Formal Specification and Verification of the Intrusion-Tolerant Enclaves Protocol

Mohamed Layouni[1], Jozef Hooman[2], and Sofiène Tahar[3]

[1]Computer Science Department, McGill University, Montreal, Canada
[2]Computing Science Department, Radboud University, Nijmegen
& Embedded Systems Institute, Eindhoven, The Netherlands
[3]Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada
(Email: mlayou@cs.mcgill.ca, hooman@cs.ru.nl, tahar@ece.concordia.ca)

Abstract

We demonstrate the application of formal methods to the verification of intrusion-tolerant agreement protocols that have a distributed leadership and can tolerate Byzantine faults. As an interesting case study, the Enclaves group-membership protocol has been verified using two techniques: model checking and theorem proving. We use the model checker Murphi to prove the correctness of authentication, and the interactive theorem prover PVS to formally specify and verify Byzantine agreement, termination of agreement, and integrity.

Keywords: Formal Verification Methods, Theorem Proving, Model Checking, Group Membership Protocols, Byzantine Agreement, Intrusion Tolerance.

1 Introduction

The explosive growth in the amount of electronic information that individuals and organizations generate, and the ever-increasing value of that information, make its protection one of today's top priorities. A number of cryptographic protocols and techniques have been developed over the last couple of decades to protect information transfer and processing. Nevertheless, it is still widely recognized that cryptographic protocols are a tricky issue. Even seemingly simple protocols like authentication and authorization protocols have often turned out, years later, to be wrong (see [7] for a survey).

Clearly, simulation and testing are not sufficient to detect all errors in complex distributed protocols. Potentially, formal verification methods offer a logical basis to prove that *all* possible executions of a protocol satisfy a set of desired properties. In these methods, both protocol and properties are expressed in languages with a precise mathematical meaning. In addition, there are mechanisms to prove in a logically sound way that the protocol satisfies the properties (or discover that this is not the case). In this paper, we consider both model checking and theorem proving. Model checking tools construct the proof (or a counter example) automatically, but there are restrictions on the protocol model (usually it has to be finite and relatively small) and the properties that can be proved. Theorem proving supports more general models and properties, allowing the verification of unbounded infinite systems, but proofs require user interaction and are much more difficult to construct.

A substantial progress in the formal verification of cryptographic protocols has been achieved during the last decade. A wide variety of techniques has been developed to verify a number of security properties such as confidentiality, integrity, authentication, and non-repudiation [24, 18] (more related work is described in Section 2). The focus, however, was either on two-party protocols (i.e., involving only a pair of users) or on group protocols with a *trusted central* leadership (i.e., a trusted fault-free server managing a group of users). In this paper, we address the verification of a more general setting, where group protocols have a *distributed*

leadership, a portion of which could be corrupted in accordance with the Byzantine failure model.

In the Byzantine failure model [16] corrupted servers may maliciously collude, and behave in an arbitrary way. Two important questions that arise in the formal verification of Byzantine fault-tolerant protocols are: how much power should be given to a Byzantine fault? And how general should the model be to capture the arbitrary nature of a Byzantine behavior? These questions have been studied in the literature (e.g., [4, 3, 16]) and continue to be a center of focus. In this paper, we limit Byzantine faults only by cryptographic constraints. That is, we assume encryption primitives semantically secure. Faulty leaders can, for instance, send arbitrarily random messages, reset their local clocks and perform any action without satisfying its precondition. They cannot, however, decrypt a message without having the appropriate key, or impersonate other participants by forging cryptographic signatures.

As an interesting case study, we consider in this paper the verification of the intrusion-tolerant protocol Enclaves [11]. This is a group-membership protocol with a distributed leadership architecture, where the authority of the traditional single server is shared among a set of n servers, of which at most f could fail at the same time. Enclaves assumes the Byzantine failure model mentioned above. More details about our fault assumptions can be found in Section 3. The protocol has a maximum resilience of one third (i.e., $f \leq \lfloor \frac{n-1}{3} \rfloor$) and uses a fault-tolerant broadcast algorithm similar to that in [3].

