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: Fault-tolerance, Byzantine faults, Hybrid protocols, MinBFT, Formal verification, Compositional reasoning, Coq, Knowledge calculus, Monad, Step-indexing

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 the correctness of these systems, relying on a minimal trusted computing base, and to the highest standards possible, e.g., using theorem provers. However, because current 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 [1].

One standard technique to mitigate this problem is to use Byzantine fault-tolerant state machine replication (BFT-SMR) [2, 3, 4] 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 [5], there is a non-negligible chance that they will later be found incorrect [6]. Adding on top of that the fact that many variants of these protocols

[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.

Authors' addresses: Ivana Vukotic, SnT, University of Luxembourg, ivana.vukotic@uni.lu; Vincent Rahli, University of Birmingham, vincent.rahli@gmail.com; Paulo Esteves-Veríssimo, SnT, University of Luxembourg, paulo.verissimo@uni.lu.

2019. XXXX-XXXX/2019/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

are being developed and adopted in critical sectors (e.g., in blockchain technology [7, 8, 9, 10, 11, 12]), it is clear that ensuring the correctness of these protocols is extremely important.

Moreover, because traditional BFT-SMR is extremely expensive,[2] "hybrid" architectures [15, 16, 17, 18, 19, 20, 21] 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 [22]. 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 [20, 23, 24, 25, 26, 27, 28, 29, 30], by relying on trusted-trustworthy components (e.g., message counters in MinBFT [25]) 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 [31, 32, 33, 34], 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 [35], 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 [25],[5] and managed to simplify some of the original proofs of those properties [37]. Verifying MinBFT-like protocols is important because: (1) MinBFT is part of other protocols, such as [27, 29]; (2) many protocols such as [26, 27, 25, 30] rely on the same kind of trusted components as MinBFT (see Sec. 7.1); and (3) to the best of our knowledge MinBFT's trusted components (called USIGs) have the smallest TCB compared to other trusted components used in contemporary hybrid protocols.

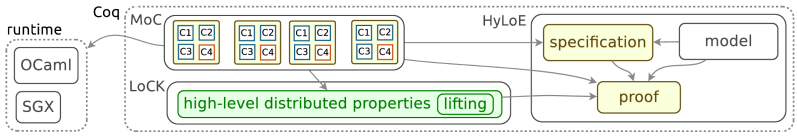
[2] Seminal BFT protocols such as [3, 13, 14] are expensive both in terms of the messages exchanged, and the required number of replicas, which in addition have to be diverse enough to enforce independence of failures.

[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 [25] is part of the Hyperledger Fabric umbrella [36].

Fig. 1 Overview of Asphalion



Contributions. To summarize, our contributions are as follows: (1) We introduce Asphalion, a generic and extensible Coq-based [38, 39] 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 verified within LoCK several reasoning patterns that are used to prove standard properties about 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 [24] (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, that enables running trusted components 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). In addition, Sec. 3 illustrates how it can be used to verify the correctness of fault-tolerant distributed systems. Asphalion provides three languages, which are based on extensions/variants of well-known and established formalisms: MoC is a component-based programming language, where components interact through a monad; HyLoE is based on Lamport's *happened before relation* [40]—one of the two main models of distributed systems, along with *distributed snapshots* [41]; and LoCK is a knowledge calculus, and, as discussed below, knowledge calculi have been shown over the years to provide convenient abstraction layers to reason about distributed systems without having to worry about low-level details.

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 monadic component language shallowly embedded into Coq.[6] A distributed system is a collection of local sub-systems, which are themselves collections of trusted/non-trusted sub-components. Fig. 1 depicts a system composed of 4 local sub-systems, each being composed of 4 components—3 non-trusted blue components, and 1 trusted in orange. Then, one has to provide a specification *Spec* (e.g. agreement) for *Sys* within HyLoE, our hybrid logic of events, which provides a model of distributed systems. 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: (1) using the general high-level distributed properties proved within our knowledge calculus

[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.

LoCK, 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. (we only cite a few relevant papers here): analyze distributed systems [42, 43, 44, 45, 46]; reason about synchronous systems [47, 48, 49, 50, 51, 52]; derive protocols [53]; synthesize systems [54]; and reason about blockchain protocols [55]. 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 [56], 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 [57, 42]).

2.3 Rationale for Designing Asphalion

As it turns out, Asphalion is not a simple extension of Velisarios, but is inspired by and uses part of Velisarios. Starting from the foundations of Velisarios (its logic of events), we designed an entirely new 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, in Velisarios, a local state machine is essentially a single component. To add axioms

[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.

about trusted components to it, we would first need the notion of components, which is why we developed MoC (see Sec. 5). MoC allows implementing distributed systems 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 the non-trusted components of processes to misbehave, while the trusted components keep following their specification. We captured this by changing the semantics of events (namely the trigger function described in Sec. 4.3) to also handle events at which a trusted component of a compromised process is 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 knowledge library in several ways. 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 (thanks to the fact that it is deeply embedded in Coq), which does not exist in Velisarios's simple knowledge library. In addition, LoCK allows reasoning at a high-level of abstraction about trusted and non-trusted knowledge (among other things), while Velisarios's knowledge 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 Byzantine 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) As discussed in Sec. 5, MoC 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) As discussed in Sec. 4, HyLoE is a logic 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) as discussed in Sec. 6, LoCK is a high-level

[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.

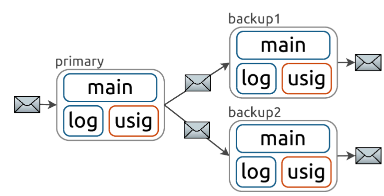
knowledge calculus deeply embedded in Coq, whose expressiveness is constrained by its inference rules. We leave studying LoCK's proof-theoretic strength for future work. LoCK is optional but recommended because: (a) it allows stating system properties at a high-level of abstraction, without having to worry about 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 $A \rightarrow B$ 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 $\boldsymbol{a} \in \{$primary, backup1, backup2$\}$, returns a local sub-system ($\boldsymbol{a}$'s code). Each local sub-system is composed of three components (state machines), namely, a main component called main and two sub-components: (1) a log called log containing all received/generated requests; and (2) a trusted message counter, called usig, similar to the one used in MinBFT (Sec. 7.1 elaborates on MinBFT and its trusted USIG component).

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—the request along with the sequence number are signed by the trusted component 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 signature of the pair $\langle r, i\rangle$ generated by its usig component. 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), 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 $\boldsymbol{a}$ 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.

```
λs, m.𝕀( match m with
       | request(_) ⇒ handleRequest(a, s, m)
       | commit(_, _) ⇒ handleCommit(a, s, m)
       | accept(_, _) ⇒ handleAccept(a, s, m) )
```

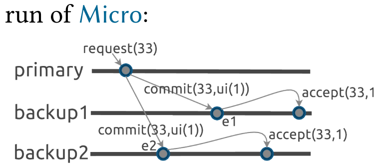
Let us explain what it does. As discussed in Sec. 5.4, the three handlers are expressed in a deep embedding of a simple language, which is more amenable to automation than our general monadic programming language shallowly embedded in Coq (and therefore rather unwieldy).[10] $\mathbb{I}(\_)$ lifts processes 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.

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, $\boldsymbol{a}$ is a backup, and $\boldsymbol{a}$ 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 the 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:

$$
\begin{aligned}
&\mathsf{Lemma}\ \mathsf{micro\_agreement}\ :\\
&\forall (eo : \mathsf{EO})(e_1, e_2 : \mathsf{Event}(eo))(r_1, r_2 : \mathsf{Request})(i : \mathbb{N}).\\
&\quad \mathsf{accept}(r_1, i) \in \mathsf{Micro} \rightsquigarrow e_1\\
&\quad \rightarrow \mathsf{accept}(r_2, i) \in \mathsf{Micro} \rightsquigarrow e_2\\
&\quad \rightarrow r_1 = r_2
\end{aligned}
$$



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) 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

[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.

[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.

captured by eo. Therefore, this property states that *in each possible run of* Micro, if it outputs two messages of the form $\mathtt{accept}(r_1, i)$ and $\mathtt{accept}(r_2, i)$ 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, $\mathtt{accept}(r_1, i) \in \mathsf{Micro} \rightsquigarrow e_1$ states that $\mathtt{accept}(r_1, i)$ belongs to Micro's outputs at e_1.

As discussed in Sec. 4, one particularity of HyLoE is that it allows reasoning about the behavior of trusted components running at compromised locations. In our example here, it allows reasoning about usig components even though the main and log components might have been compromised. In general, to prove a system property, one has to prove that it holds for all event orderings, even the ones where the events happen at compromised nodes where only the trusted components are still running. As it turns out, micro_agreement is true even for the runs where the primary, except for its usig component, has been compromised.

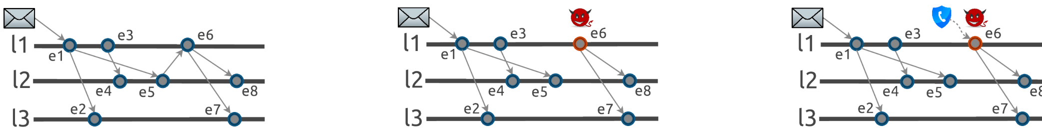
Micro's verification using LoCK. One could prove Micro's agreement property using only HyLoE, i.e., using only HyLoE's axioms and properties of the above mentioned constructs to reason about systems' inputs, states and outputs. However, in general we recommend to use LoCK instead for two main reasons. (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 a simple 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 is presented in See 6.2, and its semantics in Sec. 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 regarding this proof) 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: a USIG-based MinBFT, and a TrInc-based MinBFT). Roughly speaking, 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 get 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. We rely on 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

Fig. 2 Examples of message sequence diagrams



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

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 do not need to be 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 it in the past; or (2) we just learned it from someone else; 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 a new hybrid variant of the Logic of Events (LoE) that was originally introduced in [58] to reason about crash fault tolerant protocols [59, 60, 61], and later used to reason about cyber-physical systems [62]. LoE was then extended in [35] to reason about Byzantine fault tolerant protocols. We extend LoE further here to enable reasoning about hybrid fault models and hybrid protocols (i.e., protocols that 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 below. First, we start by introducing basic concepts such as names, messages, etc., which we use later to define our new Hybrid LoE (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 later for a given protocol. One of HyLoE's parameters is a type Node of node names, ranged over by $\boldsymbol{a}$. 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 composed of sub-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 [40] at which something happened. In EventML [59, 60, 61] events are abstract objects that only correspond to the handling of a message by a node that follows its specification (kind 1—see Fig. 2a). As opposed to EventML, in Velisarios [35], 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

Fig. 3 HyLoE parameters

Types:	Event (ranged over by e)	AuthData (ranged over by $auth$)	Keys (ranged over by ks)
Functions:	$\prec \in$ Event $\to$ Event $\to \mathbb{P}$ loc $\in$ Event $\to$ Node	trigger $\in$ Event $\to$ TriggerInfo pred $\in$ Event $\to$ option(Event)	keys $\in$ Event $\to$ Keys nfo2auth $\in$ TriggerInfo $\to$ list(AuthData)

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

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):

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

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*.[12] 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).[13]

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)).[14] 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: $\mathsf{pred}(e)$ returns e's local direct predecessor, if it exists. As in Velisarios [35, 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, 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, which one can instantiate for example using RSA signatures) that has been generated using keys (which one can instantiate for example using RSA keys). Keys are associated

[12] *Event orderings* formalize the *message sequence diagrams* used by system designers to describe the behavior of systems.
[13] 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.
[14] Our model is based on Lamport's happened before relation [40], as opposed to the "global state" semantics [41].

with nodes as follows: $\mathsf{keys}(e)$ returns the keys available at e. Finally, $\mathsf{nfo2auth}(nfo)$ 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 it 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 specification statements (e.g., for fault-tolerant protocols, 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 $\mathsf{first?}(e)$ be $\mathtt{true}$ iff $\mathsf{pred}(e) = \mathtt{None}$; let $e_1 \subset e_2$ be $\mathsf{pred}(e_2) = \mathtt{Some}(e_1)$; let $\mathsf{pred}^{=}(e)$ 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 $e_1 \prec e_2 \wedge \mathsf{loc}(e_1) = \mathsf{loc}(e_2)$; and let $e_1 \sqsubseteq e_2$ be $e_1 \preceq e_2 \wedge \mathsf{loc}(e_1) = \mathsf{loc}(e_2)$.

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 $\mathsf{CompName}$ 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 $\mathsf{InputTrusted}$—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, Local and Distributed Systems

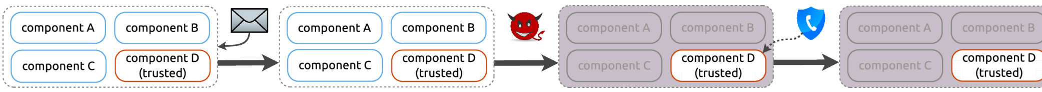
Components. A component is a named state machine, which essentially consists of an update function and the current state of the machine. To support the fact that components are allowed to call each other, we define state machines using a state monad [63]. Therefore, instead of traditionally defining update functions as functions that take an input and a state and return an output and an updated state, we combine those with a monad (see $\mathsf{M}^n(T)$'s definition below), such that in addition update functions take components as input and return possibly modified components. Consequently, state machines can call other state machines through this state monad. Therefore, to avoid a circularity in the definition of state machines, we now use step indexing [64] to define them, requiring that machines at level n can only use machines of lower levels. Let $\mathsf{Component}^n$ (ranged over by $comp$) be the collection of components at level n, which we define recursively over n below. This definition uses the monad mentioned above, which looks like this (where T is a type):

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

Going back to state machines, a machine at level $n+1$ (of type $\mathsf{Component}^{n+1}$—by definition there are no level 0 machines) with name cn is either a state machine at level n, or a pair of: (1) an update function of type $\mathsf{Upd}^n(cn) = \mathcal{S}(cn) \rightarrow \mathcal{I}(cn) \rightarrow \mathsf{M}^n(\mathcal{S}(cn) * \mathcal{O}(cn))$; and (2) a state of type $\mathcal{S}(cn)$.[15]

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

Fig. 4 An execution of a local system



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 $m \gg\!= f = (\lambda s.\mathtt{let}\ s', a = m(s)\ \mathtt{in}\ f(a, s'))$ 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 $subs_2$.[16]

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 $\to$ 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 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 list of 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 preceding events has been triggered by some information of the form $\mathtt{TItrust}(it)$ (in case the trusted component[17] 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 preceding events is a Byzantine event and ls does not include a trusted component. Fig. 4 provides an example of 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. As shown in Fig. 4, in case one of those preceding events is Byzantine, $ls@^{-}e$ keeps on running the trusted component because it cannot be compromised. However, $ls@^{-}e$ loses track of the rest of the system since a Byzantine event has occurred, and the other non-trusted components could be in any state.

[16]See Appx. A for an example of a local system and of how call works.

[17]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.

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 $\mathsf{trigger}$; 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 $\mathtt{accept}(r, i) \in \mathsf{Micro} \rightsquigarrow e$, 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 $\mathtt{accept}(r, i) \in \mathsf{Micro} \rightsquigarrow e$, 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, (2) we compose this proof 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: one can build a component essentially from any update function of type $\mathsf{Upd}^{n}(cn)$. Indeed, our framework provides a shallow embedding of components that can make use of any Coq expression as long as it has the right type. Unfortunately, this is also sometimes a disadvantage because it entails that we cannot prove many general lemmas about the behavior of components. For example, a component could simply throw away all its sub-components. However, often components simply use their sub-components, and return them updated. This is useful information, which we would like to easily derive. A standard technique to prove such generic results about such "well-behaved" components is to: (1) define a deep embedding of these "well-behaved" components; (2) define an "interpretation" function from the deep embedding to the shallow embedding; and (3) prove that these generic properties hold for the deep embedding.

