Modeling and Verification of Leaders Agreement in the Intrusion-Tolerant Enclaves Using PVS

Mohamed Layouni[1], Jozef Hooman[2] and Sofiène Tahar[1]

[1] Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
{layouni,tahar}@ece.concordia.ca
[2] Computing Science Department,
University of Nijmegen, Nijmegen, The Netherlands
hooman@cs.kun.nl

Abstract. Enclaves is a group-oriented intrusion-tolerant protocol. Intrusion-tolerant protocols are cryptographic protocols that implement fault-tolerance techniques to achieve security despite possible intrusions at some parts of the system. Among the most tedious faults to handle in security are the so-called Byzantine faults, where insiders maliciously exhibit an arbitrary (possibly dishonest) behavior during executions of the protocol. This class of faults poses formidable challenges to current verification techniques and has been formally verified only in simplified forms and under restricted fault assumptions. In this paper we present our work on the formal verification of the Byzantine fault-tolerant Enclaves [1] protocol. We use PVS to formally specify and prove Proper Byzantine Agreement, Agreement Termination and Integrity.

1 Introduction

We have seen in the last decade a substantial progress in the formal verification of cryptographic protocols. A wide variety of techniques have been developed to verify a number of key security properties ranging from *confidentiality*, *authentication* to *atomic transactions* and *non-repudiation* [2–5]. Nevertheless, all the focus was either on two-party protocols (i.e. involving only a pair of users) or, in the best cases, on group protocols with centralized leadership (i.e. a presumably trusted fault-free server managing a group of users). In the present work, we are concerned with the verification of the intrusion-tolerant Enclaves [1]: a group-membership protocol with a distributed leadership architecture, where the authority of the traditional single server is shared among a set of n independent elementary servers, f of which at most could fail at the same time. The protocol has a maximum resilience of one third (i.e. $f \leq \lfloor \frac{n-1}{3} \rfloor$) and uses a similar algorithm to the consistent broadcast of Bracha and Toueg [8].

The primary goal of Enclaves is to preserve an acceptable group-membership service of the overall system despite intrusions at some of its subparts. For instance, an authorized user u who requests to join an active group of users should be eventually accepted, despite the fact that faulty leaders may coordinate their messages in such a way as to mislead non-faulty leaders (the majority) into disagreement, and thus into rejecting user u.

To achieve its intrusion-tolerant capabilities, Enclaves relies on the combination of a cryptographic authentication protocol, a Byzantine fault-tolerant leader agreement protocol and a secret sharing scheme. Although we assume the underlying cryptographic primitives and fault-tolerant components to be perfect, one cannot easily guarantee security of the whole protocol. In fact, several protocols had been long thought to be secure until a simple attack was found (see [20] for a survey). Therefore, the question of whether or not a protocol actually achieves its security goals becomes paramount. To date, most of the research in protocol analysis has been devoted to finding attacks on known, either two-party or centralized protocols. In this paper we are concerned with the verification of a distributed multi-leader group communication protocol.

Enclaves is intended to tolerate Byzantine faults [7]. Modeling Byzantine behavior has been always a big issue in formal verification. It arises the problem of how much power should be given to a Byzantine fault and how general the model should be to capture the arbitrary nature of a Byzantine fault behavior. These questions have been extensively studied [11–13] and continue to be a center of focus. In this paper faults are only limited by cryptographic constraints. For instance, they can arbitrarily send random messages, reset their local clocks and perform any action without satisfying its preconditions. Faults, however, cannot decrypt a message without having the appropriate key, or impersonate other participants by forging cryptographic signatures. More details about our fault assumptions are discussed in Section 2.

In this paper is we discuss a formal analysis of the Byzantine fault-tolerant leaders agreement module used of Enclaves. This module relies, to a large extent, on the timing and the coordination of a set of distributed actions, possibly performed by faulty processes whose behavior is hard to assess in any automatic verification tool. Therefore, we found it more convenient to proceed by means of theorem proving. In fact, we use PVS [14] and formalize the protocol in the style of Timed-Automata [9]. This formalism makes it easy to express timing constraints on transitions. It also captures several useful aspects of real-time systems such as liveness, periodicity and bounded timing delays. Using this formalism, we specified the protocol for any instance of size n, and we proved safety and liveness properties such as Proper Agreement, Agreement Termination and Integrity.

