzantine, $ls@^{-}e$ keeps 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 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 system 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 $S(\mathsf{loc}(e))@^{-}e\lfloor_{cn}$, which we write as $S@^{-}e\lfloor_{cn}$, and similarly for after the event.

Computing systems' outputs. Let $ls \rightsquigarrow e$ 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, $ls \rightsquigarrow e$ produces instead the outputs of the trusted component, which we are keeping track of (as explained above). We write $S \rightsquigarrow e$ for $S(\mathsf{loc}(e)) \rightsquigarrow e$; and $d \in ls \rightsquigarrow e$ to mean that d occurs within the outputs computed by $ls \rightsquigarrow e$.

As illustrated in Sec. 5.3, Asphalion 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 $S \rightsquigarrow e$.

5.3 Example: a Compositional Proof of a Simple Micro Property

Let us provide an example. As defined in Sec. 3, Micro is a distributed system composed of three local sub-systems, each of which is composed of three components called main, log, and usig. Let us prove prove that if $\texttt{accept}(r, i) \in \mathsf{Micro} \rightsquigarrow e$, i.e., if a backup accepts a request r with sequence number i, then r is logged, i.e., it is in $\mathsf{Micro}@^{+}e\lfloor_{\mathsf{log}}$. First, (1) we prove that whenever log is called, it logs the commit given as input. We prove this about the local system composed of log only (which does not use any sub-components). Then, (2) from $\texttt{accept}(r, i) \in \mathsf{Micro} \rightsquigarrow e$, we obtain that this output, as well as $\mathsf{Micro}@^{+}e$, was produced by running Micro on $\mathsf{Micro}@^{-}e$. We then inspect the code run by Micro, and we see that log, through the use of call, was requested to log a commit containing r. Finally, (3) we compose this proof in step (2) with the one in step (1), and conclude by showing that $\mathsf{Micro}@^{+}e\lfloor_{\mathsf{log}}$ is the new state computed in step (1).

5.4 Lifting Through "Deep" Restrictions

We now describe a compositional method to lift properties proved about (trusted) sub-components of a local system to the level of that system. One advantage of MoC is its expressiveness and flexibility: because MoC is shallowly embedded in Coq, one can make use of any Coq expression to define a component's update function, as long as it has the right type, i.e., $\mathsf{Upd}^{n}(cn)$. However, this is sometimes a disadvantage because it entails that we cannot prove many general lemmas about the behavior of components. For example, nothing prevents components from throwing away their sub-components, even though components often simply use their sub-components, and return them updated. This is useful information, which we want to derive easily. A standard technique to prove such generic results about such "well-behaved" programs is to: (1) define a deep embedding of these "well-behaved" programs; (2) define an "interpretation" function from the deep embedding to the shallow one; and (3) prove that the 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.[17] Namely, let

[17] See [Vukotic et al. 2019, Appx.B] for another example of such a language that also allows spawning new sub-components.

$\mathsf{Proc}(A)$ be the set of terms p of the following form (left), and let $\mathbb{I} \in \mathsf{Proc}(A) \to \mathsf{M}^n(A)$ (for any level n) be the following interpretation of this language (right):

$$\begin{array}{lll} \mathtt{RET}(a) & \text{where} & a \in A \\ \mathtt{BIND}(p_1, p_2) & \text{where} & p_1 \in \mathsf{Proc}(B) \;\&\; p_2 \in B \to \mathsf{Proc}(A) \\ \mathtt{CALL}(cn, i) & \text{where} & i \in \mathcal{I}(cn) \;\&\; \mathcal{O}(cn) = A \end{array} \qquad \begin{array}{lcl} \mathbb{I}(\mathtt{RET}(a)) & = & \mathsf{ret}(a) \\ \mathbb{I}(\mathtt{BIND}(m, f)) & = & \mathbb{I}(m) \gg\!= \lambda x.\mathbb{I}(f(x)) \\ \mathbb{I}(\mathtt{CALL}(cn, i)) & = & \mathsf{call}(cn, i) \end{array}$$

Then, given a component name cn, a level n (indicating what sub-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 sub-components $subs_1$ then it produces sub-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 Sec. 5.1). For example, we proved the following general result,[18] which we in turn used to prove that our MinBFT implementations satisfy the **Mon** property presented in Eq. 4 in Sec. 6.6:[19]

Theorem 5.1 (Local Lifting). *Given a local system 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 need not be at level* 1*, but this constraint in Thm. 5.1 is convenient to obtain a simple lifting theorem. Otherwise, without this constraint, i.e., for higher-level components, this theorem would be more complicated because it would have to also take into account the sub-components such higher-level components rely on. More precisely, it would not be enough to run the sub-system ls' composed of cn and its sub-components $subs$ (the sub-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 sub-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 sub-components $subs$. It is worth noting that all the "standard" trusted components used in the literature [Chun et al. 2007; Levin et al. 2009; Veronese et al. 2013] are level* 1 *components. Therefore, we leave developing local lifting lemmas for higher-level components for future work.*

6 LOCK: A HYBRID KNOWLEDGE CALCULUS

In order for a distributed system to achieve some objective as a whole, its nodes typically need to generate, disseminate, and gather some information. The way they exchange this information forms the high-level *logic* of the system. Understanding and being able to reason about this logic is one of the major difficulties when dealing with distributed systems. Moreover, the same high-level logic is typically shared by many systems. Therefore, we introduce LoCK: a calculus to reason at a high-level of abstraction about the *knowledge* exchanged between the nodes of a distributed system. Although LoCK is inspired by Velisarios's knowledge library, one advantage of LoCK is that it exposes the primitive concepts necessary to reason about knowledge through sound inference rules,[20] which further opens the door to automation.[21] Moreover, unlike in Velisarios, LoCK enables

[18] See the lemma called `M_byz_compose_step_trusted` in the file called `model/ComponentSM3.v` in our implementation.
[19] See `ASSUMPTION_monotonicity_true` in `MinBFT/MinBFTass_mon.v` and `MinBFT/TrIncass_mon.v`.
[20] We proved the soundness of our inference rules using Coq—see the file called `model/CalculusSM.v`.
[21] Automating proofs within LoCK is left for future work. We have started developing proof tactics that automatically apply the adequate introduction and elimination rules, in the spirit of Coq's *destruct* and *intro* tactics. In addition, we would like to develop both simple "brute-force" proof search engines, and decision procedures for fragments of LoCK.

Types:	Data (ranged over by d)	Identifier (ranged over by i)	Trust $\subseteq$ Data (ranged over by t)
Functions:	sys $\in$ System	trustHasId $\in$ Trust $\to$ Identifier $\to \mathbb{P}$	verify $\in$ Event $\to$ AuthData $\to \mathbb{B}$
	mem $\in$ CompName	genFor $\in$ Data $\to$ Trust $\to \mathbb{P}$	trusted2id $\in \mathcal{S}$(trust) $\to$ Identifier
	trust $\in$ CompName	know $\in$ Data $\to \mathcal{S}$(mem) $\to \mathbb{P}$	lt $\in$ Identifier $\to$ Identifier $\to \mathbb{P}$
	owner $\in$ Data $\to$ Node	auth2data $\in$ AuthData $\to$ list(Data)	initId $\in$ Identifier
Axioms:	(1) lt is transitive and anti-reflexive		(3) $\forall t, d_1, d_2$.genFor(d_1, t) $\to$ genFor(d_2, t) $\to$ $d_1 = d_2$
	(2) know(d, m) is decidable		(4) $\neg$know(d, m) for all initial states m of sys's components
	(5) all initial identifiers of sys's trusted components are equal to initId		

Fig. 5. LoCK's parameters

$$\theta \in \text{KType} ::= \texttt{KTi} \mid \texttt{KTn} \mid \texttt{KTd} \mid \texttt{KTt} \qquad v \in \text{KVal} ::= i \mid \boldsymbol{a} \mid d$$

$$\tau \in \text{KExp} ::= \top \mid \bot \mid \tau_1 \to \tau_2 \mid \tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2 \mid \exists \phi \mid \forall \phi \mid v_1 = v_2 \mid i_1 \prec i_2 \mid @(\boldsymbol{a})$$
$$\mid \mathcal{L}(d) \mid \mathcal{D}(d) \mid \mathcal{K}^+(d) \mid \mathcal{I}^+(i) \mid \mathcal{HI}(t, i) \mid \boldsymbol{O}(d, \boldsymbol{a}) \mid \mathcal{G}(d, t) \mid \odot \mid \subset \tau \mid \prec \tau \mid \sqsubset \tau$$

$$\phi \in (\theta \in \text{KType}) * \{v \in \text{KVal} \mid \text{oftype}(v, \theta)\} \to \text{KExp}$$

Fig. 6. LoCK's syntax

reasoning about both trusted and non-trusted knowledge. First, Sec. 6.1 introduces the parameters on which LoCK depends. Sec. 6.2 describes its syntax and Sec. 6.3 its semantics. Sec. 6.4 presents LoCK's derivation rules, and their semantics. Finally, Sec. 6.6 and 6.7 show how to derive within LoCK general results about systems from typical assumptions. We among other things show how to lift properties about trusted sub-components to the level of distributed systems.