One can define as many deep embeddings as needed. We define here a simple one (which we used to implement MinBFT) that contains only three operators: return/bind/call.[18] Namely, let $\mathsf{Proc}(A)$ 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

[18] Appx. B presents another example of such a language that also allows spawning new sub-components.

level n) be the following interpretation of this language (right):

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

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,[19] which we in turn used to prove that our MinBFT implementations satisfy the **Mon** property presented in Eq. 4 in Sec. 6.6:[20]

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 $ls@^{+}e\downharpoonleft_{cn}$ is obtained by running cn on l, starting from the state $ls@^{-}e\downharpoonleft_{cn}$.*

Remark 1. *Trusted components do not need to be at level 1. However, this constraint in Thm. 5.1 is convenient to obtain a simple lifting theorem. Otherwise, without this constraint, i.e., for higher-level components, such a local lifting theorem would be more complicated because it would have to also take into consideration the sub-components such higher-level components use to compute their new state. 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 $ls@^{+}e\downharpoonleft_{cn}$, 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 [23, 24, 25] are level 1 components. Therefore, we leave developing such 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,[21] which further opens the door to automation.[22] Moreover, unlike in Velisarios, LoCK enables reasoning about both trusted and non-trusted knowledge. First, Sec. 6.1 introduces the parameters

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

Fig. 5 LoCK's parameters

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 $\rightarrow$ Identifier $\rightarrow \mathbb{P}$	verify $\in$ Event $\rightarrow$ AuthData $\rightarrow \mathbb{B}$
	mem $\in$ CompName	genFor $\in$ Data $\rightarrow$ Trust $\rightarrow \mathbb{P}$	trusted2id $\in \mathcal{S}$(trust) $\rightarrow$ Identifier
	trust $\in$ CompName	know $\in$ Data $\rightarrow \mathcal{S}$(mem) $\rightarrow \mathbb{P}$	lt $\in$ Identifier $\rightarrow$ Identifier $\rightarrow \mathbb{P}$
	owner $\in$ Data $\rightarrow$ Node	auth2data $\in$ AuthData $\rightarrow$ list(Data)	initId $\in$ Identifier

Axioms:
(1) lt is transitive and anti-reflexive
(2) know(d, m) is decidable
(3) $\forall t, d_1, d_2$.genFor(d_1, t) $\rightarrow$ genFor(d_2, t) $\rightarrow d_1 = d_2$
(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. 6 LoCK's syntax

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

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

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

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[23] (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 ($\top$, $\bot$, $\wedge$, $\vee$, $\rightarrow$, $\exists$, $\forall$), 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

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

Fig. 7 LoCK's semantics (predicate logic)

$$\llbracket\top\rrbracket_e = \mathsf{True} \qquad \llbracket\bot\rrbracket_e = \mathsf{False}$$

$$\llbracket\tau_1 \wedge \tau_2\rrbracket_e = \llbracket\tau_1\rrbracket_e \wedge \llbracket\tau_2\rrbracket_e \qquad \llbracket\tau_1 \vee \tau_2\rrbracket_e = \llbracket\tau_1\rrbracket_e \vee \llbracket\tau_2\rrbracket_e \qquad \llbracket\tau_1 \rightarrow \tau_2\rrbracket_e = \llbracket\tau_1\rrbracket_e \rightarrow \llbracket\tau_2\rrbracket_e$$

$$\llbracket\exists\phi\rrbracket_e = \exists v \in \{v \in \mathsf{KVal} \mid \mathsf{oftype}(v, \phi.\mathbf{1})\}.\phi.\mathbf{2}(v) \qquad \llbracket\forall\phi\rrbracket_e = \forall v \in \{v \in \mathsf{KVal} \mid \mathsf{oftype}(v, \phi.\mathbf{1})\}.\phi.\mathbf{2}(v)$$

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

$$\llbracket\odot\rrbracket_e = \mathsf{first?}(e) = \mathsf{true} \qquad \llbracket @(a)\rrbracket_e = \mathsf{loc}(e) = a$$

$$\llbracket\subset\tau\rrbracket_e = \begin{cases} \llbracket\tau\rrbracket_{e'}, \text{ if } \mathsf{pred}(e) = \mathsf{Some}(e') \\ \mathsf{False} \text{ otherwise} \end{cases}$$

$$\llbracket\prec\tau\rrbracket_e = \exists e' \prec e.\llbracket\tau\rrbracket_{e'} \qquad \llbracket\sqsubset\tau\rrbracket_e = \exists e' \sqsubset e.\llbracket\tau\rrbracket_{e'}$$

Fig. 9 LoCK's semantics (knowledge)

$$\llbracket\mathcal{L}(d)\rrbracket_e = \mathsf{learns}(e, d) \qquad \llbracket\mathcal{D}(d)\rrbracket_e = d \in \mathsf{sys} \rightsquigarrow e \qquad \llbracket\mathcal{K}^+(d)\rrbracket_e = \mathsf{knows}^+(e, d)$$

$$\llbracket\mathcal{I}^+(i)\rrbracket_e = \mathsf{ident}^+(e, i) \qquad \llbracket v_1 = v_2\rrbracket_e = v_1 = v_2 \qquad \llbracket i_1 < i_2\rrbracket_e = \mathsf{lt}(i_1, i_2)$$

$$\llbracket\mathcal{HI}(t, i)\rrbracket_e = \mathsf{trustHasId}(t, i) \qquad \llbracket\mathcal{O}(d, a)\rrbracket_e = \mathsf{owner}(d) = a \qquad \llbracket\mathcal{G}(d, t)\rrbracket_e = \mathsf{genFor}(d, t)$$

the form $\forall\phi$ takes a dependent pair ϕ as argument: (1) a type θ and (2) a function from values of type θ to expressions. The predicate $\mathsf{oftype}(v, \theta)$ is true iff $(v, \theta) \in \{(i, \mathtt{KTi}), (d, \mathtt{KTd}), (t, \mathtt{KTt}), (a, \mathtt{KTn})\}$.

LoCK also provides general operators to capture properties about distributed knowledge. Reasoning about distributed knowledge is a well studied topic [42, 43, 44, 45, 53, 46, 54, 49, 50, 55, 51, 52]. However, as opposed to the papers listed above, 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 $^-$ and the other with $^+$. The ones annotated with $^-$ are used to state properties about the knowledge of a system right before handling an event, and are defined below; while the ones annotated with $^+$ are used to state properties once events have been handled, and are primitives of the language.

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

$$\preceq\tau = \prec\tau \vee \tau \qquad \sqsubseteq\tau = \sqsubset\tau \vee \tau \qquad \subseteq\tau = \subset\tau \vee (\tau \wedge \odot)$$

$$\mathcal{K}^-(\tau) = \subset\mathcal{K}^+(\tau) \qquad \mathcal{I}^-(i) = \subset\mathcal{I}^+(i) \vee (i = \mathsf{initId} \wedge \odot) \qquad i_1 \leq i_2 = i_1 < i_2 \vee i_1 = i_2$$

$$\mathcal{O}(d) = \exists_{\mathtt{n}}\lambda a.@(a) \wedge \mathcal{O}(d, a) \qquad \mathcal{OD}(d) = \mathcal{O}(d) \wedge \mathcal{D}(d)$$

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

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

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

Fig. 10 Syntax of knowledge calculus rules

$x \in \mathsf{HypName}$	(a set of hypothesis names)		$y \in \mathsf{GuardName}$	(a set of guard names)	
$\sigma \in \mathsf{KExpAt}$	$::=$	$\tau \ @\ e$	$\alpha \in \mathsf{EventRel}$	$::=$	$e_1 \equiv e_2 \mid e_1 \subset e_2 \mid e_1 \prec e_2 \mid e_1 \preceq e_2 \mid e_1 \sqsubset e_2 \mid e_1 \sqsubseteq e_2$
$h \in \mathsf{Hyp}$	$::=$	$x : \sigma$	$H \in \mathsf{Hyps}$	$::=$	$\emptyset \mid H, h$
$g \in \mathsf{Guard}$	$::=$	$y : \alpha$	$G \in \mathsf{Guards}$	$::=$	$\emptyset \mid G, g$
$seq \in \mathsf{Sequent}$	$::=$	$\langle G \rangle\, H \vdash \sigma$	$R \in \mathsf{Rule}$	$::=$	$\dfrac{\Lambda[\overline{e}, \overline{t}, \overline{i}] \quad seq_1 \quad \cdots \quad seq_n}{seq}$

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

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

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:

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

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).[24] For convenience, hypotheses and guards are all named in a sequent, allowing rules to point to them (expressions do not depend on names). We write H_1, H_2 for the list H_1 appended with the list H_2, and similarly for guards. A rule R is essentially a pair of a list of sequents (R's hypotheses) and a sequent (R's conclusion). In addition, the hypotheses of a rule can depend on a list of events $\overline{e}$, a list of trusted values $\overline{t}$, and a list of trusted identifiers $\overline{i}$, allowing rules to introduce new symbols. We omit the $\Lambda[\_]$ part in rules that do not introduce new symbols. We sometime write $H[\sigma]$, for a list of hypotheses H that contains an hypothesis of the form $x : \sigma$, and similarly for guards. We then sometimes write $H[\sigma']$ to denote the same list of hypotheses where $x : \sigma$ is replaced by $x : \sigma'$.

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

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

where $(\Box, \circ) \in \{(\sqsubset, \sqsubset), (\sqsubseteq, \sqsubseteq), (\prec, <), (\preceq, \leq), (\subset, \subset), (\equiv, =)\}$. Note that $\Box$ 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 standard structural and predicate logic rules are presented in 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 since they are all straightforward).

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

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

Fig. 11 LoCK's event relation rules

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

$$\frac{\Lambda[e'] \quad \langle G, y : e' \square e \rangle\, H[x : \tau \mathbin{@} e'] \vdash \sigma}{\langle G \rangle\, H[x : \square\tau \mathbin{@} e] \vdash \sigma}\ \square_{\mathsf{E}} \qquad \frac{\langle G[e' \square e] \rangle\, H \vdash \tau \mathbin{@} e'}{\langle G[e' \square e] \rangle\, H \vdash \square\tau \mathbin{@} e}\ \square_{\mathtt{I}} \qquad \frac{\langle G[e' \square e] \rangle\, H \vdash \square\tau \mathbin{@} e'}{\langle G[e' \square e] \rangle\, H \vdash \square\tau \mathbin{@} e}\ \square_{\mathtt{It}}$$

$$\frac{\langle G, y : \mathsf{pred}^{=}(e) \subset e \rangle\, H \vdash \sigma \qquad \langle G \rangle\, H \vdash \neg\odot \mathbin{@} e}{\langle G \rangle\, H \vdash \sigma}\ \mathtt{if}\neg\odot \qquad \frac{\langle G, y : \mathsf{pred}^{=}(e) \equiv e \rangle\, H \vdash \sigma \qquad \langle G \rangle\, H \vdash \odot \mathbin{@} e}{\langle G \rangle\, H \vdash \sigma}\ \mathtt{if}\odot \qquad \frac{\langle G[e' \blacktriangleleft e] \rangle\, H \vdash \sigma}{\langle G[e' \lhd e]\boldsymbol{o} \rangle\, H \vdash \sigma}\ \mathsf{weak}$$

$$\frac{\langle G[e_1 \equiv e_2] \rangle\, H[\tau \mathbin{@} e_2] \vdash \sigma}{\langle G[e_1 \equiv e_2] \rangle\, H[\tau \mathbin{@} e_1] \vdash \sigma}\ \mathsf{sub}_{\mathsf{H}} \qquad \frac{\langle G[y : e_1 \equiv e_2] \rangle\, H \vdash \tau \mathbin{@} e_1}{\langle G[y : e_1 \equiv e_2] \rangle\, H \vdash \tau \mathbin{@} e_2}\ \mathsf{sub}_{\mathsf{C}} \qquad \frac{\langle G, y : e \equiv e \rangle\, H \vdash \sigma}{\langle G \rangle\, H \vdash \sigma}\ \equiv_{\mathtt{refl}}$$

Fig. 12 LoCK's logic of events rules

$$\frac{}{\langle G[e_1 \sqsubset e_2] \rangle\, H \vdash \neg\odot \mathbin{@} e_2}\ \neg\odot \qquad \frac{}{\langle G \rangle\, H \vdash \odot \vee \neg\odot \mathbin{@} e}\ \odot_{\mathsf{dec}}$$

$$\frac{\Lambda[e'] \quad \langle G, y : e' \sqsubseteq e \rangle\, H \vdash \odot \rightarrow \tau \mathbin{@} e' \qquad \langle G, y : e' \sqsubseteq e \rangle\, H \vdash \subset\tau \rightarrow \tau \mathbin{@} e'}{\langle G \rangle\, H \vdash \tau \mathbin{@} e}\ \mathtt{ind} \qquad \frac{\langle G \rangle\, H \vdash @(\boldsymbol{a}) \mathbin{@} e_1 \quad \langle G \rangle\, H \vdash @(\boldsymbol{a}) \mathbin{@} e_2 \quad \langle G, y : e_1 \equiv e_2 \rangle\, H \vdash \sigma \quad \langle G, y : e_1 \sqsubset e_2 \rangle\, H \vdash \sigma \quad \langle G, y : e_2 \sqsubset e_1 \rangle\, H \vdash \sigma}{\langle G \rangle\, H \vdash \sigma}\ \mathtt{tri}$$

Fig. 13 LoCK's knowledge rules

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

$$\frac{}{\langle G \rangle\, H \vdash \mathcal{K}^{+}(d) \vee \neg\mathcal{K}^{+}(d) \mathbin{@} e}\ \mathsf{K}_{\mathsf{dec}}$$

$$\frac{\langle G \rangle\, H \vdash v_2 = v_1 \mathbin{@} e}{\langle G \rangle\, H \vdash v_1 = v_2 \mathbin{@} e}\ \mathtt{sym} \qquad \frac{\langle G \rangle\, H \vdash i_1 \ \pi\ i \mathbin{@} e \qquad \langle G \rangle\, H \vdash i \ \kappa\ i_2 \mathbin{@} e}{\langle G \rangle\, H \vdash i_1 \ \rho\ i_2 \mathbin{@} e}\ \mathtt{trans} \qquad \frac{}{\langle G \rangle\, H[i < i] \vdash \sigma}\ \mathtt{irrefl}$$

$$\frac{\langle G \rangle\, H \vdash \boldsymbol{O}(d, \boldsymbol{a}_1) \mathbin{@} e \quad \langle G \rangle\, H \vdash \boldsymbol{O}(d, \boldsymbol{a}_2) \mathbin{@} e}{\langle G \rangle\, H \vdash \boldsymbol{a}_1 = \boldsymbol{a}_2 \mathbin{@} e}\ \mathtt{1owner} \qquad \frac{\langle G \rangle\, H \vdash \boldsymbol{\mathcal{G}}(d_1, t) \mathbin{@} e \quad \langle G \rangle\, H \vdash \boldsymbol{\mathcal{G}}(d_2, t) \mathbin{@} e}{\langle G \rangle\, H \vdash d_1 = d_2 \mathbin{@} e}\ \mathtt{1data} \qquad \frac{\langle G \rangle\, H \vdash \boldsymbol{\mathcal{I}}^{+}(i_1) \mathbin{@} e \quad \langle G \rangle\, H \vdash \boldsymbol{\mathcal{I}}^{+}(i_2) \mathbin{@} e}{\langle G \rangle\, H \vdash i_1 = i_2 \mathbin{@} e}\ \mathtt{1id}$$

and $\mathtt{if}\odot$ provide an axiomatization of $\mathsf{pred}^{=}$. The weak family of rules allows weakening guards, e.g., from $\prec$ to $\preceq$ (strengthening rules are presented in Appx. C). Finally, the substitution rules $\mathsf{sub}_{\mathsf{H}}$ and $\mathsf{sub}_{\mathsf{C}}$ allow substituting events in a sequent's hypotheses and conclusion.

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 $\boldsymbol{a}$, 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.[25]

[25] We omit here the $\Lambda[\_]$ part for readability. Moreover, we use some standard rules such as $\rightarrow_{\mathsf{E}}$ (implication elimination); $\vee_{\mathsf{E}}$ (or elimination); $\vee_{\mathtt{Il}}/\vee_{\mathtt{Ir}}$ (or introduction left/right); or hyp (hypothesis rule), which are described in 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 figure on the right).[26] In this first example, we only navigate between events in the hypothesis x: we use the $\Box_{\mathsf{E}}$ 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.

$$
\AxiomC{}
\RightLabel{$\neg\odot$}
\UnaryInfC{$\langle y : e'\sqsubset e\rangle\ x : \tau \ @\ e' \vdash \neg\odot \ @\ e$}
\RightLabel{$\Box_{\mathsf{E}}$}
\UnaryInfC{$\langle \oslash\rangle\ x : \sqsubset\tau \ @\ e \vdash \neg\odot \ @\ e$}
\RightLabel{$\to_{\mathsf{E}}$}
\UnaryInfC{$\langle \oslash\rangle\ \oslash \vdash \sqsubset\tau \to \neg\odot \ @\ e$}
\DisplayProof
$$

Collapsing. We now prove another simple, though slightly more involved, example (see figure 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.[27] We use the $\Box_{\mathsf{E}}$ elimination rules twice to go from the point where $\sqsubset\sqsubset\tau$ is true (i.e., e), to the point where τ is true (i.e., e''). We then use the $\Box_{\mathtt{It}}$ introduction rule to navigate to the e' intermediary point. Finally, we use the $\Box_{\mathtt{I}}$ introduction rule to navigate to e'', while eliminating $\sqsubset$ (as opposed to the previous step, which keeps the operator).

$$
\AxiomC{}
\RightLabel{hyp}
\UnaryInfC{$\langle y : e'\sqsubset e, y' : e''\sqsubset e'\rangle\ x : \tau \ @\ e'' \vdash \tau \ @\ e''$}
\RightLabel{$\Box_{\mathtt{I}}$}
\UnaryInfC{$\langle y : e'\sqsubset e, y' : e''\sqsubset e'\rangle\ x : \tau \ @\ e'' \vdash \sqsubset\tau \ @\ e'$}
\RightLabel{$\Box_{\mathtt{It}}$}
\UnaryInfC{$\langle y : e'\sqsubset e, y' : e''\sqsubset e'\rangle\ x : \tau \ @\ e'' \vdash \sqsubset\tau \ @\ e$}
\RightLabel{$\Box_{\mathsf{E}}$}
\UnaryInfC{$\langle y : e'\sqsubset e\rangle\ x : \sqsubset\tau \ @\ e' \vdash \sqsubset\tau \ @\ e$}
\RightLabel{$\Box_{\mathsf{E}}$}
\UnaryInfC{$\langle \oslash\rangle\ x : \sqsubset\sqsubset\tau \ @\ e \vdash \sqsubset\tau \ @\ e$}
\RightLabel{$\to_{\mathsf{E}}$}
\UnaryInfC{$\langle \oslash\rangle\ \oslash \vdash \sqsubset\sqsubset\tau \to \sqsubset\tau \ @\ e$}
\DisplayProof
$$

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:[28]

$$
\AxiomC{$\Lambda[e']\ \langle G, y : e'\sqsubseteq e\rangle\, H[x : \tau \ @\ e'] \vdash \sigma$}
\RightLabel{weak}
\UnaryInfC{$\Lambda[e']\ \langle G, y : e'\sqsubset e\rangle\, H[x : \tau \ @\ e'] \vdash \sigma$}
\RightLabel{$\Box_{\mathsf{E}}$}
\UnaryInfC{$\langle G\rangle\, H[x : \sqsubset\tau \ @\ e] \vdash \sigma$}
\AxiomC{$\Lambda[e']\ \langle G, y : e'\sqsubseteq e\rangle\, H[x : \tau \ @\ e'] \vdash \sigma$}
\UnaryInfC{$\langle G, y : e\sqsubseteq e\rangle\, H[x : \tau \ @\ e'] \vdash \sigma$}
\RightLabel{weak}
\UnaryInfC{$\langle G, y : e \equiv e\rangle\, H[x : \tau \ @\ e'] \vdash \sigma$}
\RightLabel{$\equiv_{\mathtt{refl}}$}
\UnaryInfC{$\langle G\rangle\, H[x : \tau \ @\ e] \vdash \sigma$}
\RightLabel{$\vee_{\mathsf{E}}$}
\BinaryInfC{$\langle G\rangle\, H[x : \sqsubseteq\tau \ @\ e] \vdash \sigma$}
\DisplayProof
$$