The remainder of this paper is organized as follows. In Section 2, we give an overview of the Enclaves protocol architecture and goals, and we explicitly state our system model assumptions. In Section 3, we present how we model the elementary components of the Byzantine leader agreement module in PVS and how we build the final protocol model out of these ingredients. In Section 4, we formulate and prove our theorems. In Section 5, we discuss some related work. Finally in Section 6, we comment our results and state some perspectives for a future work.

2 Overview on the Enclaves Protocol

Enclaves [1] is a protocol that enables users to share information and collaborate securely through insecure networks such as the Internet. Enclaves provides services for building and managing groups of users. Access to a given group is granted only to sets of users who have the right credentials to do so. Authorized users can dynamically and at their will join, leave, and rejoin an active group.

The group communication service relies on a secure multicasting channel that ensures integrity and confidentiality of group communication. All messages sent by a group member are encrypted and delivered to all the other group members.

The group-management service consists of user authentication, access control, and group-key distribution. Figure 1 shows the different phases of the protocol execution. Initially at time t_0, user u sends requests to join the group to a set of leaders. These leaders locally authenticate u within time interval $[t_1, t_2]$. When done, the agreement procedure starts and terminates at time t_4 by reaching a consensus as whether or not to accept user u. Finally on acceptance, user u is provided with the current group composition, as well as the group-key. Once in the group, each member is notified when a new user joins or a member leaves the group in such a way that all members are in possession of a consistent image of the current group-key holders.

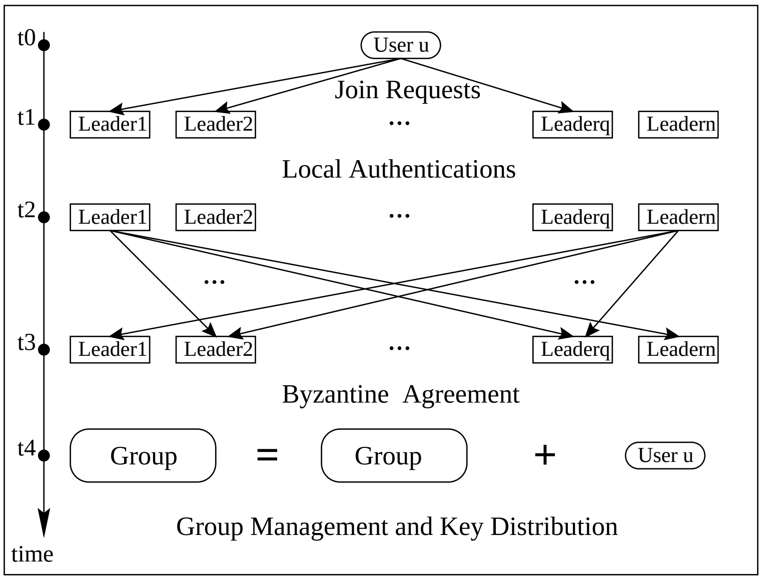

Fig. 1. Protocol execution

In summary, we prove that Enclaves satisfies the *Proper authentication and access control* requirement even in the presence of f compromised leaders. The latter requirement states that only authorized users can join the application and an authorized user cannot be prevented from joining the application. This has been established in PVS through the Proper Agreement, Agreement Termination and Integrity theorems (Sections 3).

The description of Enclaves in [1] assumes a reliable network where messages eventually reach their destinations within an upper bound delivery time. In this paper we make the same assumptions. Concerning the intruder, we adopt a standard model where an intruder fully monitors the network, proactively augments its knowledge, and chooses to send, either adaptively or randomly, messages on the network. The intruder, however, cannot block messages from reaching their destination and is limited by cryptographic constraints. For instance, the intruder cannot decrypt messages without having the right key, or impersonating other participants by forging cryptographic signatures. Given the above settings, we assume the cryptography layer to be perfect (i.e. messages format is well chosen to prevent any leakage of sensitive information), and we concentrate rather on the Byzantine fault-tolerance capabilities of the protocol.