6.1 LoCK's Parameters

To be as general as possible, LoCK is parametrized by the types and functions described in Fig. 5. Sec. 7.2 explains how we can instantiate those parameters to derive high-level properties of several versions of MinBFT. LoCK can be instantiated for any kind of data (Data), trusted data[22] (Trust—a subset of Data), and identifier (Identifier—a partially 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 Sec. 7.1); (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.

6.2 LoCK's Syntax

As shown in Fig. 6, besides standard first-order logic operators ($\top$, $\bot$, $\wedge$, $\vee$, $\to$, $\exists$, $\forall$), LoCK also provides HyLoE-specific operators to state properties 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

[22] A piece of data is trusted if generated by a trusted component (e.g. UIs generated by USIGs in MinBFT—see Sec. 7.1).

$$\llbracket \top \rrbracket_e = \text{True} \qquad \llbracket \tau_1 \wedge \tau_2 \rrbracket_e = \llbracket \tau_1 \rrbracket_e \wedge \llbracket \tau_2 \rrbracket_e \qquad \llbracket \exists \phi \rrbracket_e = \exists v \in \{v \in \text{KVal} \mid \text{oftype}(v, \phi.\mathbf{1})\}.\phi.\mathbf{2}(v)$$
$$\llbracket \bot \rrbracket_e = \text{False} \qquad \llbracket \tau_1 \vee \tau_2 \rrbracket_e = \llbracket \tau_1 \rrbracket_e \vee \llbracket \tau_2 \rrbracket_e \qquad \llbracket \forall \phi \rrbracket_e = \forall v \in \{v \in \text{KVal} \mid \text{oftype}(v, \phi.\mathbf{1})\}.\phi.\mathbf{2}(v)$$
$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_e = \llbracket \tau_1 \rrbracket_e \rightarrow \llbracket \tau_2 \rrbracket_e$$

Fig. 7. LoCK's semantics (predicate logic)

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

Fig. 8. LoCK's semantics (logic of events)

$$\llbracket \mathcal{L}(d) \rrbracket_e = \text{learns}(e, d) \qquad \llbracket \mathcal{I}^+(i) \rrbracket_e = \text{ident}^+(e, i) \qquad \llbracket \mathcal{HI}(t, i) \rrbracket_e = \text{trustHasId}(t, i)$$
$$\llbracket \mathcal{D}(d) \rrbracket_e = d \in \text{sys} \rightsquigarrow e \qquad \llbracket v_1 = v_2 \rrbracket_e = v_1 = v_2 \qquad \llbracket \mathcal{O}(d, \boldsymbol{a}) \rrbracket_e = \text{owner}(d) = \boldsymbol{a}$$
$$\llbracket \mathcal{K}^+(d) \rrbracket_e = \text{knows}^+(e, d) \qquad \llbracket i_1 < i_2 \rrbracket_e = \text{lt}(i_1, i_2) \qquad \llbracket \mathcal{G}(d, t) \rrbracket_e = \text{genFor}(d, t)$$

Fig. 9. LoCK's semantics (knowledge)

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 $\text{oftype}(v, \theta)$ is true iff $(v, \theta) \in \{(i, \texttt{KTi}), (d, \texttt{KTd}), (t, \texttt{KTt}), (\boldsymbol{a}, \texttt{KTn})\}$.

LoCK also provides general operators to capture properties about distributed knowledge. As explained in Sec. 2.2, reasoning about distributed knowledge is a well studied topic. However, as opposed to the papers listed there, we follow here a more computational approach, i.e. one can always compute the knowledge at a given location. LoCK supports the standard knowledge *knows* ($\mathcal{K}^+$) operator, which is at the core of several knowledge calculi such as the ones mentioned above. LoCK also adopts *learns* ($\mathcal{L}$) and *owns* ($\mathcal{O}$) operators from Velisarios; and introduces a new *disseminate* ($\mathcal{D}$) operator. In addition, LoCK also includes the *knows identifier* ($\mathcal{I}^+$), *has identifier* ($\mathcal{HI}$), and *generated for* ($\mathcal{G}$) 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 $^-$ (see below) 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 $\exists_{\mathbf{i}} f$ stand for $\exists\langle \texttt{KTi}, f \rangle$, and $\exists_{\mathbf{i}} \lambda i_1, \ldots, i_n.\tau$ for $\exists_{\mathbf{i}} \lambda i. \ldots \exists_{\mathbf{i}} \lambda i_n.\tau$; and similarly for the other quantifiers. As usual, let $\neg\tau$ be $\tau \rightarrow \bot$. In addition, let

$$\preceq \tau = \prec \tau \vee \tau \qquad \mathcal{K}^-(\tau) = \subset \mathcal{K}^+(\tau) \qquad \boldsymbol{O}(d) = \exists_{\mathbf{n}} \lambda \boldsymbol{a}.@(\boldsymbol{a}) \wedge \boldsymbol{O}(d, \boldsymbol{a})$$
$$\sqsubseteq \tau = \sqsubset \tau \vee \tau \qquad \mathcal{I}^-(i) = \subset \mathcal{I}^+(i) \vee (i = \text{initId} \wedge \odot) \qquad \boldsymbol{OD}(d) = \boldsymbol{O}(d) \wedge \mathcal{D}(d)$$
$$\subseteq \tau = \subset \tau \vee (\tau \wedge \odot) \qquad i_1 \preceq i_2 = i_1 < i_2 \vee i_1 = i_2$$

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

6.3 LoCK's Semantics

Fig. 7, 8, and 9 describe LoCK's semantics: $\llbracket \tau \rrbracket_e$ 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, $\mathcal{L}$'s semantics is defined in terms of the learns predicate:

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

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

$x \in \mathsf{HypName}$ (a set of hypothesis names)
$\sigma \in \mathsf{KExpAt} ::= \tau \ @ \ e$
$h \in \mathsf{Hyp} ::= x : \sigma$
$g \in \mathsf{Guard} ::= y : \alpha$
$seq \in \mathsf{Sequent} ::= \langle G \rangle \ H \vdash \sigma$

$y \in \mathsf{GuardName}$ (a set of guard names)
$\alpha \in \mathsf{EventRel} ::= e_1 \equiv e_2 \mid e_1 \subset e_2 \mid e_1 \prec e_2 \mid e_1 \preceq e_2 \mid e_1 \sqsubset e_2 \mid e_1 \sqsubseteq e_2$
$H \in \mathsf{Hyps} ::= \oslash \mid H, h$
$G \in \mathsf{Guards} ::= \oslash \mid G, g$
$R \in \mathsf{Rule} ::= \dfrac{\Lambda[\overline{e}, \overline{t}, \overline{i}] \quad seq_1 \quad \cdots \quad seq_n}{seq}$

Fig. 10. Syntax of knowledge calculus rules

one has to be able to verify its authenticity. Then, $\mathcal{K}^+$ is interpreted by the knows^+ predicate:

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

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, $\mathcal{I}^+$ is interpreted by the ident^+ predicate:

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

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

6.4 LoCK's Rules

Syntax. Fig. 10 presents the syntax of rules. Expressions are annotated with 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).[23] 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:

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

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 Fig. 10) is true if $[\![seq]\!]$ (R's conclusion) follows from $[\![seq_1]\!] \wedge \cdots \wedge [\![seq_n]\!]$ (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. Additional rules such as LoCK's structural and predicate logic rules are presented in [Vukotic et al. 2019, Appx.C]. As mentioned 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 for space reasons).