Predecessor. Next, we prove that if τ was true at $\mathsf{pred}^{=}(e)$ (denoted e_p below) then it must be that τ happened before or at e.[29] 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 $\odot_{\mathsf{dec}}$); 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:

$$
\AxiomC{Π_1}
\AxiomC{}
\RightLabel{hyp}
\UnaryInfC{$\langle y : e_p \equiv e\rangle\ x : \tau \ @\ e, o : \odot \ @\ e \vdash \tau \ @\ e$}
\RightLabel{$\mathsf{sub}_{\mathsf{H}}$}
\UnaryInfC{$\langle y : e_p \equiv e\rangle\ x : \tau \ @\ e_p, o : \odot \ @\ e \vdash \tau \ @\ e$}
\AxiomC{Π_2}
\RightLabel{$\mathtt{if}\odot$}
\BinaryInfC{$\langle \oslash\rangle\ x : \tau \ @\ e_p, o : \odot \ @\ e \vdash \tau \ @\ e$}
\RightLabel{$\vee_{\mathtt{Ir}}$}
\UnaryInfC{$\langle \oslash\rangle\ x : \tau \ @\ e_p, o : \odot \ @\ e \vdash \sqsubseteq\tau \ @\ e$}
\AxiomC{}
\RightLabel{hyp}
\UnaryInfC{$\langle y : e_p\sqsubset e\rangle\ x : \tau \ @\ e_p, o : \neg\odot \ @\ e \vdash \tau \ @\ e_p$}
\RightLabel{$\Box_{\mathtt{I}}$}
\UnaryInfC{$\langle y : e_p\sqsubset e\rangle\ x : \tau \ @\ e_p, o : \neg\odot \ @\ e \vdash \sqsubset\tau \ @\ e$}
\RightLabel{weak}
\UnaryInfC{$\langle y : e_p\subset e\rangle\ x : \tau \ @\ e_p, o : \neg\odot \ @\ e \vdash \sqsubset\tau \ @\ e$}
\AxiomC{Π_3}
\RightLabel{$\mathtt{if}\neg\odot$}
\BinaryInfC{$\langle \oslash\rangle\ x : \tau \ @\ e_p, o : \neg\odot \ @\ e \vdash \sqsubset\tau \ @\ e$}
\RightLabel{$\vee_{\mathtt{Il}}$}
\UnaryInfC{$\langle \oslash\rangle\ x : \tau \ @\ e_p, o : \neg\odot \ @\ e \vdash \sqsubseteq\tau \ @\ e$}
\RightLabel{$\vee_{\mathsf{E}}$}
\BinaryInfC{$\langle \oslash\rangle\ x : \tau \ @\ e_p, o : \odot \vee \neg\odot \ @\ e \vdash \sqsubseteq\tau \ @\ e$}
\RightLabel{cut}
\BinaryInfC{$\langle \oslash\rangle\ x : \tau \ @\ e_p \vdash \sqsubseteq\tau \ @\ e$}
\DisplayProof
$$

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

[26] See `DERIVED_RULE_local_before_implies_not_first_true` in `model/CalculusSM_derived3.v`.
[27] See `DERIVED_RULE_twice_local_before_implies_once_true` in `model/CalculusSM_derived3.v`.
[28] See `DERIVED_RULE_unlocal_before_eq_hyp_true` in `model/CalculusSM.v`.
[29] 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'.[30] 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 $\mathsf{K}_{\mathtt{dec}}$, 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:

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

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

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

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

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

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:[31] $\mathcal{K}^{+}(t) \to \sqsubseteq\mathcal{L}(t) \vee \sqsubseteq\mathcal{OD}(t)$. 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:[32]

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

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:[33]

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

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 result

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

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

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

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

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

Using the above mentioned rules and assumptions, we derived among other things the following lemma (see below for a proof sketch):[34]

Theorem 6.1 (Distributed Lifting). *The following rule is derivable within LoCK:*

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

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* `ldata` *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

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

$\langle r_1, i\rangle$, 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. $[\![\mathcal{G}(\langle r_2, i\rangle, ui_2)]\!]_{e_2}$. 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. $[\![\mathcal{O}(ui_1, \mathsf{primary})]\!]_{e_1}$ and $[\![\mathcal{O}(ui_2, \mathsf{primary})]\!]_{e_2}$. 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.[35]

7 CASE STUDIES: USIG- AND TRINC-BASED MINBFT

We exercised Asphalion by implementing and verifying two versions of the seminal MinBFT hybrid protocol [25]: one based on USIGs (as in the original version), and one based on TrIncs [24]. As discussed below, USIGs and TrIncs have different pros and cons that make them both interesting to use and verify correct. We proved the agreement property of both versions using 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 [27, 30, 23]. We now present MinBFT (see [25, 37] for further details), starting with a description of the trusted components our implementations rely on.

7.1 MinBFT Recap

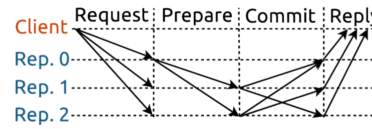
```
function createUI(msg) : UI {
  counter++;
  H:=hash(msg,id,counter,keys);
  return⟨id,counter,H⟩; }

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

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.

TrInc. In [24], the authors introduced a new kind of trusted components called TrInc (which stands for Trusted Incrementer). 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 counter value is provided by the client of the trusted component and has to be greater than the current value (see [24] 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 [30] to be based on TrInc instead of USIG.

[35] For example, as discussed in Appx. D, we have also proved the crux of Micro's validity property using a general high-level LoCK lemma.

MinBFT details. As other such protocols do, MinBFT works in a succession of configurations called *views*. In each view v, the distinguished replica $p = v \ mod \ n$ (n is the total number of replicas), called the *primary*, is in charge of ordering client requests by assigning sequence numbers (the counter values generated by its USIG) to them. As long as the primary is not suspected to be faulty, MinBFT executes its normal case operation (see figure above on the right); and switches to a *view-change* operation otherwise.[36] 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, replicas only handle messages if: (1) the message is signed properly (for requests); (2) *prepare* messages come from the current primary; (3) the view number is the current one; and (4) upon receipt of a UI from a replica i, replicas check that they have already received all the UIs from i with lower counter values.

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

As opposed to the USIG-based version, to reason about the TrInc-based version, we have instantiated Identifier with the type of counter value lists, because TrInc maintains multiples counters.

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

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).

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):[38]

```
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[39] a request *r* with counter value *i1*, then no other correct replica will execute the same request with a different counter value *i2* ≠ *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 [37, pp.151–153]. The original proof of the ⟵ direction, which we claim here to be unnecessarily convoluted, goes as follows: given that a quorum of $f + 1$ replicas have committed *(r,i1)*, and a quorum of $f + 1$ replicas have committed *(r,i2)*, there must be a replica at the intersection of the two quorums that has committed both *i1* and *i2* (since there are $2f + 1$ replicas in total). Then, their proof goes by cases on whether or not that replica and the primary are correct, leading to four cases. However, such a replica at the intersection of the two quorums is not required because if a replica has executed a request, it must have received at least one prepare/commit for this request containing a UI created by the primary's USIG. Therefore, we can deduce that the primary's USIG must have created UIs for the two counter values corresponding to the two quorums mentioned above. We can then trace back these two counters to the time that primary's USIG generated UIs for them, and conclude using monotonicity. Note that we do not need to go by cases on whether replicas are correct or not because trusted components of hybrid systems (USIGs here) cannot be tampered with, and the above reasoning rely solely on properties that the system inherits from the trusted components. Thanks to Asphalion's operators, such as *ls* ⇝ *e* described in Sec. 5.2, we can always reliably access these trusted components because they cannot be compromised and because

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

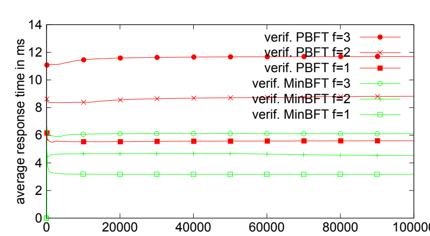
[39] In our implementations, replicas send "accept" messages whenever they execute a request. In addition to the executed request, these messages include the counter value generated for the request by the primary.

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.

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.[40] 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.[41] 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 [65] in order to run MinBFT's trusted USIG components inside Intel SGX enclaves (see `MinBFT/runtime_w_sgx/README.md` or Appx. G for further details). Graphene-SGX is a library for running unmodified applications inside SGX enclaves. Because Graphene-SGX's driver closes enclaves after each call, and because only part of the extracted code is meant to run inside SGX enclaves, our SGX-based runtime environment uses a TCP interface for replicas to interact with USIGs running in Graphene-SGX enclaves. Moreover, because to the best of our knowledge, at the time of writing, Intel SGX only supports C applications, our SGX-based runtime environment includes C wrappers around the OCaml code of the USIG components, as well as OCaml wrappers around the TCP interface implemented in C (these wrappers use [66]). 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 the figure on the right shows, the average latency of our USIG-based version of MinBFT is lower than the average latency of the verified version of PBFT presented in [35]. 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 (i.e. HMACs as opposed to RSA in [35]). We ran our experiments using a desktop with 16GB of memory, and 8 i7-6700 cores running at 3.40GHz. The experiments we report here are with one client, where $f \in \{1, 2, 3\}$, and the replicated service is a state machine whose state is a number and whose operation is addition.

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 12.5K lines of spec. and 11.5K lines of proofs, while our MinBFT proofs are about 8K lines of spec. and 4.5K lines of proofs (excluding the code we reused from Velisarios). Developing Asphalion and partially verifying MinBFT took us about one person-year.

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

Fig. 14 Comparison with related 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	✓	✓	✓	✓

9 RELATED WORK

As shown in Fig. 14 and as discussed below, several logics, models and tools have been developed over the years to reason about distributed systems. 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.

9.1 Logics and Models

Event-B [67, 68] 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 [69, 70] (not all features are covered), and has been used in a number of projects [71, 72, 73], e.g., to prove the agreement and validity of synchronous Byzantine agreement algorithms [73].

The Heard-Of (HO) model [74, 75] requires protocols to be divided into rounds, allowing processes to execute in lock-step. It was implemented in Isabelle/HOL [76] and used to verify the EIGByz [77] Byzantine agreement algorithm for synchronous systems. Model checking and the HO-model have also been used in [78, 79, 80] to verify crash fault-tolerant consensus algorithms [74].

IOA [81, 82, 83, 84] is a programming/specification language for describing asynchronous distributed systems as I/O automata [85] and for stating their properties.

TLA$^+$ [86, 87, 88] 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 [89, 90, 91, 92, 93, 94], including to prove the safety and liveness of Multi-Paxos [94], and the safety of a variant of an abstract model of PBFT [95].

9.2 Tools

ConsL [96] is a language for expressing crash-fault tolerant asynchronous and partially synchronous consensus algorithms, whose semantics is expressed in HO, and that connects to the Spin model checker [97]. 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 [98] is a framework for modular verification of implementations of crash fault tolerant systems. It provides a programming language shallowly embedded in Coq, as well as a separation-style program logic. It introduces two techniques enabling modular verification: the WithInv inference rule to strengthen assumptions, and *send-hooks* to allow logical access between components.

EventML [59, 99, 61] is a domain specific language implemented on top of the Nuprl prover [100]. It provides expressive and modular combinators for implementing and reasoning about crash-fault tolerant distributed systems (e.g., the authors proved Multi-Paxos' safety [101, 102, 60]).

IronFleet [103, 104] 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 state machine replication protocol (IronRSL), as well as a distributed key value store (IronKV).

Ivy [105] initially supported debugging infinite-state systems using bounded verification, and verifying their safety by gradually building universally quantified inductive invariants. The novel notion of *decidable decomposition* [106] allowed Ivy to automatically verify the correctness of *implementations* of crash-fault tolerant distributed systems such as Raft and Multi-Paxos (as opposed to models in [107]). Systems, models and proofs should be structured in a modular way to allow Ivy to use different decidable logics. Ivy also supports proving liveness by reducing it to safety [108].

ModP [109] is a programming framework to build, specify and compositionally test dynamic, asynchronous distributed systems. Using their framework, the authors implemented modularly and validated (through testing) two fault-tolerant distributed systems (including Multi-Paxos).

PSync [110] 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) [111]. To prove safety, users have to provide invariants, which CL checks for validity.

Verdi [112, 113] 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 [114] transforms a distributed system into a crash-tolerant one).

PVS was extensively used for verification of synchronous systems that tolerate malicious faults [115], to the extent that its design was influenced by these verification efforts [116].

ByMC [117, 118, 119, 120] is a model checker for verifying the safety and liveness of BFT algorithms. It uses an automated method for model checking parametrized threshold-guarded algorithms (e.g., processes waiting 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 (e.g. the number of tolerated faults) independent counterexample of bounded length.

Velisarios [35] is a Coq-based framework for verifying the correctness of homogeneous BFT protocols. As mentioned above it relies on a knowledge library to reason about distributed systems at a high-level of abstraction. Using Velisarios, the authors verified PBFT's agreement property [14].

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

[1] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. "An Empirical Study on the Correctness of Formally Verified Distributed Systems". In: *EUROSYS 2017*. ACM, 2017, pp. 328–343. URL: http://doi.acm.org/10.1145/3064176.3064183.

[2] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. URL: http://doi.acm.org/10.1145/357172.357176.

[3] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI 1999*. USENIX Association, 1999, pp. 173–186. URL: http://doi.acm.org/10.1145/296806.296824.

[4] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *DSN 2014*. IEEE, 2014, pp. 355–362. URL: http://dx.doi.org/10.1109/DSN.2014.43.

[5] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. "The Need for Language Support for Fault-Tolerant Distributed Systems". In: *SNAPL 2015*. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 90–102. URL: https://doi.org/10.4230/LIPIcs.SNAPL.2015.90.

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

[7] Christian Decker, Jochen Seidel, and Roger Wattenhofer. "Bitcoin meets strong consistency". In: *ICDCN 2016*. ACM, 2016, 13:1–13:10. URL: http://doi.acm.org/10.1145/2833312.2833321.

[8] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 279–296. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias.

[9] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. "Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus". In: *OPODIS 2017*. Vol. 95. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:19. URL: https://doi.org/10.4230/LIPIcs.OPODIS.2017.25.

[10] Rafael Pass and Elaine Shi. "Hybrid Consensus: Efficient Consensus in the Permissionless Model". In: *DISC 2017*. Vol. 91. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 39:1–39:16. URL: https://doi.org/10.4230/LIPIcs.DISC.2017.39.

[11] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. "A Secure Sharding Protocol For Open Blockchains". In: *CCS 2016*. ACM, 2016, pp. 17–30. URL: http://doi.acm.org/10.1145/2976749.2978389.

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

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

[14] Miguel Castro. "Practical Byzantine Fault Tolerance". Also as Technical Report MIT-LCS-TR-817. Ph.D. MIT, Jan. 2001.

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

[16] Paulo Veríssimo and Antonio Casimiro. "The Timely Computing Base Model and Architecture". In: *IEEE Trans. Computers* 51.8 (2002), pp. 916–930. URL: https://doi.org/10.1109/TC.2002.1024739.

[17] Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. "The Design of a COTSReal-Time Distributed Security Kernel". In: *EDCC-4*. Vol. 2485. LNCS. Springer, 2002, pp. 234–252. URL: https://doi.org/10.1007/3-540-36080-8_21.

[18] Paulo Veríssimo. "Uncertainty and Predictability: Can They Be Reconciled?" In: *FDDC 2003*. Vol. 2584. LNCS. Springer, 2003, pp. 108–113. URL: https://doi.org/10.1007/3-540-37795-6_22.

[19] Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. "Wormhole-aware Byzantine protocols". In: *2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability – Obstacles and Solutions* (2004). URL: http://www.di.fc.ul.pt/~nuno/PAPERS/SOS04.pdf.
[20] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems". In: *SRDS 2004*. IEEE Computer Society, 2004, pp. 174–183. URL: http://dx.doi.org/10.1109/RELDIS.2004.1353018.
[21] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. "Low complexity Byzantine-resilient consensus". In: *Distributed Computing* 17.3 (2005), pp. 237–249. URL: https://doi.org/10.1007/s00446-004-0110-7.
[22] Paulo Veríssimo. "Travelling through wormholes: a new look at distributed systems models". In: *SIGACT News* 37.1 (2006), pp. 66–81. URL: http://doi.acm.org/10.1145/1122480.1122497.
[23] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. "Attested append-only memory: making adversaries stick to their word". In: *SOSP 2007*. ACM, 2007, pp. 189–204. URL: http://doi.acm.org/10.1145/1294261.1294280.
[24] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. "TrInc: Small Trusted Hardware for Large Distributed Systems". In: *USENIX 2009*. USENIX Association, 2009, pp. 1–14. URL: http://www.usenix.org/events/nsdi09/tech/full_papers/levin/levin.pdf.
[25] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. "Efficient Byzantine Fault-Tolerance". In: *IEEE Trans. Computers* 62.1 (2013), pp. 16–30. URL: http://doi.ieeecomputersociety.org/10.1109/TC.2011.221.
[26] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. "EBAWA: Efficient Byzantine Agreement for Wide-Area Networks". In: *HASE 2010*. IEEE Computer Society, 2010, pp. 10–19. URL: https://doi.org/10.1109/HASE.2010.19.
[27] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. "CheapBFT: resource-efficient byzantine fault tolerance". In: *EuroSys '12*. ACM, 2012, pp. 295–308. URL: http://doi.acm.org/10.1145/2168836.2168866.
[28] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "BFT-TO: Intrusion Tolerance with Less Replicas". In: *Comput. J.* 56.6 (2013), pp. 693–715. URL: http://dx.doi.org/10.1093/comjnl/bxs148.
[29] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. "Resource-Efficient Byzantine Fault Tolerance". In: *IEEE Trans. Computers* 65.9 (2016), pp. 2807–2819. URL: https://doi.org/10.1109/TC.2015.2495213.
[30] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Hybrids on Steroids: SGX-Based High Performance BFT". In: *EUROSYS 2017*. ACM, 2017, pp. 222–237. URL: http://doi.acm.org/10.1145/3064176.3064213.
[31] *SGX*. 2019. URL: https://software.intel.com/en-us/sgx.
[32] *ARM TrustZone*. 2019. URL: https://www.arm.com/products/security-on-arm/trustzone.
[33] *Secure Blue*. 2019. URL: https://researcher.watson.ibm.com/researcher/view_page.php?id=6904.
[34] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)". In: *WiSec 2017*. ACM, 2017, pp. 99–110. URL: https://doi.org/10.1145/3098243.3098261.
[35] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. "Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq". In: *ESOP 2018*. Vol. 10801. LNCS. Springer, 2018, pp. 619–650. URL: https://doi.org/10.1007/978-3-319-89884-1_22.
[36] *Hyperledger*. 2019. URL: https://github.com/hyperledger-labs.
[37] Guiliana Santos Veronese. "Intrusion Tolerance in Large Scale Networks". PhD thesis. Universidade de Lisboa, 2010.
[38] *The Coq Proof Assistant*. 2019. URL: http://coq.inria.fr/.
[39] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. http://www.labri.fr/perso/casteran/CoqArt. SpringerVerlag, 2004.
[40] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. URL: http://doi.acm.org/10.1145/359545.359563.
[41] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (1985), pp. 63–75. URL: http://doi.acm.org/10.1145/214451.214456.