The primary goal of Enclaves is to preserve an acceptable group-membership service of the overall system despite intrusions at some of its sub-parts. 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 try to coordinate their messages in such a way as to mislead non-faulty leaders (the majority) into disagreement, and thus into rejecting user u. In addition, malicious leaders should be prevented from leaking sensitive information (e.g., group keys) or providing clients with fake group keys.

To achieve its intrusion-tolerant capabilities, Enclaves combines an authentication protocol, a Byzantine fault-tolerant leader agreement protocol, and a verifiable secret sharing scheme. Although the underlying cryptographic primitives and fault-tolerant components are assumed to be perfectly secure on their own, one cannot easily guarantee the security of the whole protocol.

In this work, we discuss a formal verification of the overall Byzantine fault-tolerant Enclaves protocol. We experimented with various techniques, chosen according to the nature of the correctness arguments in each module, the environment assumptions, and the easiness of performing verification. We found it profitable to check the authentication module by taking advantage of the reduction techniques available in the model checker Murphi [9]. The Byzantine leaders agreement module, however, was a little trickier. In fact, the latter relies, to a large extent, on the timing and the coordination of a set of distributed actions, possibly performed by Byzantine faulty processes whose behavior is hard to represent in a model checker. Instead, we used the interactive theorem prover PVS [21] and formalized the protocol in the style of Timed-Automata [1]. 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* number of leaders, and we proved safety and liveness properties such as proper agreement, agreement termination and integrity using the interactive proof checker of PVS.

The remainder of this paper is organized as follows. In Section 2, we review previous work on the formal verification of fault-tolerant distributed protocols. In Section 3, we give an overview of the architecture and design goals of Enclaves, and explicitly state our system model assumptions. In Section 4, we describe the model checking of the authentication module in Murphi. In Section 5, we present how we model the elementary components of the Byzantine leaders agreement module in PVS and how we build the final protocol model out of these ingredients. In Section 6, we formulate and prove the correctness theorems for the Byzantine leaders' agreement. Finally, in Section 7, we conclude the paper by commenting

on our results and stating some perspectives for future work.

2 Related Work

Much work has been done to formally verify fault-tolerance in distributed protocols. Some of these verifications dealt with the Byzantine failure model [4], while others remained limited to the benign form [20]. A variety of automata formalisms has been adopted to specify such protocols.

For instance, Castro and Liskov [4] specified their Byzantine fault-tolerant replication algorithm using the I/O automata of Tuttle and Lynch [19]. They have manually proved their algorithm's safety, but not its liveness. The work in [4] has never been mechanized in any theorem prover. In our work, we prove both safety (e.g., proper agreement) and liveness (e.g., termination) properties, as well as mechanize all proofs with PVS.

Timed automata were also used to model the fault-tolerant protocols PAXOS [23] and Ensemble [13]. The authors assume a partially synchronous network and support only benign failures. This bears some similarities with our verification in the sense that we assume some bounds on timing, but unlike the work in [23, 13] we are dealing with the more general Byzantine failure model.

In [2], Archer et al presented the formal verification of a number of distributed protocols using the Timed Automata Modeling Environment (TAME). TAME provides a set of theory templates to specify and prove I/O automata. As of the time of writing this paper, the TAME environment does not support security protocols. The results we have achieved in this work could be used to extend the TAME environment to model and analyze protocols such as Enclaves.

In [18], Paulson et al extend their inductive approach to cope with the so-called *second-level* security protocols. Our work uses induction as well (among other techniques), but is not constrained to second-level security protocols only.

3 The Enclaves Protocol

Enclaves [11] 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 other group members.

The group-management service consists of 3 sub-blocks: 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 $[t_1, t_2]$. At time t_3 the agreement procedure starts, and a consensus as to whether accept user u or not is reached at time t_4. Finally, on acceptance, user u is provided with the current group composition, as well as information to construct 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.