Fig. 11 presents LoCK's event relation rules. The family of elimination rules $\square_{\mathtt{E}}$ allows turning HyLoE operators into guards, while the families of introduction rules $\square_{\mathtt{I}}$ and $\square_{\mathtt{It}}$ allow using those guards to navigate between points in space/time to prove HyLoE expressions. The two rules $\mathtt{if}\neg\odot$ and $\mathtt{if}\odot$ provide an axiomatization of $\mathsf{pred}^=$. The $\mathtt{weak}$ family of rules allows weakening guards,

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

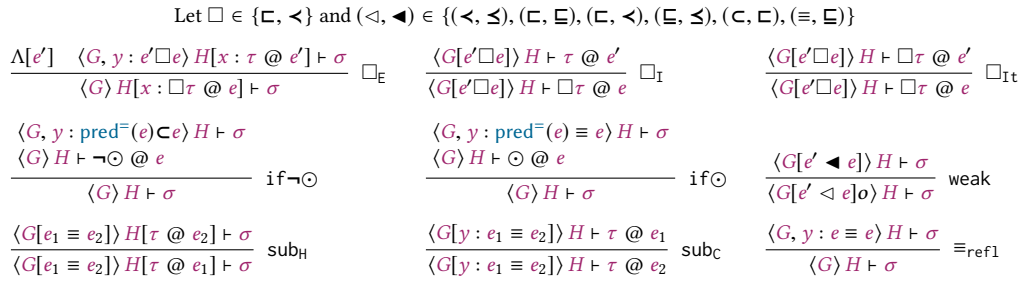


Fig. 11. LoCK's event relation rules

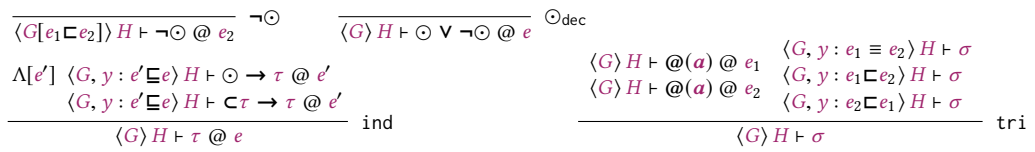


Fig. 12. LoCK's logic of events rules

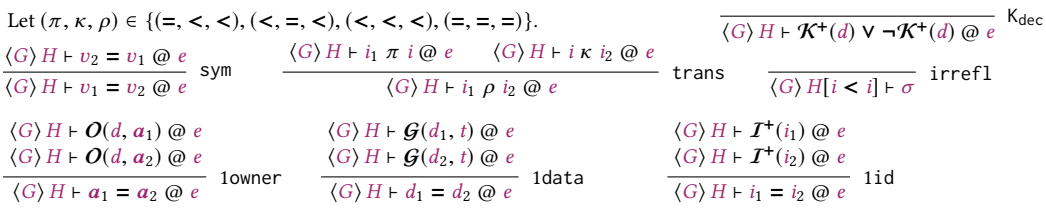


Fig. 13. LoCK's knowledge rules

e.g., from $\prec$ to $\preceq$ (strengthening rules are presented in [Vukotic et al. 2019, Appx.C]). Finally, using $\mathsf{sub}_{\mathsf{H}}$ and $\mathsf{sub}_{\mathsf{C}}$ one can substitute events in sequents' hypotheses and conclusions.

Fig. 12 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 `tri` rule axiomatizes the HyLoE fact that if two events e_1 and e_2 happen at the same location a, then either the events are equal, or one happened before the other. 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. Finally, $\odot_{\mathsf{dec}}$ states that $\odot$ is decidable.

Fig. 13 presents LoCK's knowledge rules. The $\mathsf{K}_{\mathsf{dec}}$ 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. Finally, the `1id` rule states that one can only know about a single identifier at any point in time.

6.5 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.[24]

[24] We use here some standard rules such as $\rightarrow_{\mathsf{E}}$ (implication elimination); $\vee_{\mathsf{E}}$ (or elimination); $\vee_{\mathtt{I1}}/\vee_{\mathtt{Ir}}$ (or introduction left/right); or `hyp` (hypothesis rule), which are described in [Vukotic et al. 2019, Appx.C].

Non-initial-events. We start by proving that if τ happened before, then the current event cannot be the initial event, i.e.: $\sqsubset\tau \to \neg\odot$ (see derivation on the right).[25] In this first example, we only navigate between events in the hypothesis x: we use the $\Box_E$ 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.

$$\dfrac{\dfrac{\dfrac{}{\langle y : e' \sqsubset e\rangle\ x : \tau\ @\ e' \vdash \neg\odot\ @\ e}\ \neg\odot}{\langle \oslash\rangle\ x : \sqsubset\tau\ @\ e \vdash \neg\odot\ @\ e}\ \Box_E}{\langle \oslash\rangle \oslash \vdash \sqsubset\tau \to \neg\odot\ @\ e}\ \to_E$$

Collapsing. We now prove another simple, though slightly more involved, example (see derivation on the right), 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.[26] We use the $\Box_E$ elimination rule 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 $\Box_{It}$ introduction rule to navigate to the e' intermediary point. Finally, we use the $\Box_I$ introduction rule to navigate to e'', while eliminating $\sqsubset$ (as opposed to the previous step, which keeps the operator).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle y : e' \sqsubset e, y' : e'' \sqsubset e'\rangle\ x : \tau\ @\ e'' \vdash \tau\ @\ e''}\ \text{hyp}}{\langle y : e' \sqsubset e, y' : e'' \sqsubset e'\rangle\ x : \tau\ @\ e'' \vdash \sqsubset\tau\ @\ e'}\ \Box_I}{\langle y : e' \sqsubset e, y' : e'' \sqsubset e'\rangle\ x : \tau\ @\ e'' \vdash \sqsubset\tau\ @\ e}\ \Box_{It}}{\langle y : e' \sqsubset e\rangle\ x : \sqsubset\tau\ @\ e' \vdash \sqsubset\tau\ @\ e}\ \Box_E}{\langle \oslash\rangle\ x : \sqsubset\sqsubset\tau\ @\ e \vdash \sqsubset\tau\ @\ e}\ \Box_E}{\langle \oslash\rangle \oslash \vdash \sqsubset\sqsubset\tau \to \sqsubset\tau\ @\ e}\ \to_E$$

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 $\langle G\rangle\ H[x : \sqsubseteq\tau\ @\ e] \vdash \sigma$ from $\langle G, y : e' \sqsubseteq e\rangle\ H[x : \tau\ @\ e'] \vdash \sigma$, 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:[27]

$$\dfrac{\dfrac{\dfrac{\Lambda[e']\ \langle G, y : e' \sqsubseteq e\rangle\ H[x : \tau\ @\ e'] \vdash \sigma}{\Lambda[e']\ \langle G, y : e' \sqsubset e\rangle\ H[x : \tau\ @\ e'] \vdash \sigma}\ \text{weak}}{\langle G\rangle\ H[x : \sqsubset\tau\ @\ e] \vdash \sigma}\ \Box_E \qquad \dfrac{\dfrac{\Lambda[e']\ \langle G, y : e' \sqsubseteq e\rangle\ H[x : \tau\ @\ e'] \vdash \sigma}{\langle G, y : e \sqsubseteq e\rangle\ H[x : \tau\ @\ e'] \vdash \sigma}\ \text{weak}}{\langle G, y : e \equiv e\rangle\ H[x : \tau\ @\ e'] \vdash \sigma}\ \text{weak}}{\langle G\rangle\ H[x : \tau\ @\ e] \vdash \sigma}\ \equiv_{\text{refl}}}{\langle G\rangle\ H[x : \sqsubseteq\tau\ @\ e] \vdash \sigma}\ \vee_E$$

Predecessor. Next, we prove that if τ was true at $\text{pred}^=(e)$ (denoted e_p below) then it must be that τ happened before or at e.[28] 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 $\odot_{\text{dec}}$); 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:

$$\dfrac{\Pi_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle y : e_p \equiv e\rangle\ x : \tau\ @\ e, o : \odot\ @\ e \vdash \tau\ @\ e}\ \text{hyp}}{\langle y : e_p \equiv e\rangle\ x : \tau\ @\ e_p, o : \odot\ @\ e \vdash \tau\ @\ e}\ \text{sub}_H \quad \Pi_2}{\langle \oslash\rangle\ x : \tau\ @\ e_p, o : \odot\ @\ e \vdash \tau\ @\ e}\ \text{if}\odot}{\langle \oslash\rangle\ x : \tau\ @\ e_p, o : \odot\ @\ e \vdash \sqsubseteq\tau\ @\ e}\ \vee_{Ir} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\langle y : e_p \sqsubset e\rangle\ x : \tau\ @\ e_p, o : \neg\odot\ @\ e \vdash \tau\ @\ e_p}\ \text{hyp}}{\langle y : e_p \sqsubset e\rangle\ x : \tau\ @\ e_p, o : \neg\odot\ @\ e \vdash \sqsubset\tau\ @\ e}\ \Box_I}{\langle y : e_p \sqsubset e\rangle\ x : \tau\ @\ e_p, o : \neg\odot\ @\ e \vdash \sqsubset\tau\ @\ e}\ \text{weak} \quad \Pi_3}{\langle \oslash\rangle\ x : \tau\ @\ e_p, o : \neg\odot\ @\ e \vdash \sqsubset\tau\ @\ e}\ \text{if}\neg\odot}{\langle \oslash\rangle\ x : \tau\ @\ e_p, o : \neg\odot\ @\ e \vdash \sqsubseteq\tau\ @\ e}\ \vee_{I1}}{\langle \oslash\rangle\ x : \tau\ @\ e_p, o : \odot \vee \neg\odot\ @\ e \vdash \sqsubseteq\tau\ @\ e}\ \vee_E}{\langle \oslash\rangle\ x : \tau\ @\ e_p \vdash \sqsubseteq\tau\ @\ e}\ \text{cut}$$

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

[25] See `DERIVED_RULE_local_before_implies_not_first_true` in `model/CalculusSM_derived3.v`.
[26] See `DERIVED_RULE_twice_local_before_implies_once_true` in `model/CalculusSM_derived3.v`.
[27] See `DERIVED_RULE_unlocal_before_eq_hyp_true` in `model/CalculusSM.v`.
[28] See `DERIVED_RULE_at_pred_implies_local_before_eq_true` in `model/CalculusSM_derived3.v`.