Next, we formalize the elementary components of the Byzantine leader agreement module in PVS and we build the final protocol model out of these ingredients. Then in Section 4, we formulate and prove our theorems.

3 Modeling Intrusion-Tolerant Enclaves in PVS

Most group communication protocols, including Enclaves, can be modeled by an automaton whose initial state is modified by the participants' actions as the group mutates (e.g., new members join). Because Enclaves depends also on time (participants timeout, timestamp group views etc.), it is natural to model it as a timed automaton. Participants in a typical run of Enclaves consist of a set of n leaders (f of which are faulty), a group of members, and one or more users requiring to join the group. Similarly to the PAXOS protocol in [16], the leaders communicate with each others and with users via a partially asynchronous network. Messages sent on this network are assumed to be eventually delivered to their destinations within an upper bound of time, but no assumption is made on the reception order.

In the remainder of this section, we first explain our general PVS theory about timed automata. The parameters of this theory are used here to formalize Enclaves by defining the actions, the states, and the preconditions and effects of each action. Finally, the resulting executions of the protocol and fault assumptions are described.

3.1 Timed Automata Model in PVS

We present a general, protocol-independent, theory called `TimedAutomata`. Given a number of parameters, which represent the essential characteristics of a protocol, it defines all possible executions of the protocol as a set of `Runs`. A *run* is a sequence of the form $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} s_2 \overset{a_2}{\to} s_3 \overset{a_3}{\to} \ldots$ where the s_i are *states*, representing a snapshot of the system during execution and the a_i are the executed *actions*. A particular protocol is characterized by sets of possible `States` and `Actions`, a condition `Init` on the initial state, the precondition `Pre` of each state, expressing which states can be executed, the effect `Effect` of each action, expressing the possible state changes by the action, and a function `now` which gives the current time in each state. In a typical application, there is a special *delay* action which models the passage of time and increases the value of `now`. All other actions do not change time. In PVS, the theory and its parameters are defined as follows[1].

```
TimedAutomata  [ States, Actions: TYPE+,
                 Init : pred[States],
                 Pre : [Actions -> pred[States]] ,
                 Effect  :  pred[[States, Actions, States]],
                 now : [States -> nonneg_real]
                ] : THEORY
```

To define runs, let `PreRuns` be a record with two fields, `states` and `events`.

```
PreRuns : TYPE = [# states  : sequence[States],
                    events  : sequence[Actions] #]
```

A `Run` is a `PreRun` where the first state satisfies `Init`, the precondition and effect predicates of all actions are satisfied, the current time never decreases and increases above any arbitrary bound (avoiding Zeno-behaviour [6]). In PVS this is formalized as follows.

```
PreEffectOK(pr) : bool = FORALL i :
   Pre(events(pr)(i)) (states(pr)(i)) AND
   Effect(states(pr)(i), events(pr)(i), states(pr)(i + 1))

NoTimeDecrease(pr) : bool =
   FORALL i : now(states(pr)(i)) <= now(states(pr)(i + 1))

NonZeno(pr): bool =
   FORALL t :   EXISTS i :  t < now(states(pr)(i))

Runs : TYPE =
   { pr: PreRuns | Init(states(pr)(0)) AND PreEffectOK(pr) AND
                   NoTimeDecrease(pr) AND NonZeno(pr) }
```

[1] More details about the PVS theories and proofs can be found at the project web page: http://hvg.ece.concordia.ca/Publications/TECH_REP/PVS_TR03/PVS_TR03.html