In summary, Enclaves should guarantee the following properties, even in the presence of up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ corrupted leaders, where n is the total number of leaders:

- *Proper authentication and access control*: Only authorized users can join the group and an authorized user cannot be prevented from joining the group.

- *Confidentiality of group communication*: Messages from a member u can be read only by the users who were in u's view of the group at the time the message was sent.

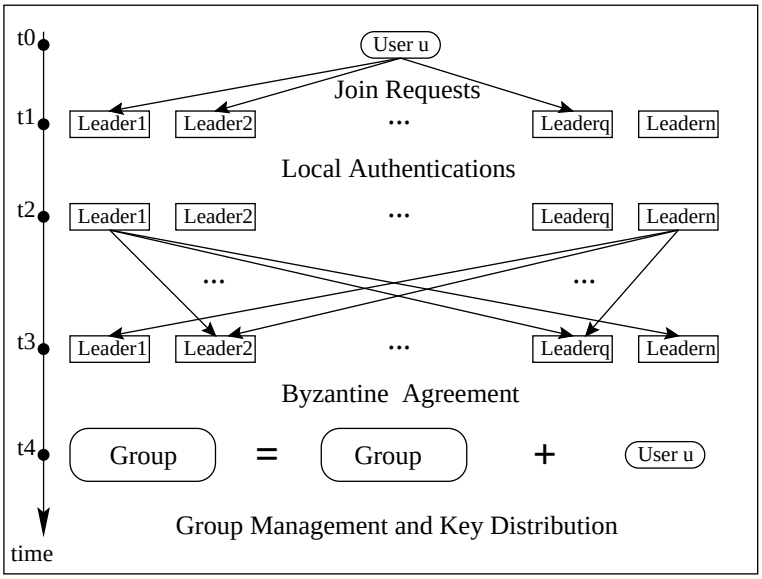

Figure 1: Enclaves protocol execution

The description of Enclaves in [11] 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 assumptions, we prove that the *Proper authentication and access control* requirement holds through (1) the model checking of the Proper Authentication invariant in Murphi (cf. Section 4), and (2) the proofs of proper agreement, agreement termination and agreement integrity theorems in PVS (cf. Sections 5 and 6)[1]. The *confidentiality of group communications* is addressed in

[1]More details about the Murphi specifications, as well as the PVS theories and proofs, can be found at http://hvg.ece.concordia.ca/Publications/TECH_REP/PVS_TR03/PVS_TR03.html

the group key management module by means of a verifiable secret sharing scheme. This module is, however, outside the scope of this paper. Further details on the security of this module can be found in [17].

4 Model Checking Authentication in Murphi

Murphi [9] is a verification tool that has been applied to several protocols, notably in the areas of distributed memory systems, and authentication protocols [20, 22]. To use Murphi, one has to first model the protocol using the Murphi language. Later this model is augmented with a specification of the desired properties. Typically one would start with a small protocol instance and gradually increase the protocol size until the verification does not terminate anymore. In many cases, errors in the general protocol will also show up in smaller down-scaled instances of the protocol. The Murphi tool is based on explicit state enumeration and supports a number of reduction techniques such as symmetry and data independency [14, 15]. The desired properties of a protocol can be specified in Murphi by invariants. If a state is reached where some invariant is violated, Murphi generates an error trace exhibiting the problem.

Our verification has been conducted as follows. First, we formulated the protocol by identifying the protocol participants, the state variable and messages, and the key actions to be taken. Then we added an intruder to the system. In our model, the intruder is a participant in the protocol, capable of eavesdropping messages in transit, decrypting cipher-text when it has the appropriate keys, and generating new messages using any combination of previously gained knowledge. Finally, we stated the desired correctness conditions and ran the verification for few size parameters.

4.1 System Model

The local authentication module (as shown in Figure 1) aims at mutual authentication between group leaders and users trying to join an active group. Group

leaders need to be assured about the users identity in order to convince the rest of the leaders to accept them in the group, and the users, from their side, want to have a guarantee that they are not being fooled by some impostor.