[42] Joseph Y. Halpern. “Using Reasoning About Knowledge to Analyze Distributed Systems”. In: *Annual Review of Computer Science* 2.1 (1987), pp. 37–68. eprint: https://doi.org/10.1146/annurev.cs.02.060187.000345. URL: https://doi.org/10.1146/annurev.cs.02.060187.000345.
[43] Joseph Y. Halpern and Yoram Moses. “Knowledge and Common Knowledge in a Distributed Environment”. In: *J. ACM* 37.3 (1990), pp. 549–587. URL: http://doi.acm.org/10.1145/79147.79161.
[44] Cynthia Dwork and Yoram Moses. “Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures”. In: *Inf. Comput.* 88.2 (1990), pp. 156–186. URL: https://doi.org/10.1016/0890-5401(90)90014-9.
[45] Prakash Panangaden and Kim Taylor. “Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems”. In: *Distributed Computing* 6.2 (1992), pp. 73–93. URL: https://doi.org/10.1007/BF02252679.
[46] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. “Knowledge-Based Programs”. In: *Distributed Computing* 10.4 (1997), pp. 199–225. URL: https://doi.org/10.1007/s004460050038.
[47] Ido Ben-Zvi. “Causality, Knowledge and Coordinaltion in Distributed Systems”. PhD thesis. Technion – Computer Science Department, Sept. 2011.
[48] Ido Ben-Zvi and Yoram Moses. “Beyond Lamport’s Happened-before: On Time Bounds and the Ordering of Events in Distributed Systems”. In: *J. ACM* 61.2 (2014), 13:1–13:26. URL: https://doi.org/10.1145/2542181.
[49] Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. “Unbeatable Consensus”. In: *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*. Vol. 8784. LNCS. Springer, 2014, pp. 91–106. URL: https://doi.org/10.1007/978-3-662-45174-8%5C_7.
[50] Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. “Unbeatable Set Consensus via Topological and Combinatorial Reasoning”. In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*. ACM, 2016, pp. 107–116. URL: https://doi.org/10.1145/2933057.2933120.
[51] Asa Dan, Rajit Manohar, and Yoram Moses. “On Using Time Without Clocks via Zigzag Causality”. In: *PODC 2017*. ACM, 2017, pp. 241–250. URL: https://doi.org/10.1145/3087801.3087839.
[52] Guy Goren and Yoram Moses. “Silence”. In: *PODC 2018*. ACM, 2018, pp. 285–294. URL: https://dl.acm.org/citation.cfm?id=3212768.
[53] Joseph Y. Halpern and Lenore D. Zuck. “A Little Knowledge Goes a Long Way: Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols”. In: *J. ACM* 39.3 (1992), pp. 449–478. URL: http://doi.acm.org/10.1145/146637.146638.
[54] Mark Bickford, Robert L. Constable, Joseph Y. Halpern, and Sabina Petride. “Knowledge-Based Synthesis of Distributed Systems Using Event Structures”. In: *LPAR 2004*. Vol. 3452. LNCS. Springer, 2004, pp. 449–465. URL: https://doi.org/10.1007/978-3-540-32275-7_30.
[55] Joseph Y. Halpern and Rafael Pass. “A Knowledge-Based Analysis of the Blockchain Protocol”. In: *TARK 2017*. Vol. 251. EPTCS. 2017, pp. 324–335. URL: https://doi.org/10.4204/EPTCS.251.22.
[56] Ronald Fagin, Joseph Halpern, Yoram Moses, and Moshe Vardi. *Reasoning About Knowledge*. Jan. 2003.
[57] K. Mani Chandy and Jayadev Misra. “How Processes Learn”. In: *Distributed Computing* 1.1 (1986), pp. 40–52. URL: https://doi.org/10.1007/BF01843569.
[58] Mark Bickford. “Component Specification Using Event Classes”. In: *CBSE 2009*. Vol. 5582. LNCS. Springer, 2009, pp. 140–155.
[59] Mark Bickford, Robert L. Constable, and Vincent Rahli. “Logic of Events, a framework to reason about distributed systems”. In: *Languages for Distributed Algorithms Workshop*. 2012. URL: http://www.nuprl.org/documents/Bickford/LOE-LADA2012.html.
[60] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. “Developing Correctly Replicated Databases Using Formal Tools”. In: *DSN 2014*. IEEE, 2014, pp. 395–406. URL: http://dx.doi.org/10.1109/DSN.2014.45.
[61] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. “EventML: Specification, Verification, and Implementation of Crash-Tolerant State Machine Replication Systems”. In: *SCP* (2017).

[62] Abhishek Anand and Ross A. Knepper. “ROSCoq: Robots Powered by Constructive Reals”. In: *ITP-6*. Vol. 9236. LNCS. Springer, 2015, pp. 34–50. URL: http://dx.doi.org/10.1007/978-3-319-22102-1_3.

[63] Eugenio Moggi. “Computational Lambda-Calculus and Monads”. In: *LICS*. IEEE Computer Society, 1989, pp. 14–23.

[64] Derek Dreyer, Amal Ahmed, and Lars Birkedal. “Logical Step-Indexed Logical Relations”. In: *Logical Methods in Computer Science* 7.2 (2011). URL: https://doi.org/10.2168/LMCS-7(2:16)2011.

[65] Chia-che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *USENIX ATC 2017*. USENIX Association, 2017, pp. 645–658. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

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

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

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

[69] Dominique Méry and Neeraj Kumar Singh. “Automatic code generation from event-B models”. In: *Symposium on Information and Communication Technology, SoICT 2011*. ACM, 2011, pp. 179–188. URL: http://doi.acm.org/10.1145/2069216.2069252.

[70] Andreas Fürst, Thai Son Hoang, David A. Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. “Code Generation for Event-B”. In: *IFM 2014*. Vol. 8739. LNCS. Springer, 2014, pp. 323–338. URL: http://dx.doi.org/10.1007/978-3-319-10181-1_20.

[71] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. “Analysis of Self-⋆ and P2P Systems Using Refinement”. In: *ABZ 2014*. Vol. 8477. LNCS. Springer, 2014, pp. 117–123. URL: http://dx.doi.org/10.1007/978-3-662-43652-3_9.

[72] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

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

[74] Bernadette Charron-Bost and André Schiper. “The Heard-Of model: computing in distributed systems with benign faults”. In: *Distributed Computing* 22.1 (2009), pp. 49–71. URL: https://doi.org/10.1007/s00446-009-0084-6.

[75] Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. “Tolerating corrupted communication”. In: *PODC 2007*. ACM, 2007, pp. 244–253. URL: http://doi.acm.org/10.1145/1281100.1281136.

[76] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. “Formal Verification of Consensus Algorithms Tolerating Malicious Faults”. In: *SSS 2011*. Vol. 6976. LNCS. Springer, 2011, pp. 120–134. URL: https://doi.org/10.1007/978-3-642-24550-3_11.

[77] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. “Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement”. In: *Inf. Comput.* 97.2 (1992), pp. 205–233. URL: https://doi.org/10.1016/0890-5401(92)90035-E.

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

[79] Tatsuhiro Tsuchiya and André Schiper. “Using Bounded Model Checking to Verify Consensus Algorithms”. In: *DISC 2008*. Vol. 5218. LNCS. Springer, 2008, pp. 466–480. URL: https://doi.org/10.1007/978-3-540-87779-0_32.

[80] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. “A Reduction Theorem for the Verification of Round-Based Distributed Algorithms”. In: *RP 2009*. Vol. 5797. LNCS. Springer, 2009, pp. 93–106. URL: https://doi.org/10.1007/978-3-642-04420-5_10.

[81] Stephen J. Garland and Nancy Lynch. “Using I/O automata for developing distributed systems”. In: *Foundations of componentbased systems*. New York, NY, USA: Cambridge University Press, 2000, pp. 285–312. URL: http://dl.acm.org/citation.cfm?id=336431.336455.

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

[83] Joshua A. Tauber. "Verifiable Compilation of I/O Automata without Global Synchronization". PhD thesis. Departement of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[84] Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. "Automated implementation of complex distributed algorithms specified in the IOA language". In: *Int. J. Softw. Tools Technol. Transf.* 11 (2 Feb. 2009), pp. 153–171. URL: http://dl.acm.org/citation.cfm?id=1529842.1529844.

[85] Nancy A. Lynch and Mark R. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In: *PODC 1987.* ACM, 1987, pp. 137–151. URL: http://doi.acm.org/10.1145/41840.41852.

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

[87] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. "Verifying Safety Properties with the TLA+ Proof System". In: *IJCAR 2010.* Vol. 6173. LNCS. Springer, 2010, pp. 142–148. URL: https://doi.org/10.1007/978-3-642-14203-1_12.

[88] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923. URL: http://doi.acm.org/10.1145/177492.177726.

[89] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. "Towards Verification of the Pastry Protocol Using TLA $^+$". In: *FORTE 2011.* Vol. 6722. LNCS. Springer, 2011, pp. 244–258. URL: http://dx.doi.org/10.1007/978-3-642-21461-5_16.

[90] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. "Checking Cache-Coherence Protocols with TLA$^+$". In: *Formal Methods in System Design* 22.2 (2003), pp. 125–131. URL: http://dx.doi.org/10.1023/A:1022969405325.

[91] William J. Bolosky, John R. Douceur, and Jon Howell. "The Farsite project: a retrospective". In: *Operating Systems Review* 41.2 (2007), pp. 17–26. URL: http://doi.acm.org/10.1145/1243418.1243422.

[92] Chris Newcombe. "Why Amazon Chose TLA +". In: *ABZ 2014.* Vol. 8477. LNCS. Springer, 2014, pp. 25–39. URL: http://dx.doi.org/10.1007/978-3-662-43652-3_3.

[93] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods". In: *Commun. ACM* 58.4 (2015), pp. 66–73. URL: http://doi.acm.org/10.1145/2699417.

[94] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. "Formal Verification of Multi-Paxos for Distributed Consensus". In: *FM 2016.* Vol. 9995. LNCS. 2016, pp. 119–136. URL: https://doi.org/10.1007/978-3-319-48989-6_8.

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

[96] Ognjen Maric, Christoph Sprenger, and David A. Basin. "Cutoff Bounds for Consensus Algorithms". In: *CAV 2017.* Vol. 10427. LNCS. Springer, 2017, pp. 217–237. URL: https://doi.org/10.1007/978-3-319-63390-9%5C_12.

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

[98] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols". In: *POPL 2018.* 2018.

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

[100] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P.Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing mathematics with the Nuprl proof development system.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

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

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

[103] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct". In: *SOSP 2015*. ACM, 2015, pp. 1–17. URL: http://doi.acm.org/10.1145/2815400.2815428.

[104] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving safety and liveness of practical distributed systems". In: *Commun. ACM* 60.7 (2017), pp. 83–92. URL: http://doi.acm.org/10.1145/3068608.

[105] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. "Ivy: safety verification by interactive generalization". In: *PLDI 2016*. ACM, 2016, pp. 614–630. URL: http://doi.acm.org/10.1145/2908080.2908118.

[106] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. "Modularity for decidability of deductive verification with applications to distributed systems". In: *PLDI 2018*. ACM, 2018, pp. 662–677. URL: http://doi.acm.org/10.1145/3192366.3192414.

[107] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. "Paxos made EPR: decidable reasoning about distributed protocols". In: *PACMPL* 1.OOPSLA (2017), 108:1–108:31. URL: http://doi.acm.org/10.1145/3140568.

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

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

[110] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. "PSync: a partially synchronous language for fault-tolerant distributed algorithms". In: *POPL 2016*. ACM, 2016, pp. 400–415. URL: http://doi.acm.org/10.1145/2837614.2837650.

[111] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. "A Logic-Based Framework for Verifying Consensus Algorithms". In: *VMCAI 2014*. Vol. 8318. LNCS. Springer, 2014, pp. 161–181. URL: https://doi.org/10.1007/978-3-642-54013-4_10.

[112] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *PLDI 2015*. ACM, 2015, pp. 357–368. URL: http://doi.acm.org/10.1145/2737924.2737958.

[113] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. "Planning for change in a formal verification of the raft consensus protocol". In: *CPP 2016*. ACM, 2016, pp. 154–165. URL: http://doi.acm.org/10.1145/2854065.2854081.

[114] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

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

[116] Sam Owre, John M. Rushby, Natarajan Shankar, and Friedrich W. von Henke. "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS". In: *IEEE Trans. Software Eng.* 21.2 (1995), pp. 107–125. URL: https://doi.org/10.1109/32.345827.

[117] Igor Konnov, Helmut Veith, and Josef Widder. "SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms". In: *CAV 2015*. Vol. 9206. LNCS. Springer, 2015, pp. 85–102. URL: https://doi.org/10.1007/978-3-319-21690-4_6.

[118] Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. “A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms”. In: *POPL 2017*. ACM, 2017, pp. 719–734. URL: http://dl.acm.org/citation.cfm?id=3009860.

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

[120] Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. “Synthesis of Distributed Algorithms with Parameterized Threshold Guards”. In: *OPODIS 2017*. Vol. 95. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 32:1–32:20. URL: https://doi.org/10.4230/LIPIcs.OPODIS.2017.32.

A SIMPLE EXAMPLE OF A MOC LOCAL SYSTEM

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

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

Let OP1's update function be defined as follows:

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

Let OP2's update function be defined as follows:

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

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

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

where ⟨TOTAL(o), []⟩ is a directed message, in this case, the instruction to send the message TOTAL(o) to the empty list of recipients [].

Whenever this local system receives a message m, it applies M's update function to m and to the list of its three sub-components OP1, OP2, and ST. If m is, for example, of the form ADD1(n), then M calls OP1 on the input n. This results in looking for a component with that name in the list of M's sub-components. Because such a component exists, namely OP1, we create a new local system with main component OP1 and sub-component ST (the only sub-component with level lower than OP1's). We then apply OP1's update function to n and to the list containing its single sub-component, namely ST. This results in calling ST on the input n. Because ST is present in the list of OP1's sub-components, we then create a new local system with main component ST and no sub-components (because there are no sub-components with level lower than ST's), and we apply this system to n. This results in applying ST's update function to n and to the empty list of sub-components. If this call is the first call, and if ST's initial state is 0, then its update function returns the new state n and outputs n. It also returns the empty list of sub-components that it took as input. Going back to OP1, we then update the state of its sub-component ST from 0 to n. Finally, OP1 return this updated list of sub-components, it outputs the value n, and its state remains `tt`. Going back to M, we then update the state of OP1 from `tt` to `tt`, and we replace the sub-component ST with state 0 that M took as input, with the one with state n that OP1 returned. Finally M returns the list of updated sub-components OP1 with state `tt`; OP2 with state `tt`, which it did not call here; and ST with state n. M also returns the state `tt` and outputs a single directed message: ⟨TOTAL(n), []⟩.

Fig. 15 LoCK's structural rules

$$\frac{\langle G\rangle\, H_1, H_2 \vdash \sigma_2}{\langle G\rangle\, H_1, h, H_2 \vdash \sigma_2}\ \mathtt{thin_h} \qquad \frac{\langle G_1, G_2\rangle\, H \vdash \sigma}{\langle G_1, g, G_2\rangle\, H \vdash \sigma}\ \mathtt{thin_g} \qquad \frac{\langle G\rangle\, H[x_2 : \sigma'] \vdash \sigma}{\langle G\rangle\, H[x_1 : \sigma'] \vdash \sigma}\ \mathtt{ren_h} \qquad \frac{\langle G[y_2 : \alpha]\rangle\, H \vdash \sigma}{\langle G[y_1 : \alpha]\rangle\, H \vdash \sigma}\ \mathtt{ren_g}$$

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

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

B A DEEP-EMBEDDING TO SPAWN SUB-COMPONENTS

As mentioned in Sec. 5.4, the simple language presented here (the type $\mathsf{Proc}(A)$) is not the only choice. Note however that it is enough for a large number of protocols. To allow other features, one can simply introduce extensions of this language. For example, to allow spawning sub-processes, one could define the following language (see the file called `model/ComponentSM5.v` in our implementation): let $\mathsf{SpawnProc}(A)$ be the set of terms sp of the form:

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

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

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

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

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

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

C ADDITIONAL LOCK RULES

We present here some important rules of LoCK that we omitted in Sec. 6.4 for space reasons (see the file called `model/CalculusSM.v` for a list of our rules).

Fig. 15 presents LoCK's structural rules while Fig. 16 presents LoCK's predicate logic rules, which are all standard.