3.2 Leaders Actions

To define the actions of the leaders, we first state a few preliminary definitions. Let n be the number of leaders and let f be such that $3f + 1 \leq n$ (the maximum number of faulty leaders). For simplicity, leaders are identified by an element of $\{0, 1, \ldots, n-1\}$. Users are represented by some uninterpreted non-empty type. We model time as a non-negative real number and define three time constants for the maximum delay of messages in the network, the maximum delay between trypropagate actions and the maximum delay between tryaccept actions. Details are shown below:

```
n  : posnat
f : { k : nat | 3 * k + 1 <= n }

LeaderIds : TYPE = below[n]
UserIds   : TYPE+
Time :      TYPE+ = nonneg_real

i       : VAR LeaderIds
user    : VAR UserIds
t       : VAR Time
MaxMessageDelay, MaxTryPropagate, MaxTryAccept  : Time
```

The actions of the protocol are represented in PVS as a data type, which ensures, e.g., that all actions are syntactically different.

```
LeaderActions [LeaderIds, UserIds, Time : TYPE] : DATATYPE
BEGIN
   delay(del : Time) : delay?
   announce(id : LeaderIds, user : UserIds) : announce?
   trypropagate(id : LeaderIds) : trypropagate?
   tryaccept(id : LeaderIds) : tryaccept?
   receive(id : LeaderIds) : receive?
   crash(id : LeaderIds) : crash?
   misbehave(id : LeaderIds) : misbehave?
END LeaderActions
```

Informally, these actions have the following meaning:

- *delay* is a general action which occurs in all our timed models; it increases the current time (`now`), and all other clocks that may be defined in the system, with the amount specified by parameter `del`.
- The *announce* action is used to send announcement messages of new locally authenticated users to the other leaders of the protocol.
- The *trypropagate* action allows a user announcement to be further spread among leaders. This action is executed periodically, but it only changes the state of the system if enough announcements ($f + 1$) have been received for the considered user and it has not already been announced or propagated by the leader in question before.

- *Tryaccept* is used to let leaders periodically check whether they have received enough many announcements and/or propagation messages for a given user. Once this condition is satisfied, the user is accepted as a member of the group.
- The *receive* action allows a leader to receive messages. More concretely, it is used to remove a received message from the network and to add corresponding data to the leaders local buffers.
- The *crash* action models the failure of a leader. After a crash it may still perform all the actions mentioned above, but in addition may perform a *misbehave* action.
- Action *misbehave* models the Byzantine mode of failure and can only be performed by a faulty (crashed) leader.

3.3 States

In order to properly capture the distributed nature of the network, it is suitable to model two kinds of states: a local state for each leader, accessible only to the particular leader, and a global state to represent global system behavior which includes the local state of each leader, the representation of the network and a global notion of time.

An important part of the local state is the group `views`, which is a set of users in the current group. In fact, the ultimate goal of Enclaves is to assure consistency of the group views. Moreover we have a boolean flag `faulty` marking the leader status as to faulty or not, some local timers `clockp` and `clocka` to enforce upper bounds on the occurrence of `trypropagate` and `tryaccept` actions, and finally `received`, a list of the leaders from which the local leader received proposals for a given user.

```
Views : TYPE = setof[UserIds]

LeaderStates : TYPE =
  [# view          : Views,
     faulty        : bool,
     clockp        : Time,   % clock for the trypropagate action
     clocka        : Time,   % clock for the tryaccept action
     received      : [UserIds -> list[LeaderIds]]   #]
```

We model `Messages` as quadruples containing a source, a destination, a proposed user and a timestamp indicating an upper bound on the delivery time, i.e. the message must be received before the `tmout` value.

```
Messages : TYPE = [# src      : LeaderIds,
                     tmout    : Time,
                     proposal : UserIds,
                     dest     : LeaderIds   #]
```

In the `GlobalStates` the network is modeled as a set of messages. Messages that are broadcast by leaders are added to this set, with a particular time-out value, and they are eventually received, possibly with different delays and at a different order at recipient ends. The global state also contains the local state of each leader and a global notion of time, represented by `now`.

```
GlobalStates : TYPE = [# ls      : [LeaderIds -> LeaderStates],
                         now     : Time,
                         network : setof[Messages]  #]
s, s0, s1   : VAR GlobalStates
```

Predicate `Init` expresses conditions on the initial state, requiring that all views, received sets and the network are empty, all clocks and now are zero.