This module is designed to work in a malicious environment, where messages can be overheard, replayed and created by a standard Dolev-Yao intruder [10]. The protocol assumes "perfect" cryptography, namely, if a message m is encrypted with some participant's public key K, then only this participant is able to decrypt the ciphertext $\{m\}_K$.

It has been proved in [5] that the Dolev-Yao intruder is the most powerful of all possible attackers (when perfect cryptography is considered). Furthermore, it has been shown in [6] that considering a single Dolev-Yao intruder is no more restrictive than considering multiple ones. This is a very important result for model checking-based security, because reducing the threat model to a single intruder (instead of n) dramatically reduces the search space. In this work, we follow these observations and consider one single intruder.

We study the following version of the local authentication module:

$$
\begin{array}{lll}
i. & U \longrightarrow L_i: & \mathsf{AuthInitReq}, U, L_i, \{U, L_i, N_1\}_{P_{U,i}} \\
ii. & L_i \longrightarrow U: & \mathsf{AuthKeyDist}, L_i, U, \{L_i, U, N_1, N_2, K_{U,i}\}_{P_{U,i}} \\
iii. & U \longrightarrow L_i: & \mathsf{AuthAckKey}, U, L_i, \{U, L_i, N_2, N_3\}_{K_{U,i}}
\end{array}
$$

The user U sends a freshly generated random nonce N_1 along with its identifier to Leader L_i, both encrypted with the long term key $P_{U,i}$ shared by L_i and U. Leader L_i decrypts the message and obtains knowledge of N_1. It checks U's identity in a predefined database, and then generates a nonce N_2 and a session key $K_{U,i}$ and sends the whole encrypted with the shared key $P_{U,i}$. User U decrypts the message and concludes that it is indeed talking to L_i, since only L_i was able to decrypt U's initial message containing nonce N_1 (L_i is hence authenticated). Similarly U is authenticated, in the third step of the protocol, after sending an acknowledgment including N_2 and using $K_{U,i}$.

4.1.1 Modeling Users and Leaders

First, we consider the users component, referred to as $clients$ in our model. In Murphi the data structure for the $clients$ is as follows:

```
const
  NumClients: 3;            -- A small example
type
  ClientId: scalarset (NumClients);
  ClientStates : enum {
    C_SLEEP,                -- Initial state
    C_WAIT,                 -- Waiting for response from leader
    C_ACK                   -- Acknowledging the session key
                      };
  Client : record
    state:  ClientStates;
    leader: AgentId;        -- Leader with whom the client
  end;                      -- starts the protocol
var
  clnt: array[ClientId] of Client;
```

The number of clients is scalable and is defined by the constant $NumClients$. The type $ClientId$ is a $scalarset$ of size $NumClients$; that is, a Murphi data structure used to denote intervals $[1 \ldots NumClients]$, and to enable reductions on instances of that type. The state of each client is stored in the array $clnt$. In the initialization statement of the model, the local state (stored in field $state$) of each client is set to C_SLEEP, indicating that no client has started the protocol yet.

The behavior of a client is modeled with two Murphi rules. The first rule is used to start the protocol by sending the initial message to some agent (supposedly a leader), and then change the sender's local state from C_SLEEP to C_WAIT. The second rule models the reception and checking of the reply from an agent, the commitment, and the sending of the final message. The Murphi model for the first rule is as follows:

```
ruleset i: ClientId do
  ruleset j: AgentId do
    rule "client starts protocol (step 1)"
      clnt[i].state = C_SLEEP &   -- play protocol only with
      !ismember(j,ClientId) &     -- leaders and intruders
      multisetcount (l:net, true) < NetworkSize
```

```
    ==>
    var
      outM: Message;                -- outgoing message
    begin
      undefine outM;
      outM.psource  := i;
      outM.pdest    := j;
      outM.mType    := M_AuthInitReq;
      ...
      multisetadd (outM,net);
      clnt[i].state  := C_WAIT;
      clnt[i].leader := j;
    end;
  end;
end;
```

The condition of the rule is that client i is in the local state C_SLEEP, that agent j is not trivially a client (and hence should be either a leader or an intruder), and that there is space in the network for an additional message. The network is modeled by the shared variable net. Once the rule is enabled, the outgoing message is constructed and added to the network. In addition, the local state is updated and the identifier of the intended destination is stored in state variable $clnt[i].leader$.