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'.[29] We state this fact as a derived rule as follows:

$$\frac{\langle G\rangle\, H \vdash \mathcal{K}^{+}(d)\ @\ e}{\langle G\rangle\, H \vdash \sqsubseteq(\mathcal{K}^{+}(d) \wedge \neg\mathcal{K}^{-}(d))\ @\ e} \tag{1}$$

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

6.6 Typical System Assumptions and Consequences

In order to derive general results about distributed knowledge, such as in Sec. 6.7, let us first present some typical assumptions about knowledge, which we express here within LoCK (see the file called `model/CalculusSM.v` for more details). We illustrate in Sec. 7.2 that those assumptions indeed make sense, by validating them to, in turn, derive properties about MinBFT from those general results.

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

$$\mathbf{LID} = \forall_{\mathtt{t}}\lambda t.\mathcal{L}(t) \rightarrow \prec(\mathcal{OD}(t)) \tag{2}$$

$$\mathbf{KLD} = \forall_{\mathtt{t}}\lambda t.\mathcal{K}^{+}(t) \rightarrow (\mathcal{K}^{-}(t) \vee \mathcal{L}(t) \vee \mathcal{OD}(t)) \tag{3}$$

$$\mathbf{Mon} = (\exists_{\mathtt{i}}\lambda i.\mathcal{I}^{-}(i) \wedge \mathcal{I}^{+}(i)) \vee (\exists_{\mathtt{i}}\lambda i_1, i_2.i_1 \prec i_2 \wedge \mathcal{I}^{-}(i_1) \wedge \mathcal{I}^{+}(i_2)) \tag{4}$$

$$\mathbf{New} = \forall_{\mathtt{t}}\lambda t.\forall_{\mathtt{i}}\lambda i, i_1, i_2.(\mathcal{OD}(t) \wedge \mathcal{I}^{-}(i_1) \wedge \mathcal{I}^{+}(i_2)) \rightarrow (i_1 \prec i \wedge i \preceq i_2 \wedge \mathcal{HI}(t, i) \wedge \neg\mathcal{HI}(t, i_1)) \tag{5}$$

$$\mathbf{Uniq} = \forall_{\mathtt{t}}\lambda t_1, t_2.\forall_{\mathtt{i}}\lambda i.(\mathcal{OD}(t_1) \wedge \mathcal{OD}(t_2) \wedge \mathcal{HI}(t_1, i) \wedge \mathcal{HI}(t_2, i)) \rightarrow t_1 = t_2 \tag{6}$$

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. 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 disjunct), or it increases (right disjunct). 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** (Eq. 3) and using LoCK's induction on causal time rule (`ind`), we can derive:[30] $\mathcal{K}^{+}(t) \rightarrow \sqsubseteq\mathcal{L}(t) \vee \sqsubseteq\mathcal{OD}(t)$. Then, using **LID** (Eq. 2), and using a similar *collapsing* result as the one presented in Sec. 6.5 above (to collapse $\sqsubseteq\prec$ into $\preceq$ here), we can further derive:[31]

$$\mathcal{K}^{+}(t) \rightarrow \preceq(\mathcal{OD}(t)) \tag{7}$$

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:[32]

$$\frac{\Lambda[e']\ \ \langle G\rangle\, H \vdash \mathbf{Mon} \wedge \mathbf{New} \wedge \mathbf{Uniq}\ @\ e' \qquad \begin{array}{l}\langle G\rangle\, H \vdash \mathcal{OD}(t_1) \wedge \mathcal{HI}(t_1, i) \wedge @(\boldsymbol{a})\ @\ e_1\\ \langle G\rangle\, H \vdash \mathcal{OD}(t_2) \wedge \mathcal{HI}(t_2, i) \wedge @(\boldsymbol{a})\ @\ e_2\end{array}}{\langle G\rangle\, H \vdash t_1 = t_2\ @\ e} \tag{8}$$

This derived rule is critical to prove Thm. 6.1 in Sec. 6.7. 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

[29] See the lemma called `DERIVED_RULE_knowledge_acquired_true` in the filed called `model/CalculusSM.v`.
[30] See the lemma called `DERIVED_RULE_trusted_KLD_implies_or_true` in the file called `model/CalculusSM.v`.
[31] See the lemma called `DERIVED_RULE_trusted_KLD_implies_gen_true` in the file called `model/CalculusSM.v`.
[32] See the lemma called `DERIVED_RULE_trusted_disseminate_unique_ex_true` in the file called `model/CalculusSM.v`.

result using LoCK's trichotomy rule `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 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.

6.7 Distributed Lifting

Theorem 6.1 (Distributed Lifting). *Using the above mentioned rules and assumptions, we derived the following rule within LoCK:*[33]

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

This derived rule allows lifting properties of trusted sub-components to the level of a distributed system. It states that if all assumptions presented in Sec. 6.6 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 Sec. 7.2).

Proof Sketch 1. *We derive here Thm. 6.1 essentially from the "derived knowledge" formula 7 and the "uniqueness" derived rule 8 presented above. From $\mathcal{K}^{+}(t_1)$ (at e_1) and $\mathcal{K}^{+}(t_2)$ (at e_2), we can derive using Eq. 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 8. Finally, we derive that $d_1 = d_2$ using LoCK's* `1data` *inference rule.*

6.8 Example: Micro's Agreement

As mentioned above, we used Thm. 6.1 to prove the agreement property of the Micro system defined in Sec. 3 (as well as of the MinBFT variants discussed in Sec. 7). For that we first need to instantiate LoCK's parameters (we only discuss some of the most interesting parameters—see `MinBFT/MicroBFTkn.v` for more details). We instantiate Data by the union type that contains commit messages, accept messages, and UIs, i.e., all pieces of data that mention a counter; Identifier is instantiated by $\mathbb{N}$; and Trust is the type of UIs as generated by usig components. The sys parameter is instantiated by Micro; mem is instantiated by log; trust is instantiated by usig; trustHasId(ui, i) is true if i is the counter contained in ui; know(d, m) is true if d occurs in the list of commits m maintained by log; verify(e, $auth$) returns true iff the usig component running at e can indeed verify $auth$; trusted2id returns the counter maintained by the usig component; lt is $<$; and initId is 0.

Getting back to Micro's agreement property: we have to prove that if the backups accept two requests r_1 and r_2 both with trusted counter value i (generated by the primary), then those requests must be equal. See Sec. 3 for a formal statement of this property. From the facts that the two requests r_1 and r_2 were accepted at e_1 and e_2, respectively, we derive that those requests must have been known at these two points. More precisely, because as explained in Sec. 5.3, the commits corresponding to those two requests must be logged, then there must exist two pieces of trusted data (two UIs) ui_1 and ui_2, such that $[\![\mathcal{K}^{+}(ui_1)]\!]_{e_1}$, $[\![\mathcal{K}^{+}(ui_2)]\!]_{e_2}$, ui_1 corresponds to the piece of data $\langle r_1, i\rangle$, i.e. $[\![\mathcal{G}(\langle r_1, i\rangle, ui_1)]\!]_{e_1}$, and ui_2 corresponds to the piece of data $\langle r_2, i\rangle$, i.e. $[\![\mathcal{G}(\langle r_2, i\rangle, ui_2)]\!]_{e_2}$.