3.4 Precondition and Effect

For each action `A` we define its precondition, expressing when the action is enabled, and its effect. An `announce` action may always occur and hence has precondition `true`. Similarly for `trypropagate` and `tryaccept` which should occur periodically. Action `receive(i)` is only allowed when there exists a message in the network with destination `i`. For simplicity, a `crash` action is only allowed if the leader is not faulty (alternatively, we could take precondition `true`). A `misbehave` action may only occur for faulty leaders.

Most interesting is the precondition of the `delay(t)` action. This action increases `now` and all timers `clockp` and `clocka` by `t`. To ensure that messages are delivered before their time-out value, we require that condition `prenetwork` holds in the state before a `delay(t)` action.

```
prenetwork(s, t) : bool =  FORALL msg :
    member(msg, network(s)) IMPLIES  now(s) + t <= tmout(msg)
```

Note that when the time-out of a message equals the current time, no delay action is possible; first a receive action has to occur. Similarly, there is a condition `preclock` which requires that all timers `clockp` and `clocka` are not larger than `MaxTryPropagate` and `MaxTryAccept`, respectively. Since the `trypropagate` and `tryaccept` actions reset their local timers, `clockp` and `clocka` resp., to zero, this may enforce the occurrence of such an action before a time delay is possible.

```
Pre(A)(s) : bool =
   CASES A OF
     delay(t)        : prenetwork(s,t) AND preclock(s,t),
     announce(i,u)   : true,
     trypropagate(i) : true,
     tryaccept(i)    : true,
     receive(i)      : MessageExists(s,i),
     crash(i)        : NOT faulty(ls(s)(i)),
     misbehave(i)    : faulty(ls(s)(i))
    ENDCASES
```

Next we define the effect of each action, relating a state `s0` immediately before the action and a state `s1` immediately afterwards.

- `delay(t)` increments `now` and all local timers by `t`, as defined by `s0+t`.
- `announce(i,u)` adds, for each leader `j` a message to the network, with source `i`, time-out `now(s0)+MaxMessageDelay`, proposal `u`, and destination `j`. This is expressed by `AnnounceEffect(s0,i,u,s1)`.
- `trypropagate(i)` resets `clockp` to zero and adds to the network messages, to all leaders, containing proposals for each user for which at least `f+1` messages have been received.
- `tryaccept(i)` resets `clocka` to zero and adds to its local view all users for which at least `n-f` messages have been received.
- `receive(i)` removes a message with destination `i` from the network, say with source `j` and proposal `u`, and adds `j` to the list of received leaders for `u` provided it is not in this list already.
- `crash(i)` sets the flag `faulty` of `i` to `true`.
- `misbehave(i)` may just reset the local timers `clockp` and `clocka` of `i` to zero (and hence it need not try to propagate or accept periodically), as expressed by `ResetClock(s0,i,s1)`, or it may add randomly, and above all, maliciously chosen messages to the network (as long as the timing constraints are not violated). A faulty leader, however, cannot impersonate other protocol participants, i.e., cannot send messages on behalf of other participants and any message sent on the network has the identifier of its actual sender.

This leads to a predicate of the following form:

```
Effect(s0,A,s1) : bool =
   CASES A OF
     delay(t)        : s1 = s0 + t,
     announce(i,u)   : AnnounceEffect(s0,i,u,s1),
     trypropagate(i) : PropagateEffect(s0,i,s1),
     tryaccept(i)    : AcceptEffect(s0,i,s1),
     receive(i)      : ReceiveEffect(s0,i,s1),
     crash(i)        : CrashEffect(s0,i,s1),
     misbehave(i)    : ResetClock(s0,i,s1) OR SendMessage(s0,i,s1)
    ENDCASES
```

3.5 Protocol Runs and Fault Assumption

Runs of this timed automata model of Enclaves are obtained by importing the general timed automata theory. This leads to type `Runs`, with typical variable `r`.

```
IMPORTING TimedAutomata[GlobalStates, LeaderActions, Init,
                              Pre, Effect, now]
r           : VAR Runs
```

Let `Faulty(r,i)` be a predicate expressing that leader `i` has a state in which it is faulty. It is easy to check in PVS that once a leader becomes faulty, it remains faulty forever.