Fig. 16 LoCK's predicate logic rules

$$\frac{}{\langle G\rangle\, H \vdash \top \mathbin{@} e}\ \top_{\mathtt{I}} \qquad \frac{}{\langle G\rangle\, H[\bot \mathbin{@} e] \vdash \sigma}\ \bot_{\mathtt{E}}$$

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

$$\frac{\begin{array}{c}\langle G\rangle\, H_1, x : \tau_1 \mathbin{@} e, H_2 \vdash \sigma \\ \langle G\rangle\, H_1, x : \tau_2 \mathbin{@} e, H_2 \vdash \sigma\end{array}}{\langle G\rangle\, H_1, x : \tau_1 \vee \tau_2 \mathbin{@} e, H_2 \vdash \sigma}\ \vee_{\mathtt{E}} \qquad \frac{\langle G\rangle\, H \vdash \tau_1 \mathbin{@} e}{\langle G\rangle\, H \vdash \tau_1 \vee \tau_2 \mathbin{@} e}\ \vee_{\mathtt{Il}} \qquad \frac{\langle G\rangle\, H \vdash \tau_2 \mathbin{@} e}{\langle G\rangle\, H \vdash \tau_1 \vee \tau_2 \mathbin{@} e}\ \vee_{\mathtt{Ir}}$$

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

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

$$\frac{\langle G\rangle\, H_1, x : f(v) \mathbin{@} e, H_2 \vdash \sigma \quad \mathsf{oftype}(v, \theta)}{\langle G\rangle\, H_1, x : \forall\langle \theta, f\rangle \mathbin{@} e, H_2 \vdash \sigma}\ \forall_{\mathtt{E}} \qquad \frac{\Lambda[v] \quad \langle G\rangle\, H \vdash f(v) \mathbin{@} e \quad \mathsf{oftype}(v, \theta)}{\langle G\rangle\, H \vdash \forall\langle \theta, f\rangle \mathbin{@} e}\ \forall_{\mathtt{I}}$$

Fig. 17 Additional event relation rules of LoCK

$$\frac{\langle G\rangle\, H[x : \tau \mathbin{@} \mathsf{pred}^{=}(e)] \vdash \sigma}{\langle G\rangle\, H[x : \subset\tau \mathbin{@} e] \vdash \sigma}\ \subset_{\mathtt{E}}$$

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

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

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

$$\frac{\begin{array}{c}\langle G[y : e_1 \subset e_2]\rangle\, H \vdash \sigma \\ \langle G[y : e_1 \sqsubset e, e \subset e_2]\rangle\, H \vdash \sigma\end{array}}{\langle G[y : e_1 \sqsubset e_2]\rangle\, H \vdash \sigma}\ \mathtt{split}_{\sqsubset} \qquad \frac{\langle G[y : e_1 \sqsubseteq \mathsf{pred}^{=}(e_2), \mathsf{pred}^{=}(e_2) \subset e_2]\rangle\, H \vdash \sigma}{\langle G[y : e_1 \sqsubset e_2]\rangle\, H \vdash \sigma}\ \mathtt{splitPred}_{\sqsubset}$$

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

Fig. 18 Additional logic of events rules of LoCK

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

Fig. 17 presents additional event relation rules, that we omitted in Fig. 11 for space reasons. The $\subset_{\mathtt{E}}$ rule is the standard elimination rule for $\subset$, allowing to navigate to previous events. The $\mathtt{STR}_{\sqsubseteq}$ and $\mathtt{STR}_{\preceq}$ allow strengthening $\sqsubseteq$ and $\preceq$. The $\mathtt{STRl}_{\preceq}$ and $\mathtt{STRl}_{\prec}$ allow strengthening $\preceq$ and $\prec$. The $\mathtt{split}_{\sqsubset}$ and $\mathtt{splitPred}_{\sqsubset}$ allow splitting guards to get intermediate events.

Fig. 19 Additional knowledge rules of LoCK

Let τ be of one of the forms $v_1 = v_2$, $i_1 < i_2$, $\mathcal{HI}(t, i)$, $\mathcal{O}(d, a)$, $\mathcal{G}(d, t)$

$$\dfrac{\langle G\rangle\, H \vdash \tau \mathbin{@} e_2}{\langle G\rangle\, H \vdash \tau \mathbin{@} e_1}\;\textsf{change} \qquad\qquad \dfrac{\begin{array}{c}\langle G\rangle\, H \vdash v_1 = v_2 \mathbin{@} e\\ \langle G\rangle\, H \vdash \tau[v_2] \mathbin{@} e\end{array}}{\langle G\rangle\, H \vdash \tau[v_1] \mathbin{@} e}\;\textsf{valSub}$$

Fig. 18 presents additional logic of events rules, that we omitted in Fig. 12 for space reasons. The $@_{\texttt{loc}}$ rule (note that this rule is invertible) states that if $e_1 \sqsubseteq e_2$ then e_1 and e_2 happen at the same location. The `loc` rule states that each event happens at a single location.

Fig. 19 presented additional knowledge rules, that we omitted in Fig. 13 for space reasons. The change rule allows changing the current event for event agnostic expressions. The `valSub` rule allows substituting equal values in any expression.

We conclude this section with several examples of derivations within LoCK. Namely, we introduce *Owns Propagated* to illustrate the use of the following rules: cut, $\exists_{\mathsf{E}}$, $\exists_{\mathsf{I}}$ and change. Next, we show *Id After is Id Before*, that uses $\vee_{\mathsf{I1}}$ and $\square_{\mathsf{I}}$. Moreover, using *Id Before is Id After*, we demonstrates use of following rules: hyp, $\vee_{\mathsf{E}}$, $\subset_{\mathsf{E}}$, weak, $\equiv_{\mathsf{sym}}$, $\mathsf{sub}_{\mathsf{C}}$, $\wedge_{\mathsf{E}}$, $\neg\odot$, $\rightarrow_{\mathsf{E}}$, $\perp_{\mathsf{E}}$. Finally, *Causal, Equal and First* illustrates how $\mathsf{STR}_{\sqsubseteq}$ and $\mathsf{thin}_{\mathsf{h}}$ can be used.

Owns Propagated. We show here that we can derive $\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \mathcal{O}(d) \mathbin{@} e_2$ from $\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \mathcal{O}(d) \mathbin{@} e_1$, [42]

$$\dfrac{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \mathcal{O}(d) \mathbin{@} e_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \mathcal{O}(d, a) \mathbin{@} e_1, y : @(a) \mathbin{@} e_1 \vdash @(a) \mathbin{@} e_1}\;\textsf{hyp}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \mathcal{O}(d, a) \mathbin{@} e_1, y : @(a) \mathbin{@} e_1 \vdash @(a) \mathbin{@} e_2}\;@_{\texttt{loc}} \quad \Pi}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \mathcal{O}(d, a) \mathbin{@} e_1, y : @(a) \mathbin{@} e_1 \vdash @(a) \wedge \mathcal{O}(d, a) \mathbin{@} e_2}\;\wedge_{\mathsf{I}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : @(a) \wedge \mathcal{O}(d, a) \mathbin{@} e_1 \vdash @(a) \wedge \mathcal{O}(d, a) \mathbin{@} e_2}\;\wedge_{\mathsf{E}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : @(a) \wedge \mathcal{O}(d, a) \mathbin{@} e_1 \vdash \exists_{\mathsf{n}}(@(a) \wedge \mathcal{O}(d, a)) \mathbin{@} e_2}\;\exists_{\mathsf{I}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H, x : \mathcal{O}(d) \mathbin{@} e_1 \vdash \mathcal{O}(d) \mathbin{@} e_2}\;\exists_{\mathsf{E}}}{\langle G, y : e_1 \sqsubseteq e_2\rangle\, H \vdash \mathcal{O}(d) \mathbin{@} e_2}\;\textsf{cut}$$

where Π is:

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

Id After is Id Before. We show here that we can derive $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \mathcal{I}^-(i) \mathbin{@} e_2$ from $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \mathcal{I}^+(i) \mathbin{@} e_1$, [43]

$$\dfrac{\dfrac{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \mathcal{I}^+(i) \mathbin{@} e_1}{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \subset\mathcal{I}^+(i) \mathbin{@} e_2}\;\square_{\mathsf{I}}}{\langle G, y : e_1 \subset e_2\rangle\, H \vdash \mathcal{I}^-(i) \mathbin{@} e_2}\;\vee_{\mathsf{I1}}$$

Id Before is Id After. We show here that we can derive $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \mathcal{I}^+(i) \mathbin{@} e_1$ from $\langle G, y : e_1 \subset e_2\rangle\, H \vdash \mathcal{I}^-(i) \mathbin{@} e_2$ [44]

[42] See `DERIVED_RULE_owns_change_localle_true` in `model/CalculusSM.v`.
[43] See `DERIVED_RULE_id_after_is_id_before_true` in `model/CalculusSM.v`.
[44] See `DERIVED_RULE_id_before_is_id_after_true` in `model/CalculusSM.v`.

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

where Π_1 is:

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

and Π_2 is:

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

and Π_3 is:

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

Causal, Equal and First. We show here that we can derive $\langle G, y : e_1 \sqsubseteq e_2\rangle H \vdash \sigma$ from $\langle G, y : e_1 \sqsubseteq e_2\rangle H \vdash \odot @ e_2$ and $\langle G, y : e_1 \equiv e_2\rangle H \vdash \sigma$ [45]

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

where Π is:

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

D TRUSTED KNOWLEDGE DISSEMINATION

We present here another high-level result, which we derived within LoCK, and which is the crux of Micro's validity. Roughly speaking, this general high-level LoCK lemma (presented below) says that if a correct node disseminates a piece of data d (i.e. $\mathcal{D}(d)$), then there must have been a disseminated trusted piece of data t (i.e. $\prec\mathcal{OD}(t)$), that was generated for d (i.e. $\mathcal{G}(d, t)$) by the owner of the data. From this, we can straightforwardly derive that if a Micro backup accepts a request r with counter value i, then it must have been that the primary generated a unique identifier with counter value i for this request.

[45]See `DERIVED_RULE_causalle_is_equal_if_first_true` in `model/CalculusSM.v`.

Theorem D.1 (Trusted Knowledge Dissemination). *The following rule is derivable within LoCK:*[46]

$$\frac{\begin{array}{l}\Lambda[e'] \ \langle G\rangle\, H \vdash \mathbf{DKT} \land \mathbf{KLD} \land \mathbf{LID} \ @\ e' \\ \langle G\rangle\, H \vdash \mathcal{D}(d)\ @\ e \\ \langle G\rangle\, H \vdash \boldsymbol{C}\ @\ e\end{array}}{\langle G\rangle\, H \vdash \exists_{\mathsf{t}}\lambda t.(\mathcal{G}(d,t) \land \mathcal{K}^{+}(t) \land (\prec\mathcal{OD}(t) \lor \mathcal{O}(t)))\ @\ e}$$

This rule says that a disseminated piece of data must correspond to some trusted piece of data that was generated in the past by the owner of the data. More precisely, it states that if **DKT** (see below), **KLD** and **LID** (see Sec. 6.6) are satisfied, and if some data d was disseminated by a node, say $\boldsymbol{a}$, at event e,[47] then there must exist some trusted piece of data t that was generated for d, such that $\boldsymbol{a}$ knows t at event e, and either: (1) $\boldsymbol{a}$ owns t at e; or (2) some other node owns and disseminated the data in the past. Let us now get back to **DKT**, which we have not discussed so far:

$$\mathbf{DKT} = \forall_{\mathsf{d}}\lambda d.\mathcal{D}(d) \rightarrow \boldsymbol{C} \rightarrow (\exists_{\mathsf{t}}\lambda t.\mathcal{K}^{+}(t) \land \mathcal{G}(d,t)) \tag{9}$$

This states that if a correct node $\boldsymbol{a}$ disseminates some piece of data d, then there must exist some trusted piece of data t, that $\boldsymbol{a}$ knows and was generated for data d.

E OPENING THE LID

E.1 Primitive Principles Behind LID

Sec. 6.6 presents typical assumptions about knowledge, expressed within LoCK. In particular, it presents the following **LID** assumption in Eq. 2, which states that if one learns about a trusted piece of data, then this trusted piece of data must have been disseminated by its owner in the past:

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

As mentioned in Sec. 7.2, **LID** essentially follows from our generic HyLoE communication assumption called AXIOM_auth_messages_were_sent_or_byz (see `model/ComponentAxiom.v` for more details). Given a distributed system such as MinBFT, it is not complicated to prove that **LID** (its HyLoE interpretation) holds assuming AXIOM_auth_messages_were_sent_or_byz. However, it requires using induction in HyLoE, which we are trying to avoid: we are aiming at having all the inductive reasoning done in LoCK in order to keep the reasoning done in HyLoE as simple as possible. The reason for the inductive nature of this proof is that **LID** allows going back directly to the owner of the learned trusted piece of data, while AXIOM_auth_messages_were_sent_or_byz only allows getting back to *some* point in space/time, where the trusted piece of data was disseminated: it does not have to be disseminated by the owner at that point because the data might have been relayed by an intermediary node. As it turns out, **LID** can be derived within LoCK from more primitive principles, which we present next.[48]

Let **Com** be the following LoCK expression:

$$\forall_{\mathsf{t}}\lambda t.\mathcal{L}(t) \rightarrow (\exists_{\mathsf{d}}\lambda d.\prec(\mathcal{ND}(d) \land t \in d \land \boldsymbol{C})) \lor \prec\mathcal{OD}(t)$$

As for $\boldsymbol{C}$, which we discuss above in Appx. D, $t \in d$ is not discussed in the main body of this paper because it is scarcely used. It expresses that the trusted piece of data t occurs in the piece of data d.

[46] See the lemma called `DERIVED_RULE_disseminate_if_learned_and_disseminated2_true` in the file called `model/CalculusSM_derived.v`.

[47] We also assume that the node that disseminated the data is correct, i.e., $\boldsymbol{C}$ holds at e. This operator is not discussed in this paper because, even though simple, it is only used scarcely. See the file called `model/CalculusSM.v` for more information. Note there that the $\boldsymbol{C}$ operator is defined in terms of more primitive operators. The only primitive operator used in $\boldsymbol{C}$'s definition, which is not presented in this paper is an operator stating that the current event is correct as discussed in Sec. 4.2.

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

$\mathcal{ND}(d)$ is defined as $\mathcal{N} \wedge \mathcal{D}(d)$, where $\mathcal{N} = \exists_{\mathsf{n}}\lambda a.@(a)$. We have proved that **Com** is a straightforward consequence of the communication axiom AXIOM_auth_messages_were_sent_or_byz, i.e., we have proved (assuming a few simple properties that relate HyLoE parameters and LoCK parameters):[49]

$$\begin{aligned}&\forall eo \in \mathsf{EO}.\ \text{AXIOM\_auth\_messages\_were\_sent\_or\_byz}\ eo\ \mathsf{sys}\\ &\quad\rightarrow \forall e \in \mathsf{Event}(eo). [\![\mathbf{Com}]\!]_e\end{aligned} \tag{10}$$

We can then derive the following derived rule:

$$\frac{\Lambda[e']\ \langle G\rangle\ H \vdash \mathbf{Com} \wedge \mathbf{KLD} \wedge \mathbf{DIK} \wedge \mathbf{KIK}\ @\ e'}{\langle G\rangle\ H \vdash \mathbf{LID}\ @\ e}\ \texttt{LID} \tag{11}$$

where **KLD** is defined in Eq. 3 in Sec. 6.6, and

$$\begin{aligned}\mathbf{DIK} &= \forall_{\mathsf{d}}\lambda d.\mathcal{ND}(d) \rightarrow C \rightarrow \mathcal{K}^{+}(d)\\ \mathbf{KIK} &= \forall_{\mathsf{d}}\lambda d.\forall_{\mathsf{t}}\lambda t.\mathcal{K}^{+}(d) \rightarrow t \in d \rightarrow \mathcal{K}^{+}(t)\end{aligned}$$

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

E.2 A Proof of the LID Derived Rule

Let us now discuss the proof of the `LID` derived rule (the interested reader is invited to go through `DERIVED_RULE_implies_all_trusted_learns_if_gen2_true` in `model/CalculusSM_derived4.v` for more details). First of all, we show that we can derive $\mathcal{K}^{+}(t)$ from **DIK**, **KIK**, $\mathcal{ND}(d)$, C, and $t \in d$ (we combine some steps for readability):

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

where Π is

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

The rule we just derived is then:

$$\frac{\langle G\rangle\ H \vdash \mathbf{DIK}\ @\ e \quad \langle G\rangle\ H \vdash \mathbf{KIK}\ @\ e \quad \langle G\rangle\ H \vdash \mathcal{ND}(d)\ @\ e \quad \langle G\rangle\ H \vdash C\ @\ e \quad \langle G\rangle\ H \vdash t \in d\ @\ e}{\langle G\rangle\ H \vdash \mathcal{K}^{+}(t)\ @\ e}\ \texttt{DITK}$$

Let us now go back to Eq. 11. We proved the validity of this derived rule in LoCK by induction. As it turns out, we used a different rule than `ind`, which allows us to go by induction on the *happened before* relation, as opposed to `ind`, which goes by induction on the *direct predecessor* relation (from now on we will call both rules `ind` for simplicity):[50]

$$\frac{\Lambda[e]\ \langle G\rangle\ H \vdash (\forall_{\prec}\tau) \rightarrow \tau\ @\ e}{\langle G\rangle\ H \vdash \tau\ @\ e}\ \texttt{ind}$$

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

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

[49]See `ASSUMPTION_authenticated_messages_were_sent_or_byz_true` in `model/CalculusSM_derived4.v`.