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

Moreover, both ui_1 and ui_2 have trusted counter i, i.e. $[\![\mathcal{HI}(ui_1, i)]\!]_{e_1}$ and $[\![\mathcal{HI}(ui_2, i)]\!]_{e_2}$, and both were generated (are owned) by the primary, i.e. $[\![\mathcal{O}(ui_1, \text{primary})]\!]_{e_1}$ and $[\![\mathcal{O}(ui_2, \text{primary})]\!]_{e_2}$. We are now ready to use Thm. 6.1. To use this LoCK theorem in our HyLoE proof, we use the fact that it is true w.r.t. its HyLoE semantics described in Sec. 6.3. Namely, we derive $[\![\langle r_1, i\rangle = \langle r_2, i\rangle]\!]_e$ (for any event e) from the fact that $[\![\texttt{LID}]\!]_{e'}$, $[\![\texttt{KLD}]\!]_{e'}$, $[\![\texttt{Mon}]\!]_{e'}$, $[\![\texttt{New}]\!]_{e'}$, and $[\![\texttt{Uniq}]\!]_{e'}$ are true at all events e'. These assumptions are straightforwardly true about Micro, and are proved within HyLoE directly. Finally, because $[\![\langle r_1, i\rangle = \langle r_2, i\rangle]\!]_e$, i.e., $\langle r_1, i\rangle = \langle r_2, i\rangle$, we conclude that $r_1 = r_2$. High-level results such as Thm. 6.1 allow us to capture the logic of distributed systems at a high-level of abstraction, leaving proving simple protocol-dependent properties directly within HyLoE.[34]

7 CASE STUDIES: USIG- AND TRINC-BASED MINBFT

We exercised Asphalion by implementing and verifying two versions of the seminal MinBFT hybrid protocol [Veronese et al. 2013]: one based on USIGs (as in the original version), and one based on TrIncs [Levin et al. 2009]. As discussed below, USIGs and TrIncs have different pros and cons that make them both interesting to use and verify. We proved the agreement property of both versions using Thm. 6.1, which we proved within LoCK (see Sec. 6.7). 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 [Behl et al. 2017; Chun et al. 2007; Kapitza et al. 2012]. We now present MinBFT (see [Veronese 2010; Veronese et al. 2013] for further details), starting with a description of the trusted components our implementations rely on.

7.1 MinBFT Recap

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 messages. Every sent message is supposed to be tagged with a USIG-generated 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: to 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.

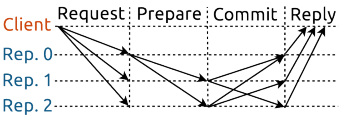
```
var counter=0; var id; var keys;

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); }
```

TrInc. In [Levin et al. 2009], the authors introduced a new kind of trusted components called Trusted Incrementer (TrInc for short). TrInc is more general than USIG in the 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 value is provided by the client of the TrInc and has to be greater than the current value (see [Levin et al. 2009] 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 [Behl et al. 2017] to be based on TrInc instead of USIG.

[34] As discussed in [Vukotic et al. 2019, Appx.D], we have also proved the crux of Micro's validity property within LoCK.



MinBFT. MinBFT works in a succession of configurations called *views*. In each view v, the distinguished replica $p = v \bmod n$ (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 above diagram); and switches to a *view-change* operation otherwise.[35] We focus here on the normal case operation, which works as follows:

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

2. When the primary p receives a request m, it calls its USIG 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. 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. 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 res of this execution in a reply $\langle \texttt{REPLY}, k, seq, res \rangle_{\sigma_k}$ to the client. In addition, upon receipt of a new commit, k calls its USIG 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, a replica i handles a message only if: (1) the message is signed properly in case of *requests*; (2) the message comes from the current primary in case of *prepares*; (3) the view number is the current one; and (4) upon receipt of a UI from a replica j, replica i has already received all the UIs from j with lower counter values.

7.2 Implementation and Verification of MinBFT

Let us now describe how we used Asphalion to implement the two variants of MinBFT mentioned above using MoC, and verify their correctness using HyLoE and LoCK. We focus on the USIG-based version, and only mention the TrInc-based one when the two versions differ.

MinBFT system. In our MoC implementation of MinBFT (see `MinBFT/MinBFT.v` for more details), a replica is a local system called MinBFTlocalSys. Each local system 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 Sec. 7.1; 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.

MinBFT knowledge. To verify properties about MinBFT using LoCK, we had to instantiate the parameters presented in Fig. 5.[36] We only discuss here some of the most interesting parameters. 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 messages. Trust is instantiated with the type of UIs, and Identifier is instantiated with the type of counters. 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).

[35] MinBFT provides a garbage collection process to discard messages so as not to exhaust the memory; and a view-change process to ensure liveness. Those are outside the scope of this paper, and are left as future work, because the normal phase operation provides the necessary and sufficient context to address the challenges of reasoning about hybrid systems.

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

Verified properties. Using Asphalion 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):[37]

```
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).
```

The AXIOM_auth_messages_were_sent_or_byz axiom is discussed below. This lemma states that if a correct replica executes a client request *r* with counter value *i1* (which triggers the sending of an *accept* message), 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 (all the other replicas could well be faulty). As mentioned above, this lemma is a straightforward consequence of the general Thm. 6.1 proved within LoCK and presented in Sec. 6.7.

Knowledge assumptions. Because Thm. 6.1 relies on some assumptions (see Sec. 6.6), we had to prove that those are indeed true about our MinBFT implementations. **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 Thm. 5.1, described in Sec. 5.4. 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. **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*.

7.3 Differences from the Original Proof

As it turns out, our proof of agreement_iff is significantly simpler than the original pen-and-paper proof [Veronese 2010, pp.151–153]. The original proof of the ← direction, which we claim here to be unnecessarily convoluted, goes as follows: given that two quorums of $f + 1$ replicas each have committed *(r,i1)* and *(r,i2)*, respectively, 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, this 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 counters corresponding to the two quorums mentioned above. We can then trace back these two counters to the time the 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 Asphalion's operators, such as $ls \rightsquigarrow e$ described in Sec. 5.2, 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.

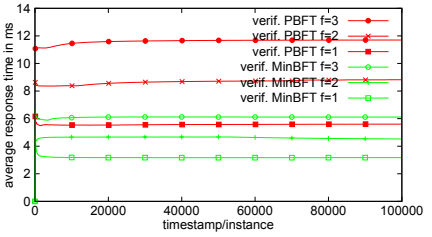
[37] See the the files called `MinBFT/MinBFTagreement_iff.v` and `MinBFT/TrIncagreement_iff.v`.

8 EVALUATION

Extraction. We use Coq's extraction mechanism to obtain executable OCaml code from our distributed systems implemented in MoC (see Sec. 5.1). However, because we want to run the different components of a local system separately (i.e. execute the trusted ones within trusted environments such as Intel SGX), the monad structure is "erased" during extraction.[38] Instead, a separate module is created for each component, and calls to sub-components are extracted to calls to those modules. In addition, the functional states of MoC components are turned into imperative ones within those modules.[39] Running the sub-components of a local system separately enables executing the trusted ones within trusted environments, in our case Intel SGX enclaves.

Trusted execution. We use Graphene-SGX [Tsai et al. 2017], a library for running unmodified applications inside Intel SGX enclaves, to execute MinBFT's trusted USIG components inside enclaves (see `MinBFT/runtime_w_sgx/README.md` or [Vukotic et al. 2019, Appx.G] for details). Because Graphene-SGX's driver closes enclaves after each call, and because only part of the extracted code is meant to run inside enclaves, our SGX-based runtime environment uses a TCP interface for replicas to interact with USIGs running in Graphene-SGX enclaves. Also, 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, and OCaml wrappers around the TCP interface implemented in C (these wrappers use [OCaml2C 2019]). Note that to support calling the interfaces of trusted components through the above mentioned TCP interface, one has to write custom serializers/deserializers (see for example `MinBFT/runtime_w_sgx/tcp_client.c` and `MinBFT/runtime_w_sgx/tcp_server.c`). We leave it for future work to generate those automatically.

Comparison. As shown on the right, the average latency of our USIG-based implementation is lower than the one of the verified version of PBFT presented in [Rahli et al. 2018]. Although Graphine-SGX incurs some overhead, our MinBFT implementation is faster because: (1) MinBFT uses less communication steps than PBFT; and (2) our MinBFT implementation uses less expensive crypto than in [Rahli et al. 2018]. Our experiments use one client, $f \in \{1, 2, 3\}$, and the replicated service is a state machine that accumulates inputs using addition. We used a desktop with 16GB of RAM, and 8 i7-6700 cores running at 3.40GHz.



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 Sec. 4); (2) the validity of the assumption described in Sec. 7.2; (3) Coq's logic and implementation; (4) our runtime environment implemented in OCaml (Sec. 8); (5) and the hardware and software on which our framework is running.

Proof Effort. Our model is about 13.8K lines of spec. and 11.9K lines of proofs, while our MinBFT proofs are about 7.9K lines of spec. and 4.7K lines of proofs (excluding the code we reused from Velisarios). Developing Asphalion and partially verifying MinBFT took us about one person-year.