Let `FaultyNumber(r)` be the number of faults in run `r` (it can be defined recursively in PVS). Then we postulate, by an axiom with name `MaxFaults`, that the maximum number of faults is `f`.

```
MaxFaults : AXIOM FaultyNumber(r) <= f
```

4 Formal Verification

We verify the following properties of the Intrusion-Tolerant Enclaves protocol:

- **Termination**: if a user u wants to join an active group and is locally authenticated by a non-faulty leader, then user u will be eventually accepted by all non-faulty leaders and become a member of the group.
- **Integrity**: a user that has been accepted in the group should have been locally authenticated and announced by a non-faulty leader earlier during the protocol execution.
- **Proper Agreement**: if a non-faulty leader decides to accept a user u, then all non-faulty leaders accept user u too.

In the remainder of this section, we first present the proof of the termination property, and next give a short outline of the proof for integrity and finally derive proper agreement.

Theorem 1 (Termination)
For all `r` *and* `u`*,* `announced_by_many(r,u)` *implies* `accepted_by_all(r, u)`

where

- `announced_by_many(r,u)` expresses that at least $f + 1$ non-faulty leaders announced user `u` during run `r`;
- `accepted_by_all(r, u)` asserts that eventually all non-faulty leaders have user `u` in their `view` during run `r`.

Proof Assume `announced_by_many(r,u)`, which implies that at least $f + 1$ non-faulty leaders broadcast a proposal for `u`. Because of the reliability of the network, these messages will be eventually delivered to their destination, and in particular to the $n - f$ non-faulty leaders of the network. They all receive $f + 1$ announcement messages for user `u`, enough to have `u` in their `PropSets` and trigger the propagation procedure for all non-faulty leaders who did not participate in the announcement phase. Now because of the network reliability, we conclude that eventually all non-faulty leaders will receive at least $n - f$ approvals for user u, enough to make a majority ($n - f > f$ as $n > 3f$). □

The formal proof of this theorem in PVS required over 25 intermediate lemmas, most of which are of average difficulty. The proof backbone resides on the following steps (which are represented as lemmas in PVS):

- `announced_by_many(r,u)` implies that eventually user `u` has been received by all non-faulty leaders at least $f+1$ times.
  Note that this requires the use of the `NonZeno` property of runs, to obtain a state with global time greater than the largest time-out value in the messages; then, by the precondition of the `delay` action, all messages must have been received.
- If all non-faulty leaders have received proposals for `u` at least $f+1$ times then all non-faulty leaders eventually broadcast a proposal for `u`.
  In this step the `NonZeno` property is used to reach a state where time has advanced more than `MaxTryPropagate` time units. Hence, given the condition on `clockp` for a `delay` action, a `trypropagate` action should have happened before.
- If all non-faulty leaders eventually broadcast a proposal for `u` then all non-faulty leaders eventually receive a proposal for `u` from all non-faulty leaders.
  Again this follows from NonZenoness and the precondition of the `delay` action.
- If all non-faulty leaders eventually receive a proposal for `u` from all non-faulty leaders, then they all have received at least $n-f$ proposals for `u`.
- If all non-faulty leaders eventually have received at least $n-f$ proposals for `u` then they all eventually accept `u`, i.e. include `u` in their `view`.
  Here we need NonZenoness and the condition on `clocka` for a `delay` action.

Details of the proof concern invariance properties about upper bounds of the local timers `clockp` and and `clocka`, and fact about the `received` list (e.g. that leaders are never removed from this list).