The leader part of the model is quite similar to the client part. For instance, the leaders also maintain a local state and store the identifier of the agent initiating the protocol in their state variable $lead[i].client$. In addition, the behavior of the leaders is also modeled with two rules: one that handles the initial authentication request of the client and another which commits to the session after receipt of the final message of the protocol.

4.1.2 Modeling Intruders

The intruder maintains a set of overheard messages and an array representing all the nonces it knows. The behavior of the intruder is modeled with three rules: one for eavesdropping and intercepting messages, one for replaying messages, and one for generating messages using the learned nonces and injecting them into the network. The model for the first rule is given in the following.

```
ruleset i: IntruderId do
  choose j: net do
     rule "intruder overhearing messages"
     !ismember (net[j].psource, IntruderId) -- not for intruder
     ==>
     var   temp: Message;

     begin
       alias msg: net[j] do   -- message to intercept
       alias intruderknowledge: int[i].messages do
       if multisetcount(f:intruderknowledge, true)
                                < MaxKnowledge then
         if msg.key=i then       -- msg encrypted with i's key
           int[i].nonces[msg.nonce1] := true;  -- learn nonces
           if msg.mType= M_AuthKeyDist  then
             int[i].nonces[msg.nonce2] := true;
           end;
         else        .....                  -- learn whole msg
         end;
    ........
end;
```

The enabling condition of the *intruder' message overhearing* rule is that the network cell in question, $net[j]$, does not contain a message sent by the intruder itself (otherwise nothing will be learned). We distinguish then two cases:

- The intercepted message is intended for the intruder (encrypted with a key known to the intruder $msg.key = i$), then the action is simply to learn the nonces (cf. Murphi model above).
- The intruder intercepts a message that is intended for another participating agent and then learns all useful message fields. The intruder can also be modeled to block and remove messages from the network.

4.2 Properties Specification

The main property we are interested in is *mutual authentication* between a given pair of leader and client, namely L_i should be able to assert that it has been talking, indeed, to client U, and vice-versa. The verification is done by means of invariant

checking under the above assumptions. The *client proper authentication* invariant is given below.

```
invariant "client proper authentication"
  forall i: LeaderId do
    lead[i].state = L_COMMIT &
    ismember(lead[i].client, ClientId)
    ->
    clnt[lead[i].client].leader = i &
    clnt[lead[i].client].state = C_ACK
end;
```

It basically states that for each leader i, if it committed to a session with a client, then this client (whose identifier is stored in $lead[i].client$), must have started the protocol with leader i, i.e., have stored i in its field $leader$ and be awaiting for acknowledgment (i.e., in state C_ACK).

In addition to the above invariant, we have checked a similar one for *leaders proper authentication*. The *leaders proper authentication* invariant asserts that for each client, if it commits to a session with a leader L_i, then L_i is, in reality, the same leader with whom the client started the session.

```
invariant "leaders proper authentication"
 forall i: ClientId do
   clnt[i].state = C_ACK &
   ismember(clnt[i].leader, LeaderId)
   ->
   lead[clnt[i].leader].client = i &
   ( lead[clnt[i].leader].state = L_WAIT |
     lead[clnt[i].leader].state = L_COMMIT )
 end;
```

4.3 Experimental Results

Table 1 summarizes the experimental results obtained from the model checking of the first invariant, *clients proper authentication*, including the number of reached states and CPU run times taken on a six-440-MHz-processor Sun Enterprise Server with 6 GB of memory, for different instance sizes of the protocol. The dashes ("–") in the table indicate that no conclusive results were obtained for