9 RELATED WORK

Several logics, models and tools have been developed over the years to reason about distributed systems (see Fig. 14). However, to the best of our knowledge, Asphalion is the first theorem prover based framework for verifying the correctness of implementations of hybrid fault-tolerant protocols.

[38] The monad erasure we perform is very simple and standard (see `MinBFT/runtime_w_sgx/MinBFTinstance.v`).
[39] Verifying the correctness of this "compilation" phase is left for future work.

	Running code	Byz. (synch.)	Byz. (asynch.)	Hybrid
ConsL/DISEL/EventML/IronFleet/Ivy/ModP/PSync/Verdi	✓	✗	✗	✗
PVS	✗	✓	✗	✗
HO-model/ByMC/IOA/TLA$^+$	✗	✓	✓	✗
Event-B	✓/✗	✓	✗	✗
Velisarios	✓	✓	✓	✗
Asphalion	✓	✓	✓	✓

Fig. 14. Comparison with related work

9.1 Logics and Models

Event-B [Abrial 2010; Abrial et al. 2010] is a set-theory-based language for modeling reactive systems and for *refining* high-level abstract specifications into low-level ones. It supports code generation [Fürst et al. 2014; Méry and Singh 2011] (not all features are covered), and has been used in a number of projects [Krenický and Ulbrich 2010; Lynch 1996], e.g., to prove the agreement and validity of synchronous Byzantine agreement algorithms [Krenický and Ulbrich 2010].

The Heard-Of (HO) model [Biely et al. 2007; Charron-Bost and Schiper 2009] requires protocols to be divided into rounds, allowing processes to execute in lock-step. It was implemented in Isabelle/HOL [Charron-Bost et al. 2011] and used to verify the EIGByz [Bar-Noy et al. 1992] Byzantine agreement algorithm for synchronous systems. Model checking and the HO-model have also been used in [Chaouch-Saad et al. 2009; Tsuchiya and Schiper 2007, 2008] to verify crash fault-tolerant consensus algorithms [Charron-Bost and Schiper 2009].

IOA [Garland et al. 2004; Garland and Lynch 2000; Georgiou et al. 2009; Tauber 2004] is a programming/specification language for describing asynchronous distributed systems as I/O automata [Lynch and Tuttle 1987] and for stating their properties.

TLA$^+$ [Chaudhuri et al. 2010; Lamport 1994, 2004] is a language for specifying and reasoning about systems, that combines a temporal logic for describing systems, and set theory to specify data structures. It has been used in a large number of projects [Bolosky et al. 2007; Chand et al. 2016; Joshi et al. 2003; Lu et al. 2011; Newcombe et al. 2015], including to prove Multi-Paxos' safety and liveness [Chand et al. 2016], and the safety of a variant of an abstract model of PBFT [BPaxos 2018].

9.2 Tools

ByMC [Konnov et al. 2015, 2017a,b; Lazic et al. 2017] is a model checker for verifying the safety and liveness of BFT algorithms, which can automatically check parametrized threshold-guarded algorithms (e.g., where nodes wait for messages from a majority of senders). It relies on a short counterexample property, which says that if a distributed algorithm violates a temporal specification then there is a parameter independent counterexample of bounded length.

ConsL [Maric et al. 2017] is a language for expressing crash-fault tolerant consensus algorithms, whose semantics is expressed in HO, and that connects to the Spin model checker [Holzmann 2004]. 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.

DISEL [Sergey et al. 2018] is a framework for modular verification of implementations of crash fault tolerant systems. It provides a programming language shallowly embedded in Coq, and a separation-style program logic. It introduces two techniques enabling modular verification: the WITHINV rule to strengthen assumptions, and *send-hooks* to allow logical access between components.

EventML [Bickford et al. 2012; Rahli et al. 2015, 2017] is a domain specific language implemented on top of the Nuprl prover [Constable et al. 1986]. It provides expressive and modular combinators

for implementing and reasoning about crash-fault tolerant distributed systems (e.g., the authors proved Multi-Paxos' safety [Rahli et al. 2012; Schiper et al. 2012, 2014]).

IronFleet [Hawblitzel et al. 2015, 2017] uses a combination of Dafny, Hoare logic and TLA to automatically verify the safety and liveness of distributed protocols. The authors proved the safety and liveness of a Paxos-based replication protocol, as well as a distributed key value store.

Ivy [Padon et al. 2016] initially supported debugging infinite-state systems using bounded verification, and verifying their safety by gradually building inductive invariants. Their *decidable decomposition* notion [Taube et al. 2018] (i.e., systems, models and proofs must be built modularly to enable the use of different decidable logics) allowed Ivy to automatically verify the correctness of *implementations* of crash-fault tolerant systems such as Raft and Paxos (as opposed to models in [Padon et al. 2017]). Ivy also supports liveness by reducing it to safety [Padon et al. 2018].

ModP [Desai et al. 2018] is a programming framework to build, specify and compositionally test dynamic, asynchronous distributed systems. The authors used it to modularly implement and validate (through testing) two fault-tolerant distributed systems (including Multi-Paxos).

PSync [Dragoi et al. 2016] is an HO-based domain specific language embedded in Scala, that enables executing and verifying synchronous and partially asynchronous crash fault-tolerant distributed algorithms. It relies on the multi-sorted first-order *Consensus verification logic* (CL) [Dragoi et al. 2014]. To prove safety, users have to provide invariants, which CL checks for validity.

PVS was extensively used to verify synchronous systems that tolerate malicious faults [Schmid et al. 2002], to the extent that these verification efforts influenced its design [Owre et al. 1995].

Velisarios [Rahli et al. 2018] is a Coq-based framework for verifying the correctness of homogeneous BFT systems. It provides a knowledge library to reason about systems at a high-level of abstraction. Using Velisarios, the authors verified PBFT's agreement property [Castro 2001].

Verdi [Wilcox et al. 2015; Woos et al. 2016] is a framework to develop and reason about crash-fault tolerant distributed systems using Coq, that can generate running OCaml code. Verdi provides a compositional way of specifying distributed systems, by applying *verified system transformers* (e.g., Raft [Ongaro and Ousterhout 2014] transforms a distributed system into a crash-tolerant one).

10 CONCLUSIONS AND FUTURE WORK

This paper introduces Asphalion, the first theorem prover-based framework to reason about executable hybrid fault-tolerant systems, which have been getting increasing attention over the past few years. It provides three novel languages: HyLoE, a hybrid logic of events to model hybrid systems; MoC, a monadic programming language to implement systems composed of interacting components; and LoCK, a sound hybrid knowledge calculus to reason about systems at a high-level of abstraction. In addition, Asphalion introduces novel proof techniques to lift properties about (trusted) sub-components to the level of distributed systems. Using Asphalion, we proved among other things the agreement property of two variants of the seminal MinBFT protocol.

In the future, we would like to extend LoCK so that some proofs about distributed knowledge could be automated. In addition, we would like to investigate whether LoCK specifications could be compiled to running code. We also wish to implement a formally verified compiler from MoC to imperative code. Finally, we plan to exercise Asphalion further by verifying other hybrid protocols.

ACKNOWLEDGMENTS

The authors thank Christoph Lambert for his invaluable help and for sharing his SGX expertise.

This work is partially supported by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

REFERENCES

2014. *DSN 2014*. IEEE. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6900116

Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017a. Revisiting Fast Practical Byzantine Fault Tolerance. *CoRR* abs/1712.01367 (2017). arXiv:1712.01367 http://arxiv.org/abs/1712.01367

Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. 2017b. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus, See [Aspnes et al. 2018], 25:1–25:19. https://doi.org/10.4230/LIPIcs.OPODIS.2017.25

Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press. http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569

Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: an open toolset for modelling and reasoning in Event-B. 12, 6 (2010), 447–466. https://doi.org/10.1007/s10009-010-0145-y

Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). 2017. *EUROSYS 2017*. ACM. https://doi.org/10.1145/3064176

Abhishek Anand and Ross A. Knepper. 2015. ROSCoq: Robots Powered by Constructive Reals. In *ITP-6 (LNCS)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 34–50. https://doi.org/10.1007/978-3-319-22102-1_3

James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão (Eds.). 2018. *OPODIS 2017*. LIPIcs, Vol. 95. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. http://www.dagstuhl.de/dagpub/978-3-95977-061-3

Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. 1992. Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement. *Inf. Comput.* 97, 2 (1992), 205–233. https://doi.org/10.1016/0890-5401(92)90035-E

Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT, See [Alonso et al. 2017], 222–237. https://doi.org/10.1145/3064176.3064213

Ido Ben-Zvi. 2011. *Causality, Knowledge and Coordinaltion in Distributed Systems*. Ph.D. Dissertation. Technion – Computer Science Department.