Theorem 2 (Integrity)
For all `r` *and* `u`*,* `accepted_by_one(r,u)` *implies* `announced_by_one(r,u)`

where

- `accepted_by_one(r,u)` holds if at least one leader eventually included `u` in its `view` during run `r`.
- `announced_by_one(r,u)` expresses that at least one non-faulty leader announced user `u` during run `r`;

Proof We proceed by contrapositive and use the non-impersonation property. We assume that for all non-faulty leaders no announcement for user `u` has been done during run `r`. Now because of non-impersonation, faulty leaders cannot send more than f different announcements. This implies that the leaders would receive no more than f announcements for user `u`, which is not enough to trigger propagation actions. This yields that `u` will never be in any of the non-faulty leaders `PropSet`, and hence in none of the `AcceptSets`. As a result user `u` will never be accepted by any of the non-faulty leaders. $\square$

Theorem 3 (Proper Agreement)
For all `r` *and* `u`*,* `accepted_by_one(r,u)` *implies* `accepted_by_all(r,u)`

Proof `accepted_by_one(r,u)` implies that there exists one non-faulty leader that received at least $n-f$ approvals (i.e. announcements or propagation messages) for user `u`. Among these approvals, at least $n-2f$ come from non-faulty leaders (by non-impersonation). Now because these leaders are non-faulty, they broadcast the same approval to all the other leaders. In addition, because of the network reliability, these messages are eventually delivered to destination. This implies that all $n-f$ non-faulty leaders receive eventually the above $n-2f$ approvals. Since $n-2f > f+1$, all $n-f$ non-faulty leaders have user `u` in their `PropSet`. Now like in the proof of Termination, the latter implies the start of the propagation procedure, then the reception of at least $n-f$ approvals for user `u`, and finally the acceptance of `u` by all non-faulty leaders. □

The above proofs were conducted in PVS and required over 40 lemmas.

5 Related work

Much work has been done to formally verify security and fault-tolerance in distributed protocols. Some of the methods dealt with the Byzantine failure model while others remained limited to the benign form. Also some of them relied on specialized *security logics* (e.g., BAN [15], while others, remained more general and adopted a number of automata formalisms to prove protocols [11, 12, 16, 18].

Castro and Liskov [11] specified a Byzantine fault-tolerant replication algorithm (similar to ours) using the I/O automata of Tuttle and Lynch [10]. They have manually proved their algorithm's safety, but not its liveness, using invariant assertions and simulation relations. The work of Castro and Liskov is related to ours in the sense that it targets a Byzantine fault-tolerant multi-leader protocol. Although successful in pen-and-paper fashion, their work has never been conducted mechanically in a theorem prover.

Kwiatkowska and Norman [12] analyzed the Asynchronous Binary Byzantine Agreement [19] (based on a similar concept to our key management module) using a combination of mechanical inductive proofs (for non-probabilistic properties) and finite state checks (probabilistic properties) plus one high-level manual proof. Our approach too takes advantage of the easiness and performance of the different earlier mentioned techniques to prove the overall Enclaves protocol.

Lynch *et al.* used also timed automata to model their fault-tolerant protocols PAXOS [16] and Ensemble [21]. They assume a partially synchronous network and support only benign failures. This bears similarities with Enclaves verification in the sense that we also assume some bounds on the timing behavior, but unlike the work in [16, 21] we are dealing in this paper with the more subtle Byzantine form of failure.

In [18, 17], Heitmeyer and Archer presented the formal verification of the TESLA protocol using the Timed Automata Modeling Environment (TAME). TAME is a user interface to PVS. It provides a set of theory templates to specify general I/O automata. Our work can be used to extend the TAME package.

6 Conclusion and future work

Although formal verification techniques have reached a certain level of maturity, making complex and safety critical aspects of systems relatively easy to undertake, reasoning about systems involving Byzantine faults remained always a challenging task. In this paper we present our attempt to the formal specification and verification of the Byzantine agreement protocol used in the intrusion-tolerant Enclaves.