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

In our proof of Eq. 11, we will also use the following derived rule, which is similar to Eq. 7, where $\forall_{\preceq}\tau = \forall_{\prec}\tau \lor \tau$ (see `DERIVED_RULE_KLD_implies_gen2_true` in `model/CalculusSM_derived4.v`):

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

In addition, we will also use the following derived rule, which strengthens a $\forall_{\preceq}$ to a $\forall_{\prec}$ by navigating to a later point in space/time (from e' to e below) (see `DERIVED_RULE_forall_node_before_eq_trans_true` in `model/CalculusSM_derived4.v`):

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

Finally, we will also use the following derived rule, which allows weakening $\prec$ to $\preceq$ by navigating to an earlier point in space/time, i.e., from e to e' below (see `DERIVED_RULE_unhappened_before_if_causal_trans` in `model/CalculusSM.v`):

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

Let us now derive Eq. 11:

$$\dfrac{\dfrac{\dfrac{\langle G\rangle\ H \vdash \mathbf{Com} \ @\ e \qquad \dfrac{\Pi_1 \qquad \dfrac{}{\langle G\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, y : \mathcal{L}(t) \ @\ e, z : \prec\mathcal{OD}(t) \ @\ e \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \mathsf{hyp}}{\langle G\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, y : \mathcal{L}(t) \ @\ e, z : \mathbf{Com} \ @\ e \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \forall_{\mathsf{E}} + \rightarrow_{\mathsf{E}} + \lor_{\mathsf{E}}}{\langle G\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, y : \mathcal{L}(t) \ @\ e \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \mathsf{cut} + \mathsf{thin_h}}{\langle G\rangle\ H \vdash \forall_{\prec}\mathbf{LID} \rightarrow \mathbf{LID} \ @\ e}\ \rightarrow_{\mathsf{I}} + \forall_{\mathsf{I}}}{\langle G\rangle\ H \vdash \mathbf{LID} \ @\ e}\ \mathsf{ind}$$

where Π_1 is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle G, u : e' \prec e\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e \vdash \forall_{\preceq}\mathbf{LID} \ @\ e'}\ \mathsf{STR}\forall_{\preceq} + \mathsf{hyp} \qquad \Pi_2}{\langle G, u : e' \prec e\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, k : \mathcal{K}^{+}(t) \ @\ e' \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \mathsf{cut} + \mathsf{KID} + \mathsf{thin_h} + \rightarrow_{\mathsf{E}}}{\langle G, u : e' \prec e\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, z : \mathcal{ND}(d) \ @\ e', i : t \in d \ @\ e', c : C \ @\ e', k : \mathcal{K}^{+}(t) \ @\ e' \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \mathsf{thin_h}}{\langle G, u : e' \prec e\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, z : \mathcal{ND}(d) \ @\ e', i : t \in d \ @\ e', c : C \ @\ e' \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \mathsf{cut} + \mathsf{DITK} + \mathsf{thin_h} + \mathsf{hyp}}{\dfrac{\langle G\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, z : (\exists_{\mathsf{d}}\lambda d.\prec(\mathcal{ND}(d) \land t \in d \land C)) \ @\ e \vdash \prec(\mathcal{OD}(t)) \ @\ e}{\langle G\rangle\ H, x : \forall_{\prec}\mathbf{LID} \ @\ e, y : \mathcal{L}(t) \ @\ e, z : (\exists_{\mathsf{d}}\lambda d.\prec(\mathcal{ND}(d) \land t \in d \land C)) \ @\ e \vdash \prec(\mathcal{OD}(t)) \ @\ e}\ \mathsf{thin_h}}\ \exists_{\mathsf{E}} + \prec_{\mathsf{E}} + \land_{\mathsf{E}}$$

and where Π_2 is:

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

F LOCK TUTORIAL

We briefly explain here how to use LoCK to prove lemmas. We provide several examples in the following files: `model/CalculusSM.v`, `model/CalculusSM2.v`, `model/CalculusSM_derived.v`, `model/CalculusSM_derived2.v`, `model/CalculusSM_derived3.v`, and `model/CalculusSM_derived3.v`. The names of the lemmas in those files either start with `PRIMITIVE` for primitive rules, or by `DERIVED` for derived rules. The example we use here is `DERIVED_RULE_unlocal_before_eq_hyp_true`, which we proved in `model/CalculusSM.v`, and which we discuss in Sec. 6.5:

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

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

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

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

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

We can then start applying rules. Every time we apply a rule, we use the proof that the rule is true. For example, `LOCKapply` (PRIMITIVE_RULE_unlocal_before_hyp_true *u*), applies the PRIMITIVE_RULE_unlocal_before_hyp rule, which we have proved to be valid in the lemma PRIMITIVE_RULE_unlocal_before_hyp_true. Incidentally, this rule is the $\square_E$ elimination rule presented in Fig. 11. The `LOCKelim` tactic automatically tries to apply the appropriate (elimination) rule. Here because the hypothesis x is of the form $\sqsubseteq\tau$, which is is defined as $\sqsubset\tau \vee \tau$ (see Sec. 6.2), `LOCKelim` automatically applies the *or* elimination rule. From this, we get two branches, one for each branch of the *or*, which is why we have two blocks below that tactic: the first one is the proof of the left branch, and the second one is the proof of the right branch.

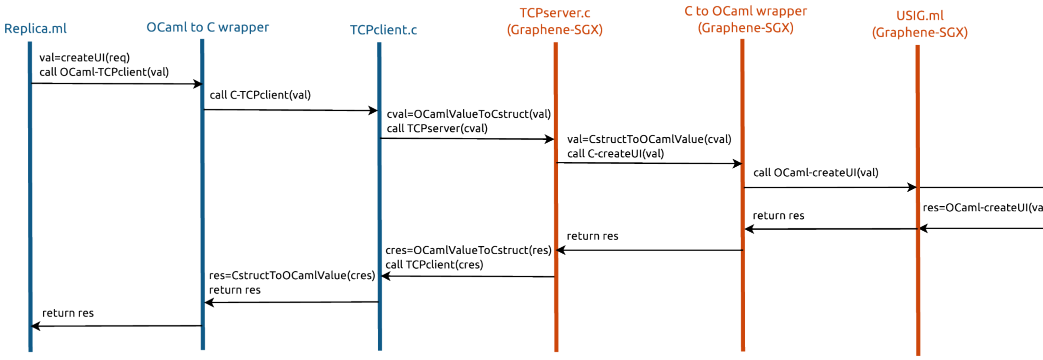
We use a couple more useful tactics in this proof, which we describe next. To prove the left branch, we use the `LOCKapply@` *u rule* tactic, which is similar to `LOCKapply`, but in addition gets either the guards or the hypotheses (depending on whether the name u is a guard name or an hypothesis name) in the right shape whenever a rule mentions a guard or an hypothesis. For example DERIVED_RULE_add_localle_refl_true, is the validity proof of one of our weakening rule (see the rule called weak in Fig. 11 in Sec. 6.4). The guards in the conclusion of that rule are of the form $G_1, y : e' \sqsubset e, G_2$. The tactic `LOCKapply@` helps turn the guards in the current sequent into that precise shape by pointing to the guard name y (u in our proof above).

Finally `inst_hyp` *e st*, instantiates the hypotheses of our rule, namely the function (`fun` *e'* ⇒ [⟨(*u* : *e'* ⊑ *e*), *R*⟩ *H*, (*x* : *a* @ *e'*), *K* ⊢ *b*]) with the event variable *e*, and call the instances *st*.

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

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

Fig. 20 OCaml/SGX interaction



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

G OCAML RUNTIME ENVIRONMENTS

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

SGX-free runtime. Beside the runtime environment discussed in Sec. 8 and below, that uses Intel SGX, we developed an additional runtime environment (located in `MinBFT/runtime_wo_sgx`) that does not depend on any trusted environment for two reasons: it enables testing our framework on platforms that do not contain any trusted execution environment; and it can be very useful for debugging.

SGX-based runtime. As mentioned in Sec. 8, using Asphalion, one can extract OCaml code from distributed systems implemented using MoC, such that trusted components execute inside Intel SGX enclaves. We chose to rely on Graphene-SGX [1] to do this because, to the best of our knowledge, one cannot directly run OCaml code inside SGX enclaves. Instead of using Graphene-SGX, one could run OCaml's runtime environment inside SGX enclaves, which would require creating OCALLs for all system calls made by OCaml's runtime environment that are not included in the libraries provided by SGX. Besides the fact that this solution could lead to security issues, it might be very slow.

Here, using a concrete example, i.e., a *createUI* call, we explain the interaction between a replica's main component and the Graphene-SGX enclave that runs this replica's USIG component. As mentioned above, because Graphene-SGX closes enclaves after each call, we implemented a loop around the USIG service to keep it running forever, as well as a TCP interface to access this loop. Also, because Graphene-SGX, to the best of our knowledge, provides only a C interface, we implemented this loop and this TCP interface in C. As shown in Fig. 20, when the main component of a replica calls the *createUI* function of its USIG, this call is forwarded to the client of this TCP interface. Moreover, because we extract MoC code to OCaml, we had to implement an OCaml/C wrapper around our TCP interface implemented C. Next, the TCP client forwards the value it received through this call to *createUI* to the TCP server, which runs inside a Graphene-SGX enclave. To transfer this OCaml value across the TCP connection, we had to implement a custom serializer to convert that value to a C structure. Finally, when the TCP server running a Graphene-SGX enclave receives this C structure, it uses a custom deserializer to convert it back to an OCaml value, which the server uses to call the OCaml *createUI* function (again using a C/OCaml wrapper around the USIG code). Note that similar steps have to executed to deliver the value computed by the USIG, back to the main component.

H VELISARIOS VS. ASPHALION

As explained above, Velisarios [2] is a framework to reason about homogeneous Byzantine fault-tolerant systems. It provides: (1) a general model of processes, where each local system is a state machine; (2) a Byzantine logic of events that supports arbitrary (Byzantine) events, i.e., events for which no information is available, which could for example have been triggered by malicious or corrupted nodes; and (3) a knowledge library to reason about Byzantine fault-tolerant systems at a high-level of abstraction. This knowledge library is shallowly embedded in Coq, and provides two modal operators: learn and know. As discussed in Sec. 2.3, Asphalion departs from Velisarios in several ways, but reuses a small part of it, namely Asphalion's logic of events (HyLoE) extends the one of Velisarios. The other components of Asphalion (MoC and LoCK) are new.[51] Let us now discuss some differences between Velisarios and Asphalion.

HyLoE. As in Velisarios, Asphalion provides a logic of events to model runs of distributed systems as partial orders on events. As in Velisarios, the logic of events implemented in Asphalion is shallowly embedded in Coq, thereby allowing one to use Coq's expressiveness to reason about distributed systems. This is to contrast with other approaches such as TLA or Event-B that rely on less expressive logics. However, Asphalion's logic of events extends Velisarios's so that in addition to supporting arbitrary events, HyLoE also supports events where trusted components of compromised processes are called. As explained above Asphalion supports three kinds of events, the first two being also supported by Velisarios: an event is either (1) *a correct event* triggered at a correct location by the receipt of a message; or (2) *a Byzantine event* that corresponds to an arbitrary action, and therefore no reliable information can be extracted from that event; or (3) *a hybrid event* that corresponds to the call of a trusted component at a possibly compromised node. HyLoE gives access to the inputs on which those trusted components are called at those "hybrid" events. See Sec. 4 for more details (especially the `TItrust`(*it*) constructor, which is one of the constructs assigned by `trigger` to events). This enables reasoning about hybrid systems, because thanks to those trusted inputs associated with hybrid events, we can now compute the inputs, states and outputs of trusted components. Note also that HyLoE does not prevent from reasoning about homogeneous systems, because one can implement systems that do not contain trusted components.

MoC. Unfortunately, Velisarios only provides a rudimentary language to implement systems. Distributed systems there are collections of state machines, one per local system, where a state machine is essentially a Coq function implementing the update function of the machine. MoC goes well beyond this, by allowing local state machines to be defined as collections of interacting

[51] The `README.md` file in the root directory of our implementation provides a short summary of the files that are part of Asphalion, but are not part of Velisarios.

components. The components of a local system interact through a monad. MoC's monad provides three main operators to build a component: a *return* operator to turn a Coq expression into a component; a *bind* operator to compose two components; and a *call* operator to allow components to call their sub-components. Those operators can be combined in any way one wants using any Coq function one desires, as long as the resulting code has the right *component* type. A local system is then a collection of components, with a distinguished component as the *main* component, and some of them being flagged as trusted; and a distributed system is a function from node names to local systems. See Sec. 5 for more details. One simple reason for building this language was to allow distinguishing between trusted and non-trusted components within a local system. In addition, an advantage of MoC is that it provides a language to devise more modular implementations and proofs than what Velisarios allows. In Asphalion, one can prove properties of sub-components and compose those to prove properties of local systems, and finally of distributed systems. In a Hoare logic-like fashion, those properties can be expressed as pre/post conditions that describe the inputs, pre-states, post-states, and outputs of the components or systems. When proving properties of a component C in isolation, one can simply abstract away the sub-components C relies on and instead assume properties about those sub-components. Also, thanks to MoC's support for deep embeddings, many properties of components can be derived automatically (see Sec. 5.4).

LoCK. As in Velisarios, in Asphalion we decided to rely on a knowledge theory to reason about distributed systems at a high-level of abstraction. Such theories have applications in many areas, such as, as mentioned in [3], economics, linguistics, artificial intelligence, theoretical computer science, and, evidently, distributed computing. One reason is that the way humans, machines, etc., manage to achieve tasks, or simply evolve is by making new discoveries and exchanging their knowledge so that others can know about it and benefit from it. Also, in our experience such theories match well with the way system experts informally reason about distributed systems. One immediate benefit of knowledge theories is that they allow reasoning about systems at a high-level of abstraction, and to focus on the fundamental reasons, in terms of knowledge, as why those systems are correct. In general, the abstract level of such theories allows reusing the results proved at that level in multiple applications.

That being said, Asphalion's knowledge calculus goes well beyond Velisarios's simple knowledge library, primarily for the following reasons: (1) In addition to learn and know operators (as in Velisarios), which allow reasoning about inputs and states at a high-level of abstraction, LoCK provides additional modalities, such as a dissemination modality, which allows reasoning about disseminated knowledge, i.e., outputs. (2) LoCK provides operators to reason about trusted knowledge. (3) As opposed to Velisarios's knowledge library, which is shallowly embedded in Coq, LoCK provides an abstraction barrier that cannot be broken because it is deeply embedded in Coq. In Velisarios, one has to be careful not to unfold the modalities to not break the abstraction barrier, which is more unwieldy. (4) LoCK comes with reasoning principles presented as primitive inference rules. Using these primitive rules, one can derive systems' properties within LoCK itself. There is no such clear separation between primitive and derived rules in Velisarios. (5) LoCK is a sequent calculus, for which we provided a semantics that interprets LoCK expressions as HyLoE formulas. Using this semantics, we proved the soundness of LoCK, in the sense that all its inference rules are valid.

Regarding Byzantine faults, as in Velisarios's knowledge library, handling Byzantine behavior within LoCK is mostly (but not entirely—see below) done through the modal operators it provides. For example, the semantics of LoCK's *learn* operator (as well as the definition of the *learn* operator in Velisarios's knowledge library) requires nodes to verify the authenticity of the pieces of knowledge they received in order to learn about them. In LoCK, this is not "visible" to the user thanks to the deep embedding of the calculus, while in Velisarios's knowledge library, it is not "visible" to the user as long as the user does not unfold these definitions. In addition, both theories provide an operator pertaining to Byzantine behavior, to essentially state that a node has a correct trace, in the sense that it has been correct so far. In Velisarios, this is done through an operator to directly state that a node has a correct trace, while this concept is *defined* within LoCK from more primitive constructs.

Summary. To summarize, Asphalion provides the following features over Velisarios:

- **(MoC,HyLoE,LoCK)** a notion of trusted components (see Sec. 5.1, 4.2, and 6)
- **(MoC)** a programming language to implement local systems as collections of interacting components, some of which are trusted, while the others are non-trusted (see Sec. 5.1)
- **(HyLoE)** a logic of events to reason about collections of both trusted and non-trusted components (see Sec. 4.2)
- **(LoCK)** a knowledge calculus to reason about collections of both trusted and non-trusted components (see Sec. 6)
- **(MoC)** support for compositional programming (see Sec. 5.1)
- **(MoC,HyLoE,LoCK)** support for compositional reasoning (see Sec. 5.2)
- **(MoC,HyLoE)** a mechanism to automatically derive properties of systems through deep embeddings (see Sec. 5.4)
- **(MoC,HyLoE)** a lifting mechanism to lift properties of trusted components to the level of local systems through deep embeddings (see Sec. 5.4)
- **(LoCK)** a lifting mechanism to lift properties of trusted components to the level of distributed systems (see Sec. 6.7)
- **(LoCK)** a knowledge calculus with primitive knowledge constructs, and primitive inference rules to reason about these constructs (see Sec. 6)
- **(LoCK)** a strictly enforced abstraction barrier (see Sec. 6)
- **(LoCK)** operators and rules to reason about trusted knowledge (see Sec. 6)

I WALK TROUGH THE CODE

As mentioned above, Asphalion relies on three novel languages, HyLoE, MoC and LoCK, and we proved the agreement property of two different implementations of the MinBFT protocol as case studies: a USIG-based version and a TrInc-based version. In this section, we provide a walk through Asphalion's code-base, by briefly describing the files that belong to HyLoE, the ones that belong to MoC, the ones that belong to LoCK, and the ones that are part of our MinBFT formalizations. We refer the reader to the `README.md` located in the root directory of our implementation for more information. In addition, Sec. J, provides a summary of the notation used in this paper, with pointers to our implementation.

HyLoE related file:

- `model/EventOrdering.v` contains HyLoE, our variant of Velisarios's logic of event, which also supports hybrid faults.

MoC related files:

- `model/ComponentSM.v` contains MoC, our monadic model of hybrid executable interacting components, which are shallowly embedded in Coq.
- `model/ComponentSM2.v` contains a deep embedding of a simple language of interacting components, that contains only return, bind, and call.
- `model/ComponentSM3.v` contains results regarding this simple language, most notably about lifting properties of (trusted) sub-components.

- `model/ComponentSM5.v` contains a deep embedding of a slightly more complex language of interacting components, that contains in addition to return, bind, and call constructor, a spawn constructor to spawn new components.
- `model/ComponentSM6.v` provides means to prove properties about collections of components compositionally.
- `model/ComponentSMExample1.v` and `model/ComponentSMExample2.v` contain simple examples of systems.
- `model/RunSM.v` contains a simulator for our component language.
- `model/ComponentAxiom.v` contains our main axiom regarding hybrid systems.

LoCK related files:

- `model/CalculusSM.v` contains our calculus of hybrid knowledge.
- `model/CalculusSM_derived.v` contains further rules.
- `model/CalculusSM_tacs.v` contains tactics that can be used within LoCK proofs.

MinBFT related files:

- `MinBFT/MinBFTheader.v` contains basic concepts necessary to implement MinBFT such as node names and messages.
- `MinBFT/USIG.v` contains an implementation of the USIG trusted component (this is currently loaded by `MinBFT/MinBFTg.v` because it also contains generic definitions such as the IO-interface of the trusted component, which is the same in both the USIG version and the TrInc version).
- `MinBFT/TrIncUSIG.v` contains an implementation of the TrInc trusted component (this is currently loaded by `MinBFT/MinBFTg.v` because it contains generic definitions).
- `MinBFT/MinBFTg.v` contains a generic definition of MinBFT (the MinBFT system, including its components—the USIG component is left abstract here), that can be instantiated for both the USIG version and the TrInc version.
- `MinBFT/MinBFTtacts.v` contains generic tactics that can be used to prove properties of our generic MinBFT implementation.
- `MinBFT/MinBFTkn0.v` contains a partial instantiation of our knowledge theory that can be used to prove properties of our generic MinBFT implementation.
- `MinBFT/MinBFTrep.v`, `MinBFT/MinBFTprops0.v`, `MinBFT/MinBFTbreak.v`, `MinBFT/MinBFTgen.v` contain simple generic definitions and properties about our generic MinBFT implementation.
- `MinBFT/MinBFTcount_gen1.v` to `MinBFT/MinBFTcount_gen5.v` contain complex (inductive) generic properties about our generic MinBFT implementation.
- `MinBFT/MinBFT.v` contains our USIG-based instantiation of our generic MinBFT implementation.
- `MinBFT/MinBFTcount.v` contains generic definitions and proofs of our USIG-based version of MinBFT that rely on the generic `MinBFT/MinBFTcount_gen` files.
- `MinBFT/MinBFTsubs.v`, `MinBFT/MinBFTstate.v`, `MinBFT/MinBFTbreak0.v`, `MinBFT/MinBFTtacts2,v`, `MinBFT/MinBFTprops1.v`, `MinBFT/MinBFTprops2.v`, `MinBFT/MinBFTview.v` are definitions and proofs concerning our USIG-based version of MinBFT.
- The `MinBFT/MinBFTass_` files contain proofs that the assumptions we made in the generic LoCK lemma we used to derive MinBFT's agreement property, are indeed correct.
- `MinBFT/MinBFTagreement.v` (and the more general `MinBFT/MinBFTagreement_iff.v`) contains a proofs of the agreement property of our USIG-based version of MinBFT.
- `MinBFT/TrInc.v` contains our TrInc-based instantiation of our generic MinBFT implementation.
- `MinBFT/TrInccount.v` contains generic definitions and proofs of our USIG-based version of MinBFT that rely on the generic `MinBFT/MinBFTcount_gen` files.