Ido Ben-Zvi and Yoram Moses. 2014. Beyond Lamport's Happened-before: On Time Bounds and the Ordering of Events in Distributed Systems. *J. ACM* 61, 2 (2014), 13:1–13:26. https://doi.org/10.1145/2542181

Yves Bertot and Pierre Casteran. 2004. *Interactive Theorem Proving and Program Development*. SpringerVerlag. http://www.labri.fr/perso/casteran/CoqArt.

Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART, See [DBL 2014], 355–362. https://doi.org/10.1109/DSN.2014.43

Mark Bickford. 2009. Component Specification Using Event Classes. In *CBSE 2009 (LNCS)*, Grace A. Lewis, Iman Poernomo, and Christine Hofmeister (Eds.), Vol. 5582. Springer, 140–155.

Mark Bickford, Robert L. Constable, Joseph Y. Halpern, and Sabina Petride. 2004. Knowledge-Based Synthesis of Distributed Systems Using Event Structures. In *LPAR 2004 (LNCS)*, Franz Baader and Andrei Voronkov (Eds.), Vol. 3452. Springer, 449–465. https://doi.org/10.1007/978-3-540-32275-7_30

Mark Bickford, Robert L. Constable, and Vincent Rahli. 2012. Logic of Events, a framework to reason about distributed systems. In *Languages for Distributed Algorithms Workshop*. http://www.nuprl.org/documents/Bickford/LOE-LADA2012.html

Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. 2007. Tolerating corrupted communication. In *PODC 2007*, Indranil Gupta and Roger Wattenhofer (Eds.). ACM, 244–253. https://doi.org/10.1145/1281100.1281136

William J. Bolosky, John R. Douceur, and Jon Howell. 2007. The Farsite project: a retrospective. *Operating Systems Review* 41, 2 (2007), 17–26. https://doi.org/10.1145/1243418.1243422

BPaxos 2018. *Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm*. http://lamport.azurewebsites.net/tla/byzpaxos.html

Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. 2014. Unbeatable Consensus. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings (LNCS)*, Fabian Kuhn (Ed.), Vol. 8784. Springer, 91–106. https://doi.org/10.1007/978-3-662-45174-8_7

Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. 2016. 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*, George Giakkoupis (Ed.). ACM, 107–116. https://doi.org/10.1145/2933057.2933120

Miguel Castro. 2001. *Practical Byzantine Fault Tolerance*. Ph.D. MIT. Also as Technical Report MIT-LCS-TR-817.

Miguel Castro and Barbara Liskov. 1999a. *A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm*. Technical Memo MIT-LCS-TM-590. MIT.

Miguel Castro and Barbara Liskov. 1999b. Practical Byzantine Fault Tolerance. In *OSDI 1999*, Margo I. Seltzer and Paul J. Leach (Eds.). USENIX Association, 173–186. https://doi.org/10.1145/296806.296824

Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *FM 2016 (LNCS)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.), Vol. 9995. 119–136. https://doi.org/10.1007/978-3-319-48989-6_8

K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75. https://doi.org/10.1145/214451.214456

K. Mani Chandy and Jayadev Misra. 1986. How Processes Learn. *Distributed Computing* 1, 1 (1986), 40–52. https://doi.org/10.1007/BF01843569

Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. 2009. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *RP 2009 (LNCS)*, Olivier Bournez and Igor Potapov (Eds.), Vol. 5797. Springer, 93–106. https://doi.org/10.1007/978-3-642-04420-5_10

Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. 2011. Formal Verification of Consensus Algorithms Tolerating Malicious Faults. In *SSS 2011 (LNCS)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer, 120–134. https://doi.org/10.1007/978-3-642-24550-3_11

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing* 22, 1 (2009), 49–71. https://doi.org/10.1007/s00446-009-0084-6

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. Verifying Safety Properties with the TLA+ Proof System. In *IJCAR 2010 (LNCS)*, Jürgen Giesl and Reiner Hähnle (Eds.), Vol. 6173. Springer, 142–148. https://doi.org/10.1007/978-3-642-14203-1_12

Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: making adversaries stick to their word. In *SOSP 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 189–204. https://doi.org/10.1145/1294261.1294280

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. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Coq 2019. *The Coq Proof Assistant.* http://coq.inria.fr/

Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. 2005. Low complexity Byzantine-resilient consensus. *Distributed Computing* 17, 3 (2005), 237–249. https://doi.org/10.1007/s00446-004-0110-7

Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2004. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. In *SRDS 2004*. IEEE Computer Society, 174–183. https://doi.org/10.1109/RELDIS.2004.1353018

Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2013. BFT-TO: Intrusion Tolerance with Less Replicas. *Comput. J.* 56, 6 (2013), 693–715. https://doi.org/10.1093/comjnl/bxs148

Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. 2002. The Design of a COTSReal-Time Distributed Security Kernel. In *EDCC-4 (LNCS)*, Fabrizio Grandoni and Pascale Thévenod-Fosse (Eds.), Vol. 2485. Springer, 234–252. https://doi.org/10.1007/3-540-36080-8_21

Asa Dan, Rajit Manohar, and Yoram Moses. 2017. On Using Time Without Clocks via Zigzag Causality. In *PODC 2017*, Elad Michael Schiller and Alexander A. Schwarzmann (Eds.). ACM, 241–250. https://doi.org/10.1145/3087801.3087839

Christian Decker, Jochen Seidel, and Roger Wattenhofer. 2016. Bitcoin meets strong consistency. In *ICDCN 2016*. ACM, 13:1–13:10. https://doi.org/10.1145/2833312.2833321

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *PACMPL* 2, OOPSLA (2018), 159:1–159:30. https://doi.org/10.1145/3276529

Tobias Distler, Christian Cachin, and Rüdiger Kapitza. 2016. Resource-Efficient Byzantine Fault Tolerance. *IEEE Trans. Computers* 65, 9 (2016), 2807–2819. https://doi.org/10.1109/TC.2015.2495213

Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-Based Framework for Verifying Consensus Algorithms. In *VMCAI 2014 (LNCS)*, Kenneth L. McMillan and Xavier Rival (Eds.), Vol. 8318. Springer, 161–181. https://doi.org/10.1007/978-3-642-54013-4_10

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2015. The Need for Language Support for Fault-Tolerant Distributed Systems. In *SNAPL 2015 (LIPIcs)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 90–102. https://doi.org/10.4230/LIPIcs.SNAPL.2015.90

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 400–415. https://doi.org/10.1145/2837614.2837650

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Cynthia Dwork and Yoram Moses. 1990. Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures. *Inf. Comput.* 88, 2 (1990), 156–186. https://doi.org/10.1016/0890-5401(90)90014-9

Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2017. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *WiSec 2017*, Guevara Noubir, Mauro Conti, and Sneha Kumar Kasera (Eds.). ACM, 99–110. https://doi.org/10.1145/3098243.3098261

Ronald Fagin, Joseph Halpern, Yoram Moses, and Moshe Vardi. 2003. *Reasoning About Knowledge*. https://doi.org/10.7551/mitpress/5803.001.0001

Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. 1997. Knowledge-Based Programs. *Distributed Computing* 10, 4 (1997), 199–225. https://doi.org/10.1007/s004460050038

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems, See [Alonso et al. 2017], 328–343. https://doi.org/10.1145/3064176.3064183

Andreas Fürst, Thai Son Hoang, David A. Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. 2014. Code Generation for Event-B. In *IFM 2014 (LNCS)*, Elvira Albert and Emil Sekerinski (Eds.), Vol. 8739. Springer, 323–338. https://doi.org/10.1007/978-3-319-10181-1_20

S. Garland, N. Lynch, J. Tauber, and M. Vaziri. 2004. *IOA user guide and reference manual.* Technical Report MIT/LCS/TR-961. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Stephen J. Garland and Nancy Lynch. 2000. Using I/O automata for developing distributed systems. In *Foundations of componentbased systems*, Gary T. Leavens and Murali Sitaraman (Eds.). Cambridge University Press, New York, NY, USA, 285–312. http://dl.acm.org/citation.cfm?id=336431.336455

Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. 2009. Automated implementation of complex distributed algorithms specified in the IOA language. *Int. J. Softw. Tools Technol. Transf.* 11 (February 2009), 153–171. Issue 2. https://doi.org/10.1007/s10009-008-0097-7

Guy Goren and Yoram Moses. 2018. Silence. In *PODC 2018*, Calvin Newport and Idit Keidar (Eds.). ACM, 285–294. https://doi.org/10.1145/3212734

Joseph Y. Halpern. 1987. Using Reasoning About Knowledge to Analyze Distributed Systems. *Annual Review of Computer Science* 2, 1 (1987), 37–68. https://doi.org/10.1146/annurev.cs.02.060187.000345 arXiv:https://doi.org/10.1146/annurev.cs.02.060187.000345

Joseph Y. Halpern and Yoram Moses. 1990. Knowledge and Common Knowledge in a Distributed Environment. *J. ACM* 37, 3 (1990), 549–587. https://doi.org/10.1145/79147.79161

Joseph Y. Halpern and Rafael Pass. 2017. A Knowledge-Based Analysis of the Blockchain Protocol. In *TARK 2017 (EPTCS)*, Jérôme Lang (Ed.), Vol. 251. 324–335. https://doi.org/10.4204/EPTCS.251.22

Joseph Y. Halpern and Lenore D. Zuck. 1992. A Little Knowledge Goes a Long Way: Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols. *J. ACM* 39, 3 (1992), 449–478. https://doi.org/10.1145/146637.146638

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2017. IronFleet: proving safety and liveness of practical distributed systems. *Commun. ACM* 60, 7 (2017), 83–92. https://doi.org/10.1145/3068608

Gerard J. Holzmann. 2004. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley.