# Clients	# Leaders	Network size	States	CPU time
2	2	1	274753	515 s
3	2	1	–	–
2	3	1	1240550	3408 s
2	4	1	3723157	18383 s
2	5	1	–	–
3	1	1	1858746	3161 s
3	2	1	–	–
2	2	2	–	–
3	1	2	–	–

Table 1: Model Checking Experimental Results

those instances because of a memory overflow. The instances of the protocol that we have considered, were chosen in a way that emphasizes the weight of each size parameter. Our approach is as follows. We start with an instance of the protocol for which the model checking terminates (e.g., the first row in the table), and from there we explore several instances, following a certain pattern, where we vary only one size parameter and keep all others unchanged. The results roughly show that the number of leaders is less significant, in terms of verification complexity, then other parameters such as the number of clients, and the network size (maximum number of messages allowed on the network at the same time). This can be explained by the fact that the average load for each individual leader is reduced when we increase their total number. Another important parameter is the intruder's maximum knowledge (or memory size). For the purpose of this experiment, we have tried few small instances.

Many of the rows in Table 1, show non-conclusive results, where Murphi ends up running out of memory before reaching all possible states. This is a well known problem of model checking in general. One way to improve this, is by deploying more computational resources. However, doing so will not bring a major change,

as the number of states grows exponentially with respect to the size parameters. A better alternative would be to use more powerful abstractions and reduction techniques (cf. [8]) than those currently available in Murphi.

4.4 Discussion

The focus in this section has been on the *mutual authentication* between a single pair of client and leader. When a client concurrently runs several authentication requests with several leaders, we consider those requests inter-independent.

The experimental results show that the efficiency of model checking is still a major problem. Only for a few number of small instances of the protocol, the Murphi tool terminated the model checking in a reasonable amount of time. Although we performed our experiments on a relatively powerful machine, and despite the fact that Murphi had mechanisms to reduce the state space, the execution time increased dramatically as we started increasing the protocol size, and the model checker was unable to terminate. This, indeed, shows the limitations of model checking when applied to security protocols. One way to circumvent this, is by using rank functions [24] in the context of a theorem prover.

5 Modeling Byzantine Agreement 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 (new members join). Because Enclaves depends also on time (participants timeout, timestamp group views, etc.), it was convenient to model it as a timed automaton. In the current verification, timing is used only to ensure actions progress. Timing, however, is essential to prove upper bounds on agreement delays (e.g., a maximum join delay), but this is beyond the scope of this paper. 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.

In this section, we first present the timed automata model of Enclaves in terms of the higher-order typed logic of the PVS specification and verification system. We explain the different components and parameters of the model, then we describe the resulting overall protocol as well as the adopted fault assumptions.

5.1 Timed Automata

We present a general, protocol-independent, theory called $TimedAutomata$. Given a number of parameters, it defines all possible executions of the protocol as a set of $Runs$. A run is a sequence of the form $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$ 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 (an instance of the timed automaton) is characterized by sets of possible $States$ and $Actions$, a condition $Init$ on the initial state, the precondition Pre of each action, expressing in which states that action 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.

```
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 [12]). 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) }
```

5.2 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, and time is modeled by the set of non-negative real numbers.

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

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

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

- A general $delay$ 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 a delay parameter del.
- An $announce$ action is used to send announcement messages of new locally

authenticated users to the other leaders of the protocol.

- A $trypropagate$ 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.

- An action $tryaccept$ used to let leaders periodically check whether they have received enough announcements and/or propagation messages for a given user. Once this condition is satisfied, the user is accepted to join the group.

- A $receive$ action allows a leader to receive messages; it removes a received message from the network and adds corresponding data to the local buffer of the leader.

- A $crash$ action models the failure of a leader. After a crash, a leader may still perform all the actions mentioned above, but in addition it may perform a $misbehave$ action.

- An action $misbehave$ models the Byzantine mode of failure and can only be performed by a faulty (crashed) leader.

Besides, we 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.