- MinBFT/TrIncsubs.v, MinBFT/TrIncstate.v, MinBFT/TrIncbreak.v, MinBFT/TrInctacts,v, MinBFT/TrIncprops1.v, MinBFT/TrIncprops2.v, MinBFT/TrIncview.v are definitions and proofs concerning our TrInc-based version of MinBFT.
- The MinBFT/TrIncass_ files contain proofs that the assumptions we made in the generic LoCK lemma we used to derive MinBFT's agreement property, are indeed correct.
- MinBFT/TrIncagreement.v (and the more general MinBFT/TrIncagreement_iff.v) contains a proofs of the agreement property of our TrInc-based version of MinBFT.

J NOTATION

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

HyLoE Notation	Meaning & File
Event	a set of events see Event field in the EventOrdering class (model/EventOrdering.v)
AuthData	a set of authenticated pieces of data see the AuthenticatedData record (model/Crypto.v)
Keys	a set of keys see class Keys (model/Crypto.v)
$\prec$	a causal ordering relation see happenedBefore field in the EventOrdering class (model/EventOrdering.v)
loc(e)	the location where the event e happens see loc field in the EventOrdering class (model/EventOrdering.v)
trigger(e)	explains why event e happened see trigger field in the EventOrdering class (model/EventOrdering.v)
TImsg(msg)	an event happened at a correct node that followed the given protocol see constructor trigger_info_data in the trigger_info (model/EventOrdering.v)
TItrust(it)	an event happened at a compromised node and the trusted component was called see constructor trigger_info_trusted in the trigger_info (model/EventOrdering.v)
TIarbitrary	an event happened at a compromised node and the trusted component was not called see constructor trigger_info_arbitrary in the trigger_info (model/EventOrdering.v)
pred(e)	local direct predecessor of e see direct_pred field in the EventOrdering class (model/EventOrdering.v)
keys(e)	the keys available at e see keys field in the EventOrdering class (model/EventOrdering.v)
nfo2auth(nfo)	a list of the authenticated pieces of data included in nfo see bind_op_list, get_contained_authenticated_data and trigger_op (model/EventOrdering.v)
first?(e) = true	pred(e) = None see definition isFirst (model/EventOrdering.v)
$e_1 \subset e_2$	pred(e_2) = Some(e_1) see direct_pred field in the EventOrdering class (model/EventOrdering.v)
pred$^=$(e)	e' if $e' \subset e$, and e otherwise see definition local_pred (model/EventOrdering.v)
$e_1 \preceq e_2$	$e_1 \prec e_2 \vee e_1 = e_2$ see definition happenedBeforeLe (model/EventOrdering.v)
$e_1 \sqsubset e_2$	$e_1 \prec e_2 \wedge$ loc(e_1) = loc(e_2) see definition localHappenedBefore (model/EventOrdering.v)
$e_1 \sqsubseteq e_2$	$e_1 \preceq e_2 \wedge$ loc(e_1) = loc(e_2) see definition localHappenedBeforeLe (model/EventOrdering.v)

Table 1. Summary of our HyLoE notation

MoC Notation	Meaning & File
$\mathcal{S}(cn)$	the type of the state of component cn see stateFun class (`model/ComponentSM.v`)
$\mathcal{I}(cn)$	the type of inputs of component cn see cio_I field in the ComponentIO record (`model/ComponentSM.v`)
$\mathcal{O}(cn)$	the type of output of component cn see cio_O field in the ComponentIO record (`model/ComponentSM.v`)
$\mathsf{Component}^n$	the collection of components at level n see definition n_proc (`model/ComponentSM.v`)
$\mathsf{M}^n(T)$	level n component monad of type T see definition M_n (`model/ComponentSM.v`)
$\mathsf{Upd}^n(cn)$	type of the update function of the component called cn see definition M_Update (`model/ComponentSM.v`)
$\mathsf{ret}(a)$	*return* operator of our component monad see definition ret (`model/ComponentSM.v`)
$m \gg= f$	*bind* operator of our component monad see definition bind (`model/ComponentSM.v`)
call	*call* operator of our component monad see definition call_proc (`model/ComponentSM.v`)
$ls@^{-}e$	local system ls after it has executed the list of events locally preceding e, excluding e see definition M_run_ls_before_event (`model/ComponentSM.v`)
$ls@^{+}e$	local system ls after it has executed the list of events locally preceding e, including e see definition M_run_ls_on_event (`model/ComponentSM.v`)
$ls\lfloor_{cn}$	accesses the state of a component named cn of a local system ls see definition state_of_component (`model/ComponentSM.v`)
$comp\lfloor_{cn}$	returns the component $comp$ if its name is cn, otherwise it is undefined see definition on_state_of_component (`model/ComponentSM.v`)
$ls@^{-}e\lfloor_{cn}$	returns the state of ls's component called cn before the event e see definition M_byz_state_ls_before_event_of_trusted (`model/ComponentSM.v`)
$ls@^{+}e\lfloor_{cn}$	returns the state of ls's component called cn after the event e see definition M_byz_state_ls_on_event_of_trusted (`model/ComponentSM.v`)
$S@^{-}e\lfloor_{cn}$	computes the state of a component cn of a system S before a given event e see definition M_byz_state_sys_before_event (`model/ComponentSM.v`)
$S@^{+}e\lfloor_{cn}$	computes the state of a component cn of a system S after a given event e see definition M_byz_state_sys_on_event (`model/ComponentSM.v`)
$ls \rightsquigarrow e$	returns the outputs of ls's main component at e when all the events preceding e are non-Byzantine, and returns the outputs of the trusted component otherwise see definition M_byz_output_ls_on_event (`model/ComponentSM.v`)
$S \rightsquigarrow e$	$S(\mathsf{loc}(e)) \rightsquigarrow e$ see definition M_byz_output_sys_on_event (`model/ComponentSM.v`)
$d \in ls \rightsquigarrow e$	the d occurs within the outputs computed by $ls \rightsquigarrow e$ this is simply the membership relation as one can see in (`model/ComponentSM.v`)
$\mathtt{RET}(a)$	*return* operator of our simple deep embedding (see Sec. 5.4) see constructor PROC_RET in the Proc (`model/ComponentSM2.v`)
$\mathtt{BIND}(p_1, p_2)$	*bind* operator of our simple deep embedding (see Sec. 5.4) see constructor PROC_BIND in the Proc (`model/ComponentSM2.v`)
$\mathtt{CALL}(cn, i)$	*call* operator of our simple deep embedding (see Sec. 5.4) see constructor PROC_CALL in the Proc (`model/ComponentSM2.v`)

Table 2. Summary of our MoC notation

LoCK Notation	Meaning
Data	a set of pieces of data see kc_data field in the KnowledgeComponents class (`model/CalculusSM.v`)
Trust	a set of trusted pieces of data see kc_trust field in the KnowledgeComponents class (`model/CalculusSM.v`)
Identifier	a set of data identifiers see kc_id field in the KnowledgeComponents class (`model/CalculusSM.v`)
trustHasId	relates trusted pieces of data and identifiers see kc_trust_has_id field in the KnowledgeComponents class (`model/CalculusSM.v`)
sys	the distributed system one wants to reason about see kc_sys field in the KnowledgeComponents class (`model/CalculusSM.v`)
mem	the name of the component holding the knowledge see kc_mem field in the KnowledgeComponents class (`model/CalculusSM.v`)
trust	the name of the trusted component see IOTrusted class (`model/EventOrdering.v`) and see trustedStateFun class (`model/ComponentSM.v`)
owner	identifies the node that generated a given piece of data see kc_data_owner field in the KnowledgeComponents class (`model/CalculusSM.v`)
verify(e, $auth$)	returns `true` if the authenticated piece of data $auth$ can indeed be authenticated at e, and `false` otherwise see definition kc_verify (`model/CalculusSM.v`)
genFor	relates trusted pieces of data and non-trusted pieces of data see kc_generated_for field in the KnowledgeComponents class (`model/CalculusSM.v`)
know	expresses what it means to hold some information see kc_knows field in the KnowledgeComponents class (`model/CalculusSM.v`)
trusted2id	returns the trusted identifier maintained by the trusted component see kc_trust_has_id field in the KnowledgeComponents class (`model/CalculusSM.v`)
initId	initial value of the identifier maintained by the trusted component see kc_init_id field in the KnowledgeComponents class (`model/CalculusSM.v`)
auth2data	extracts the pieces of data contained within an authenticated piece of data see kc_auth2data field in the KnowledgeComponents class (`model/CalculusSM.v`)
$\top$, $\bot$, $\land$, $\lor$, $\rightarrow$, $\exists$, $\forall$	standard first-order logic operators see constructors: KE_TRUE, KE_FALSE, KE_AND, KE_OR, KE_IMPLIES, KE_EX, KE_ALL, respectively (`model/CalculusSM.v`)
$\subset$, $\prec$, $\sqsubset$	HyLoE-specific operators to state properties relating different points in space/time see constructors: KE_RIGHT_BEFORE, KE_HAPPENED_BEFORE, KE_LOCAL_BEFORE, respectively (`model/CalculusSM.v`)
$\odot$	the HyLoE-specific operator to talk about initial event see constructor KE_FIRST (`model/CalculusSM.v`)
@	the HyLoE-specific operators to relate space/time coordinates see constructor KE_AT (`model/CalculusSM.v`)
$\mathcal{K}^+$	knows see constructor KE_KNOWS (`model/CalculusSM.v`)
$\mathcal{L}$	learns see constructor KE_LEARNS (`model/CalculusSM.v`)
$\mathcal{O}$	owns see constructor KE_HAS_OWNER (`model/CalculusSM.v`)
$\mathcal{D}$	disseminate see constructor KE_DISS (`model/CalculusSM.v`)
$\mathcal{I}^+$	knows identifier see constructor KE_ID_AFTER (`model/CalculusSM.v`)
$\mathcal{HI}$	has identifier see constructor KE_HAS_ID (`model/CalculusSM.v`)
$\mathcal{G}$	generated see constructor KE_GEN_FOR (`model/CalculusSM.v`)

$\mathcal{O}(d)$	"we" own the data d see definition KE_OWNS (model/CalculusSM.v)
$\mathcal{OD}(d)$	"we" disseminated the data d see definition KE_DISS_OWN (model/CalculusSM.v)
LID	if one learns some trusted piece data, it must have been disseminated by the corresponding trusted component see definition ASSUMPTION_learns_if_gen (model/CalculusSM.v)
KLD	if we know some trusted information, then we either knew it before, or we just learned it, or we just disseminated it see definition ASSUMPTION_learns_or_gen (model/CalculusSM.v)
Mon	the identifiers maintained by trusted components monotonically increase see definition ASSUMPTION_monotonicity (model/CalculusSM.v)
New	an identifier generated by a trusted component i must be between the one it recorded before and the one it recorded after it generated i see definition ASSUMPTION_generates_new (model/CalculusSM.v)
Uniq	a trusted pieces of data disseminated by a trusted component at a given point in space/time must be unique see definition ASSUMPTION_disseminates_unique (model/CalculusSM.v)
$\exists_{\mathtt{i}} f$, $\exists_{\mathtt{d}} f$, $\exists_{\mathtt{t}} f$, $\exists_{\mathtt{n}} f$	$\exists\langle\mathtt{KTi}, f\rangle$, $\exists\langle\mathtt{KTd}, f\rangle$, $\exists\langle\mathtt{KTt}, f\rangle$, and $\exists\langle\mathtt{KTn}, f\rangle$, respectively i.e., existential quantifier for our different kinds of values see KE_EX_ID, KE_EX_DATA, KE_EX_TRUST, and KE_EX_NODE, respectively (model/CalculusSM.v)
$\forall_{\mathtt{i}} f$, $\forall_{\mathtt{d}} f$, $\forall_{\mathtt{t}} f$, $\forall_{\mathtt{n}} f$	$\forall\langle\mathtt{KTi}, f\rangle$, $\forall\langle\mathtt{KTd}, f\rangle$, $\forall\langle\mathtt{KTt}, f\rangle$, and $\forall\langle\mathtt{KTn}, f\rangle$, respectively i.e., universal quantifier for our different kinds of values see KE_ALL_ID, KE_ALL_DATA, KE_ALL_TRUST, and KE_ALL_NODE, respectively (model/CalculusSM.v)
$\exists_{\mathtt{i}}\lambda i_1,\ldots,i_n.\tau$	$\exists_{\mathtt{i}}\lambda i.\ldots\exists_{\mathtt{i}}\lambda i_n.\tau$, i.e., universal multi-quantifier for node (and similarly for the other values) see KE_EX_IDs (model/CalculusSM.v)
$\forall_{\mathtt{i}}\lambda i_1,\ldots,i_n.\tau$	$\forall_{\mathtt{i}}\lambda i.\ldots\forall_{\mathtt{i}}\lambda i_n.\tau$, i.e., universal multi-quantifier for node (and similarly for the other values) see KE_ALL_IDs (model/CalculusSM.v)
$\neg\tau$	negation see KE_NOT (model/CalculusSM.v)
$\preceq\tau$	happened before or equal, i.e., $\prec\tau \lor \tau$ see KE_HAPPENED_BEFORE_EQ (model/CalculusSM.v)
$\sqsubseteq\tau$	happened locally before or equal, i.e., $\sqsubset\tau \lor \tau$ see KE_LOCAL_BEFORE_EQ (model/CalculusSM.v)
$\subseteq\tau$	direct predecessor or equal, i.e., $\subset\tau \lor (\tau \land \odot)$ see KE_RIGHT_BEFORE_EQ (model/CalculusSM.v)
$i_1 \leq i_2$	identifier is less than or equal to, i.e., $i_1 < i_2 \lor i_1 = i_2$ see KE_ID_LE (model/CalculusSM.v)
$[\![\tau]\!]_e$	interpretation of LoCK expressions see interpret (model/CalculusSM.v)
$\langle G\rangle\, H \vdash \sigma$	syntax of sequents see MkSeq (model/CalculusSM.v)
H_1, H_2	append operation on sequent hypotheses this is simply the append operation on lists (model/CalculusSM.v)

Table 3. Summary of our LoCK notation

Paper name	Implementation name
$\square_{\mathrm{E}}$ for ($\prec$)	see `PRIMITIVE_RULE_unhappened_before_hyp` in model/CalculusSM.v
$\square_{\mathrm{I}}$ for ($\prec$)	see `PRIMITIVE_RULE_unhappened_before_if_causal` in model/CalculusSM.v

$\square_E$ for ($\sqsubset$)	see DERIVED_RULE_unlocal_before_hyp in model/CalculusSM.v[52]
$\square_I$ for ($\sqsubset$)	see DERIVED_RULE_unlocal_before_if_causal in model/CalculusSM.v
$\square_{\mathtt{It}}$	see PRIMITIVE_RULE_unhappened_before_if_causal_trans_eq in model/CalculusSM.v
if¬⊙	see PRIMITIVE_RULE_introduce_direct_pred in model/CalculusSM.v
if⊙	see PRIMITIVE_RULE_introduce_direct_pred_eq in model/CalculusSM.v
weak for ($\prec$, $\preceq$)	see PRIMITIVE_RULE_causal_if_causalle_true in model/CalculusSM.v
weak for ($\sqsubset$, $\sqsubseteq$)	see PRIMITIVE_RULE_local_if_localle in model/CalculusSM.v
weak for ($\sqsubset$, $\prec$)	see PRIMITIVE_RULE_local_if_causal in model/CalculusSM.v
weak for ($\sqsubseteq$, $\preceq$)	see PRIMITIVE_RULE_localle_if_causalle in model/CalculusSM.v
weak for ($\subset$, $\sqsubset$)	see PRIMITIVE_RULE_direct_pred_if_local_pred in model/CalculusSM.v
weak for ($\equiv$, $\sqsubseteq$)	see PRIMITIVE_RULE_localle_if_eq in model/CalculusSM.v
sub_H	see PRIMITIVE_RULE_subst_causal_eq_hyp in model/CalculusSM.v
sub_C	see PRIMITIVE_RULE_subst_causal_eq_concl in model/CalculusSM.v
$\equiv_{\mathtt{refl}}$	see PRIMITIVE_RULE_add_eq_refl in model/CalculusSM.v
¬⊙	see PRIMITIVE_RULE_not_first in model/CalculusSM.v
$\odot_{\mathtt{dec}}$	see PRIMITIVE_RULE_first_dec in model/CalculusSM.v
ind	see PRIMITIVE_RULE_pred_induction in model/CalculusSM.v
tri	see PRIMITIVE_RULE_tri_if_same_loc in model/CalculusSM.v
sym	see PRIMITIVE_RULE_id_eq_sym in model/CalculusSM.v
trans for (=, <, <)	see PRIMITIVE_RULE_id_lt_trans_eq_lt in model/CalculusSM.v
trans for (<, =, <)	see PRIMITIVE_RULE_id_lt_trans_lt_eq in model/CalculusSM.v
trans for (<, <, <)	see PRIMITIVE_RULE_id_lt_trans_lt_lt in model/CalculusSM.v
trans for (=, =, =)	see PRIMITIVE_RULE_id_eq_trans_true in model/CalculusSM.v
$\mathsf{K}_{\mathtt{dec}}$	see PRIMITIVE_RULE_decidable_knows in model/CalculusSM.v
irrefl	see PRIMITIVE_RULE_id_lt_elim in model/CalculusSM.v
1owner	see PRIMITIVE_RULE_has_owner_implies_eq in model/CalculusSM.v
1data	see PRIMITIVE_RULE_collision_resistant in model/CalculusSM.v
1id	see PRIMITIVE_RULE_ids_after_imply_eq_ids in model/CalculusSM.v
$\top_I$	see PRIMITIVE_RULE_true in model/CalculusSM.v
$\bot_E$	see PRIMITIVE_RULE_false_elim in model/CalculusSM.v
$\rightarrow_E$	see PRIMITIVE_RULE_implies_elim in model/CalculusSM.v
$\rightarrow_I$	see PRIMITIVE_RULE_implies_intro in model/CalculusSM.v
$\vee_E$	see PRIMITIVE_RULE_or_elim in model/CalculusSM.v
$\vee_{\mathtt{Il}}$	see PRIMITIVE_RULE_or_intro_left in model/CalculusSM.v
$\vee_{\mathtt{Ir}}$	see PRIMITIVE_RULE_or_intro_right in model/CalculusSM.v
$\wedge_E$	see PRIMITIVE_RULE_and_elim in model/CalculusSM.v
$\wedge_I$	see PRIMITIVE_RULE_and_intro in model/CalculusSM.v
$\exists_E$ (for KTi)	see PRIMITIVE_RULE_exists_id_elim in model/CalculusSM.v
$\exists_E$ (for KTd)	see PRIMITIVE_RULE_exists_data_elim in model/CalculusSM.v
$\exists_E$ (for KTt)	see PRIMITIVE_RULE_exists_trust_elim in model/CalculusSM.v
$\exists_E$ (for KTn)	see PRIMITIVE_RULE_exists_node_elim in model/CalculusSM.v
$\exists_I$ (for KTi)	see PRIMITIVE_RULE_id_count_intro in model/CalculusSM.v
$\exists_I$ (for KTd)	see PRIMITIVE_RULE_exists_data_intro in model/CalculusSM.v
$\exists_I$ (for KTt)	see PRIMITIVE_RULE_exists_trust_intro in model/CalculusSM.v
$\exists_I$ (for KTn)	see PRIMITIVE_RULE_exists_node_intro in model/CalculusSM.v
$\forall_E$ (for KTi)	see PRIMITIVE_RULE_all_id_elim in model/CalculusSM.v
$\forall_E$ (for KTd)	see PRIMITIVE_RULE_all_data_elim in model/CalculusSM.v
$\forall_E$ (for KTt)	see PRIMITIVE_RULE_all_trust_elim in model/CalculusSM.v
$\forall_E$ (for KTn)	see PRIMITIVE_RULE_all_node_elim in model/CalculusSM.v
$\forall_I$ (for KTi)	see PRIMITIVE_RULE_all_id_intro in model/CalculusSM.v

[52]This rule, as well as $\square_I$, are not primitive anymore because $\sqsubset$ is not a primitive operator of LoCK anymore. However, we still present it as such for simplicity (see KE_LOCAL_BEFORE in model/CalculusSM.v).