Hyperledger 2019. *Hyperledger*. https://github.com/hyperledger-labs

Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. 2003. Checking Cache-Coherence Protocols with TLA$^+$. *Formal Methods in System Design* 22, 2 (2003), 125–131. https://doi.org/10.1023/A:1022969405325

Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: resource-efficient byzantine fault tolerance. In *EuroSys '12*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 295–308. https://doi.org/10.1145/2168836.2168866

Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 279–296. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

Igor Konnov, Helmut Veith, and Josef Widder. 2015. SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms. In *CAV 2015 (LNCS)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 85–102. https://doi.org/10.1007/978-3-319-21690-4_6

Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017a. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 719–734. https://doi.org/10.1145/3009837

Igor V. Konnov, Helmut Veith, and Josef Widder. 2017b. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Inf. Comput.* 252 (2017), 95–109. https://doi.org/10.1016/j.ic.2016.03.006

Roman Krenický and Mattias Ulbrich. 2010. *Deductive Verification of a Byzantine Agreement Protocol.* Technical Report 2010-7. Karlsruhe Institute of Technology, Department of Computer Science. https://lfm.iti.kit.edu/english/769.php

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923. https://doi.org/10.1145/177492.177726

Leslie Lamport. 2004. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. 2017. Synthesis of Distributed Algorithms with Parameterized Threshold Guards, See [Aspnes et al. 2018], 32:1–32:20. https://doi.org/10.4230/LIPIcs.OPODIS.2017.32

Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *USENIX 2009*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 1–14. http://www.usenix.org/events/nsdi09/tech/full_papers/levin/levin.pdf

Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. 2011. Towards Verification of the Pastry Protocol Using TLA $^+$. In *FORTE 2011 (LNCS)*, Roberto Bruni and Jürgen Dingel (Eds.), Vol. 6722. Springer, 244–258. https://doi.org/10.1007/978-3-642-21461-5_16

Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 17–30. https://doi.org/10.1145/2976749.2978389

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Nancy A. Lynch and Mark R. Tuttle. 1987. Hierarchical Correctness Proofs for Distributed Algorithms. In *PODC 1987*, Fred B. Schneider (Ed.). ACM, 137–151. https://doi.org/10.1145/41840.41852

Ognjen Maric, Christoph Sprenger, and David A. Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *CAV 2017 (LNCS)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 217–237. https://doi.org/10.1007/978-3-319-63390-9_12

Dominique Méry and Neeraj Kumar Singh. 2011. Automatic code generation from event-B models. In *Symposium on Information and Communication Technology, SoICT 2011*, Huynh Quyet Thang and Dinh Khang Tran (Eds.). ACM, 179–188. https://doi.org/10.1145/2069216.2069252

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *LICS*. IEEE Computer Society, 14–23.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73. https://doi.org/10.1145/2699417

OCaml2C 2019. *Interfacing C with OCaml*. https://caml.inria.fr/pub/docs/manual-ocaml/intfc.html

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

Sam Owre, John M. Rushby, Natarajan Shankar, and Friedrich W. von Henke. 1995. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Trans. Software Eng.* 21, 2 (1995), 107–125. https://doi.org/10.1109/32.345827

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing liveness to safety in first-order logic. *PACMPL* 2, POPL (2018), 26:1–26:33. https://doi.org/10.1145/3158114

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* 1, OOPSLA (2017), 108:1–108:31. https://doi.org/10.1145/3140568

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 614–630. https://doi.org/10.1145/2908080.2908118

Prakash Panangaden and Kim Taylor. 1992. Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems. *Distributed Computing* 6, 2 (1992), 73–93. https://doi.org/10.1007/BF02252679

Rafael Pass and Elaine Shi. 2017. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *DISC 2017 (LIPIcs)*, Andréa W. Richa (Ed.), Vol. 91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 39:1–39:16. https://doi.org/10.4230/LIPIcs.DISC.2017.39

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2015. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. *ECEASST* 72 (2015). http://journal.ub.tu-berlin.de/eceasst/article/view/1013

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2017. EventML: Specification, Verification, and Implementation of Crash-Tolerant State Machine Replication Systems. *SCP* (2017).

Vincent Rahli, Nicolas Schiper, Robbert van Renesse, Mark Bickford, and Robert L. Constable. 2012. A diversified and correct-by-construction broadcast service. In *ICNP 2012*. IEEE Computer Society, 1–6. https://doi.org/10.1109/ICNP.2012.6459943

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *ESOP 2018 (LNCS)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. 2012. ShadowDB: A Replicated Database on a Synthesized Consensus Core. In *Eighth Workshop on Hot Topics in System Dependability (HotDep'12)*. http://www.nuprl.org/documents/Schiper/ShadowDB_A_Replicated_Database_on_a_Synthesized_Consensus_Core.pdf

Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. 2014. Developing Correctly Replicated Databases Using Formal Tools, See [DBL 2014], 395–406. https://doi.org/10.1109/DSN.2014.45

Ulrich Schmid, Bettina Weiss, and John M. Rushby. 2002. Formally Verified Byzantine Agreement in Presence of Link Faults. In *ICDCS*. 608–616. https://doi.org/10.1109/ICDCS.2002.1022311

SecureBlue 2019. *Secure Blue*. https://researcher.watson.ibm.com/researcher/view_page.php?id=6904

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. In *POPL 2018*.

SGX 2019. *SGX*. https://software.intel.com/en-us/sgx

João Sousa, Alysson Bessani, and Marko Vukolic. 2018. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *DSN 2018*. IEEE Computer Society, 51–58. https://doi.org/10.1109/DSN.2018.00018

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *PLDI 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 662–677. https://doi.org/10.1145/3192366.3192414

Joshua A. Tauber. 2004. *Verifiable Compilation of I/O Automata without Global Synchronization*. Ph.D. Dissertation. Departement of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

TrustZone 2019. *ARM TrustZone*. https://www.arm.com/products/security-on-arm/trustzone

Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

Tatsuhiro Tsuchiya and André Schiper. 2007. Model Checking of Consensus Algorithm. In *SRDS 2007*. IEEE Computer Society, 137–148. https://doi.org/10.1109/SRDS.2007.20

Tatsuhiro Tsuchiya and André Schiper. 2008. Using Bounded Model Checking to Verify Consensus Algorithms. In *DISC 2008 (LNCS)*, Gadi Taubenfeld (Ed.), Vol. 5218. Springer, 466–480. https://doi.org/10.1007/978-3-540-87779-0_32

Paulo Veríssimo. 2006. Travelling through wormholes: a new look at distributed systems models. *SIGACT News* 37, 1 (2006), 66–81. https://doi.org/10.1145/1122480.1122497

Paulo Veríssimo and Antonio Casimiro. 2002. The Timely Computing Base Model and Architecture. *IEEE Trans. Computers* 51, 8 (2002), 916–930. https://doi.org/10.1109/TC.2002.1024739

Paulo Veríssimo, Antonio Casimiro, and Christof Fetzer. 2000. The timely computing base: Timely actions in the presence of uncertain timeliness. In *DSN 2000*. IEEE Computer Society, 533–542. https://doi.org/10.1109/ICDSN.2000.857587

Guiliana Santos Veronese. 2010. *Intrusion Tolerance in Large Scale Networks*. Ph.D. Dissertation. Universidade de Lisboa.

Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. IEEE Computer Society, 10–19. https://doi.org/10.1109/HASE.2010.19

Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Computers* 62, 1 (2013), 16–30. https://doi.org/10.1109/TC.2011.221

Ivana Vukotic, Vincent Rahli, and Paulo Verissimo. 2019. Asphalion: Trustworthy Shielding Against Byzantine Faults. (2019). https://vrahli.github.io/articles/asphalion-long.pdf Extended version.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI 2015*, David Grove and Steve Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *CPP 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 154–165. https://doi.org/10.1145/2854065.2854081