We believe we have achieved a promising success in verifying a complex protocol such as the Byzantine leaders agreement of Enclaves. Thanks to the high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, we have succeeded to formalize the protocol for any instance of size n, in a way that thoroughly captures the different protocol subtleties. We have also proved the protocol to satisfy its requirements of *Termination*, *Integrity* and *Proper Agreement* under the earlier mentioned model and fault assumptions. The specification and proofs required respectively around 1200 lines of code and 40 intermediary lemmas, most of which are of average difficulty.

The current verification can be further extended by widening the Byzantine faults capabilities and by bringing, to the scene, the joint cryptographic layers yet abstracted away. This should make the model more complex, and might require a compositional verification of the different layers.

Acknowledgments

The formal specification and analysis of Enclaves benefited from the fruitful discussions with Adriaan DeGroot from University of Nijmegen.

References

1. Bruno Dutertre, Valentin Crettaz and Victoria Stavridou. Intrusion-Tolerant Enclaves. In: Proc. IEEE International Symposium on Security and Privacy, p. 216-226, Oakland, CA. May, 2002.
2. Catherine Meadows. The NRL Protocol Analyzer: An Overview. Journal of Logic Programming, 26(2):113-131,1996.
3. Peter Ryan and Steve Schneider. The Modelling and Analysis of Security Protocols: the CSP Approach. Addison-Wesley, 2000.
4. Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. Journal of Computer Security, 6:85-128,1998.
5. Giampaolo Bella and Lawrence C. Paulson. Mechanical Proofs about a Non-Repudiation Protocol. In: Richard J. Boulton and Paul B.Jackson (editors), Theorem Proving in Higher Order Logics (LNCS 2152): p. 91-104, 2001.
6. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, Sergio Yovine. Symbolic Model Checking for Real-time Systems. In: Proc. 7th. Symposium of Logics in Computer Science, Santa-Cruz, California, 1992.

7. Leslie Lamport, Robert Shostak and MARSHALL Pease. The Byzantine Generals Problem. In: ACM Transactions on Programming Languages and Systems, 4 (3), p.382-401, July 1982.
8. Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing, p.12-26, August 17-19, 1983, Montreal, Quebec, Canada
9. Rajeev Alur and David L. Dill. A Theory of Timed Automata. In Theoretical Computer Science 126: p.183-235, 1994.
10. Nancy Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.
11. Miguel Castro and Barbara Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.
12. Marta Kwiatkowska and Gethin Norman. Verifying Randomized Byzantine Agreement. D.A. Peled, M.Y. Vardi (Eds.): Formal Techniques for Networked and Distributed Systems (LNCS 2529): p. 194-209, 2002.
13. Patrick Lincoln and John Rushby. A Formally Verified Algorithm for Interactive Consistency under a Hybrid Fault Model. In Fault Tolerant Computing Symposium, p. 304-313, Toulouse, France, June, 1993.
14. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In 11th International Conf. on Automated Deduction, (LNCS 607): p. 748-752, 1992.
15. M. Burrows, M. Abadi and R. Needham. A Logic of Authentication, In: ACM Transactions on Computer Systems, 8(1): p. 18-36, 1990.
16. Roberto De Prisco, Butler W. Lampson, Nancy A.Lynch. Revisiting the PAXOS Algorithm. In Mavronicolas, M. and Tsigas, P., editors, 11th International Workshop on Distributed Algorithms, (LNCS 1320): p. 111-125, 1997.
17. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving Invariants of I/O Automata with TAME. In Automated Software Engineering, Vol.9, p. 201-232, 2002.
18. Myla Archer. Proving Correctness of the Basic TESLA Multicast Stream Authentication Protocol with TAME. In Workshop on Issues in the Theory of Security, Portland, OR, January 14-15, 2002.
19. Christian Cachin, Klaus Kursawe and Victor Shoup. Random oracles in constantipole: practical asynchronous Byzantine agreement using cryptography (extended abstract). In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, p. 123-132, Portland, Oregon, 2000.
20. John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0. Draft paper available at http://www-users.cs.york.ac.uk/~jac
21. Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and Proofs for Ensemble Layers. In 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (LNCS 1579), p. 119-133, March, 1999.