$\forall_{\mathtt{I}}$ (for KTd)	see PRIMITIVE_RULE_all_data_intro in model/CalculusSM.v
$\forall_{\mathtt{I}}$ (for KTt)	see PRIMITIVE_RULE_all_trust_intro in model/CalculusSM.v
$\forall_{\mathtt{I}}$ (for KTn)	see PRIMITIVE_RULE_all_node_intro in model/CalculusSM.v
$\subset_{\mathtt{E}}$	see PRIMITIVE_RULE_unright_before_hyp in model/CalculusSM.v
$\Box_{\mathtt{E}}$ for $\subset$	see PRIMITIVE_RULE_unright_before_hyp_if_causal in model/CalculusSM.v
$\Box_{\mathtt{I}}$ for $\subset$	see PRIMITIVE_RULE_unright_before_if_causal in model/CalculusSM.v
$\mathtt{STR}_{\sqsubseteq}$	see PRIMITIVE_RULE_split_local_before_eq2 in model/CalculusSM.v
$\mathtt{STR}_{\preceq}$	see PRIMITIVE_RULE_split_happened_before_eq2 in model/CalculusSM.v
$\mathtt{STR1}_{\preceq}$	see PRIMITIVE_RULE_at_implies_localle in model/CalculusSM.v
$\mathtt{STR1}_{\prec}$	see PRIMITIVE_RULE_at_implies_local in model/CalculusSM.v
$\mathtt{split}_{\sqsubset}$	see PRIMITIVE_RULE_split_local_before in model/CalculusSM.v
$\mathtt{splitPred}_{\sqsubset}$	see PRIMITIVE_RULE_split_local_before2 in model/CalculusSM.v
$\equiv_{\mathtt{sym}}$	see PRIMITIVE_RULE_causal_eq_sym in model/CalculusSM.v
$\equiv_{\mathtt{pred}=}$	see PRIMITIVE_RULE_weaken_direct_pred_to_local_pred in model/CalculusSM.v
$@_{\mathtt{loc}}$	see PRIMITIVE_RULE_at_change_localle in model/CalculusSM.v
loc	see PRIMITIVE_RULE_at_implies_same_node in model/CalculusSM.v
change for $i_1 = i_2$	see PRIMITIVE_RULE_id_eq_change_event in model/CalculusSM.v
change for $d_1 = d_2$	see PRIMITIVE_RULE_data_eq_change_event in model/CalculusSM2.v
change for $t_1 = t_2$	see PRIMITIVE_RULE_trust_eq_change_event in model/CalculusSM.v
change for $\boldsymbol{a}_1 = \boldsymbol{a}_2$	see PRIMITIVE_RULE_node_eq_change_event in model/CalculusSM2.v
change for $i_1 < i_2$	see PRIMITIVE_RULE_id_lt_change_event in model/CalculusSM.v
change for $\mathcal{HI}(t, i)$	see PRIMITIVE_RULE_has_id_change_event in model/CalculusSM.v
change for $\mathcal{O}(d, \boldsymbol{a})$	see PRIMITIVE_RULE_has_owner_change_event in model/CalculusSM.v
change for $\mathcal{G}(d, t)$	see PRIMITIVE_RULE_gen_for_change_event in model/CalculusSM.v
valSub for $(\mathcal{HI}(t, i))$	see PRIMITIVE_RULE_trust_has_id_subst in model/CalculusSM.v
valSub for $(\mathcal{O}(d, \boldsymbol{a}))$	see PRIMITIVE_RULE_subst_node_in_has_owner in model/CalculusSM.v

Table 4. Pointers to our rules

K FURTHER RELATED WORK

In addition to the logics, models, and tools mentioned in Sec. 9, there are many more systems, tools, and techniques related to our work on hybrid systems. We mention some of those below.

K.1 Trustworthy Component-Based Programs

Orthogonal but complementary to our work is the one done on guaranteeing the trustworthiness of component-based local programs: in our work we assume that trusted local components cannot be compromised, and derive distributed properties from the properties of these components. Let us mention here a few relevant projects.

CAmkES [4, 5, 6, 7, 8] is a component based platform to reason about embedded systems built on top of seL4 [9]. It supports compositional programming and verification, and automatically generates verified "glue" code to connect the different components of a system.

SCC/RSCC [10, 11] are secure compartimentalizing compilation techniques for unsafe languages such as C. Applications are divided into components that communicate via procedure calls, and the compiler ensures that compromised components cannot contaminate the other components.

K.2 Verification of Distributed Systems

Actor Services [12] allows verifying the distributed and functional properties of programs communicating via asynchronous message passing at the level of the source code (they use a simple Java-like language). It supports modular reasoning and proving liveness. To the best of our knowledge, it does not deal with faults.

Chapar [13] is a framework for modular certification of causal consistency of replicated key-value store implementations and their client programs. The framework is written in Coq, allowing to extract OCaml code (which implies that there is no gap between the verified and the executed code). Moreover, Chapar includes a model checker, which can be used to check results on the client side. Using Chapar, the authors proved the causal-consistency of two key-value stores. As opposed to Asphalion, Chapar relies on the distributed snapshot semantics of distributed systems, and is specifically tailored to reason about causal consistency (as opposed to the strong linearizability consistency property of BFT-SMR protocols).

Sally [14] is a model checker for infinite-state systems that can automatically discover k-inductive strengthening of properties. It has been used to check properties of synchronous Byzantine fault-tolerant protocols.

Aneris [15, 16] is a higher-order, concurrent separation logic that supports modular reasoning of distributed systems through a novel technique called node-local reasoning. This technique allows reasoning about each node of a system in isolation, and then combining those to prove properties of the entire system. Using Aneris, the authors, among other things, proved correct an implementation of two-phase commit.

K.3 Interfacing With Trusted Components

Orthogonal but related to our work, many models, systems, and tools have been developed to provide safe and secure interfaces between trusted components and payload systems. Given the fact that IBM's CCA API is a standard API used by banks, many researchers have focused on studying whether it is secure [17, 18, 19, 20, 21]. Many other generic model checking-based bug finding tools have been develop to ensure that APIs are secure, such as [22, 23, 24].

Moreover, *temporal rules* are a standard technique to ensure that clients can only use APIs in a safe manner [25, 26].

K.4 Trusted Component/Environment Verification

Several new *trusted environments* have been developed this past decade, such as [27, 28, 29]. As a result, many papers [30, 31, 32, 33, 34, 35, 36, 34, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46], to mention only a few, concentrated on proving different properties about these trusted components/environments (e.g. confidentiality, integrity, linearizability, remote equivalence). Although, some of these papers were about proving properties of the security protocols, e.g. [44, 45, 46], to the best of our knowledge none of them is about proving properties of BFT-SMR protocol.

APPENDIX REFERENCES

[1] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *USENIX ATC 2017*. USENIX Association, 2017, pp. 645–658. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

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

[3] Ronald Fagin, Joseph Halpern, Yoram Moses, and Moshe Vardi. *Reasoning About Knowledge*. Jan. 2003.

[4] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. "CAmkES: A component model for secure microkernel-based embedded systems". In: *Journal of Systems and Software* 80.5 (2007), pp. 687–699. URL: https://doi.org/10.1016/j.jss.2006.08.039.

[5] Matthew Fernandez, Gerwin Klein, Ihor Kuz, and Toby Murray. *CAmkES Formalisation of a Component Platform*. Tech. rep. Australia: NICTA and UNSW, Nov. 2013.

[6] Matthew Fernandez, Peter Gammie, June Andronick, Gerwin Klein, and Ihor Kuz. *CAmkES Glue Code Semantics*. Tech. rep. Australia: NICTA and UNSW, Nov. 2013.

[7] Matthew Fernandez. "Formal Verification of a Component Platform". PhD thesis. Sydney, Australia: UNSW Computer Science & Engineering, July 2016.

[8] Gerwin Klein, June Andronick, Matthew Frenandez, Ihor Kuz, Toby Murray, and Gernot Heiser. "Formally Verified Software in the Real World". Communicatons of the ACM. 2018. URL: http://ssrg.nicta.com/publications/csiro_full_text/Klein_AKMHF_toappear.pdf.

[9] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *SOSP 2009*. ACM, 2009, pp. 207–220. URL: http://doi.acm.org/10.1145/1629575.1629596.

[10] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. "Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation". In: *CSF 2016*. IEEE Computer Society, 2016, pp. 45–60. URL: https://doi.org/10.1109/CSF.2016.11.

[11] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. "When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise". In: *CCS 2018*. ACM, 2018, pp. 1351–1368. URL: http://doi.acm.org/10.1145/3243734.3243745.

[12] Alexander J. Summers and Peter Müller. "Actor Services - Modular Verification of Message Passing Programs". In: *ESOP 2016*. Vol. 9632. LNCS. Springer, 2016, pp. 699–726. URL: https://doi.org/10.1007/978-3-662-49498-1_27.

[13] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. "Chapar: certified causally consistent distributed key-value stores". In: *POPL 2016*. ACM, 2016, pp. 357–370. URL: http://doi.acm.org/10.1145/2837614.2837622.

[14] Bruno Dutertre, Dejan Jovanovic, and Jorge A. Navas. "Verification of Fault-Tolerant Protocols with Sally". In: *NFM 2018*. Vol. 10811. LNCS. Springer, 2018, pp. 113–120. URL: https://doi.org/10.1007/978-3-319-77935-5%5C_8.

[15] Moren Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, and Lars Birkedal. "Aneris: A Logic for Node-Local, Modular Reasoning ofDistributed Systems". 2018. URL: https://iris-project.org/pdfs/2019-aneris-submission.pdf.

[16] Morten Krogh-Jespersen. "Towards Modular ReasoningforStateful and ConcurrentPrograms". PhD thesis. Aarhus University, 2018.

[17] Gavin Keighren. "Model Checking Security APIs". PhD thesis. School of informatics, Univeristy of Edinburg, 2006.

[18] Ronald Togl. "On Trusted Computing Interfaces". PhD thesis. Faculty of Computer Science, Graz University of Technology, 2013.

[19] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross J. Anderson. "Robbing the Bank with a Theorem Prover - (Abstract)". In: *SP 2007*. Vol. 5964. LNCS. Springer, 2007, p. 171. URL: https://doi.org/10.1007/978-3-642-17773-6_21.

[20] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. *Robbing the bank with a theorem prover*. Tech. rep. UCAM-CL-TR-644. University of Cambridge, Computer Laboratory, Aug. 2005. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-644.pdf.

[21] Judicaël Courant and Jean-François Monin. "Defending the bank with a proof assistant". In: *WITS 2006*. In WITS proceedings. Vienna, Mar. 2006.

[22] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. "Automatic discovery of API-level exploits". In: *ICSE 2005*. ACM, 2005, pp. 312–321. URL: http://doi.acm.org/10.1145/1062455.1062518.

[23] Hao Chen and David A. Wagner. "MOPS: an infrastructure for examining security properties of software". In: *CCS 2002*. ACM, 2002, pp. 235–244. URL: http://doi.acm.org/10.1145/586110.586142.

[24] Athanasios (Thanassis) Avgerinos. "Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs". PhD thesis. Carnegie Mellon University, 2014.

[25] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. "Synthesis of interface specifications for Java classes". In: *POPL 2005*. ACM, 2005, pp. 98–109. URL: http://doi.acm.org/10.1145/1040305.1040314.

[26] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. “A decade of software model checking with SLAM”. In: *Commun. ACM* 54.7 (2011), pp. 68–76. URL: http://doi.acm.org/10.1145/1965724.1965743.
[27] *TPM*. URL: https://trustedcomputinggroup.org/work-groups/trusted-platform-module/.
[28] *SGX*. 2019. URL: https://software.intel.com/en-us/sgx.
[29] *ARM TrustZone*. 2019. URL: https://www.arm.com/products/security-on-arm/trustzone.
[30] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. “Ironclad Apps: End-to-End Security via Automated Full-System Verification”. In: *OSDI ’14*. USENIX Association, 2014, pp. 165–181. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel.
[31] Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. “Moat: Verifying Confidentiality of Enclave Programs”. In: *CCS 2015*. ACM, 2015, pp. 1169–1184. URL: http://doi.acm.org/10.1145/2810103.2813608.
[32] Aaron Bembenek, Lily Tsai, and Ezra Zigmond. “Better Trust Zone : Verifying Security of Enclave-Aware Calculi”. In: 2017.
[33] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using verification to disentangle secure-enclave hardware from software”. In: *SOSP 2017*. ACM, 2017, pp. 287–305. URL: http://doi.acm.org/10.1145/3132747.3132782.
[34] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kirli Kaynar. “A Logic of Secure Systems and its Application to Trusted Computing”. In: *SP 2009*. IEEE Computer Society, 2009, pp. 221–236. URL: https://doi.org/10.1109/SP.2009.16.
[35] Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. “Verifying Linearizability of Intel® Software Guard Extensions”. In: *CAV 2015*. Vol. 9207. LNCS. Springer, 2015, pp. 144–160. URL: https://doi.org/10.1007/978-3-319-21668-3_9.
[36] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. “Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis”. In: *ASPLOS 2017*. ACM, 2017, pp. 555–568. URL: http://doi.acm.org/10.1145/3037697.3037739.
[37] Limin Jia, Shayak Sen, Deepak Garg, and Anupam Datta. “A Logic of Programs with Interface-Confined Code”. In: *CSF 2015*. IEEE Computer Society, 2015, pp. 512–525. URL: https://doi.org/10.1109/CSF.2015.38.
[38] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. “A Formal Foundation for Secure Remote Execution of Enclaves”. In: *CCS 2017*. ACM, 2017, pp. 2435–2450. URL: http://doi.acm.org/10.1145/3133956.3134098.
[39] Jianxiong Shao, Yu Qin, and Dengguo Feng. “Formal analysis of HMAC authorisation in the TPM2.0 specification”. In: *IET Information Security* 12.2 (2018), pp. 133–140. URL: https://doi.org/10.1049/iet-ifs.2016.0005.
[40] Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. “Formal Analysis of Enhanced Authorization in the TPM 2.0”. In: *ASIA CCS’15*. ACM, 2015, pp. 273–284. URL: http://doi.acm.org/10.1145/2714576.2714610.
[41] Guangdong Bai, Jianan Hao, Jianliang Wu, Yang Liu, Zhenkai Liang, and Andrew P. Martin. “TrustFound: Towards a Formal Foundation for Model Checking Trusted Computing Platforms”. In: *FM 2014*. Vol. 8442. LNCS. Springer, 2014, pp. 110–126. URL: https://doi.org/10.1007/978-3-319-06410-9%5C_8.
[42] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. “A Formal Analysis of Authentication in the TPM”. In: *FAST 2010*. Vol. 6561. LNCS. Springer, 2010, pp. 111–125. URL: https://doi.org/10.1007/978-3-642-19751-2%5C_8.
[43] Bai Guangdong. “Formally Analyzing and Verifying Secure System Design and Implementation”. PhD thesis. National Univeristy of Singapore, 2015.
[44] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. “Formal Analysis of Protocols Based on TPM State Registers”. In: *CSF 2010*. IEEE Computer Society, 2011, pp. 66–80. URL: https://doi.org/10.1109/CSF.2011.12.
[45] Sergiu Bursuc, Christian Johansen, and Shiwei Xu. “Automated Verification of Dynamic Root of Trust Protocols”. In: *POST 2017*. Vol. 10204. LNCS. Springer, 2017, pp. 95–116. URL: https://doi.org/10.1007/978-3-662-54455-6%5C_5.

[46] Shiwei Xu, Sergiu Bursuc, and Julian P. Murphy. “New abstractions in applied pi-calculus and automated verification of protected executions”. In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 686. URL: http://eprint.iacr.org/2013/686.