5.3 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 $view$, 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 use a Boolean flag ($faulty$) marking the leader status as faulty or not, some local timers ($clockp$ and $clocka$) to enforce upper bounds on the occurrence of $trypropagate$ and $tryaccept$ actions, and finally a list ($received$) 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 $global\ states$, 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
```

Furthermore, we define a predicate $Init$, which expresses conditions on the initial

state, requiring that all views, received sets and the network are empty, and all clocks and now are set to zero.

5.4 Precondition and Effect

For each action A, we define its precondition, expressing when the action is enabled, and its effect.

```
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
```

An $announce$ action, for instance, 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 the condition $prenetwork$, defined below, holds in the state before any $delay(t)$ action is taken, which fits our informal assumptions about network reliability.

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

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 to zero, this may enforce the occurrence of such an action before a time delay is possible.

Next we define the effect of each action, relating a state s_0 immediately before the action and a state s_1 immediately afterwards.

- $delay(t)$ increments now and all local timers by t, as defined by $s_0 + t$.
- $announce(i, u)$ adds, for each leader j a message to the network, with source i, time-out $now(s_0) + MaxMessageDelay$, proposal u, and destination j.
- $trypropagate(i)$ 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, as expressed by $ResetClock(s_0, i, s_1)$, or it may add randomly as well as maliciously chosen messages to the network (provided that timeouts are not violated). A misbehaving leader, however, cannot impersonate other protocol participants, i.e., any message sent on the network has the identifier of its actual sender.

This leads to a predicate of the form:

```
Effect(s0,A,s1) : bool =
   CASES A OF
     delay(t)      : s1 = s0 + t,
     announce(i,u) : AnnounceEffect(s0,i,u,s1),
     ...
     misbehave(i)  : ResetClock(s0,i,s1) OR SendMessage(s0,i,s1)
   ENDCASES
```

5.5 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. 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 faulty leaders in run r (it can be defined recursively in PVS). Then we postulate by an axiom that the maximum number of faults is f (`MaxFaults : AXIOM FaultyNumber(r) <= f`).

6 Proving Byzantine Agreement in PVS

We are interested in verifying the following properties of the Enclaves protocol:

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

In the remainder of this section, we formally enunciate the above theorems and briefly outline their proofs.

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, eventually these messages will be 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, which is enough to trigger the propagation procedure (for u) 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, since $(n - f) > f$ follows from $n > 3f$. □

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 proposed by any of the non-faulty leaders, and hence none of them will receive as much as $(n - f)$ messages for u (recall $(n - f) > f$). As a result, user u will never be accepted by any of the non-faulty leaders. □

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 a 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 $(n - 2f) \geq (f + 1)$, all $(n - f)$ non-faulty leaders have received at least $(f + 1)$ messages for u. Similar to 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. □

Concluding Remarks

In this section, we have verified the correctness of the Byzantine Agreement module of Enclaves using the PVS theorem prover. The high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, were very useful in formalizing the module for any number of leaders, in a way

that thoroughly captures the many subtleties on which the correctness arguments of the module rely. In fact, mechanizing the proofs with PVS, allowed us to discover many errors in our initial pen-and-paper manual proofs, and to correct them.

In addition, the PVS theorem prover provides a collection of powerful inference procedures to help derive theorems. These procedures can be combined to yield higher-level proof strategies making verification much easier. PVS also produces scripts that can be edited, attached to additional formulas, and rerun. Such capabilities have been extremely helpful in this work; they allowed similar theorems to be proved efficiently, permitted a number of proofs to be easily adjusted after modifications were brought to the specification, and helped produce readable proofs.

Using these features, we have proved the module to satisfy its requirements of termination, integrity and proper agreement. The proofs required over 40 intermediate lemmas. The integrity and termination theorems were the most challenging to prove and they helped deduce proper agreement.

7 Conclusion and Future Work

This paper describes our results about the formal verification of an Intrusion-Tolerant group-membership protocol. We experimented with various techniques, namely model checking with Murphi, and theorem proving with PVS. Our choice of the techniques was, adaptively, driven by the nature of the correctness arguments in each module of the protocol, by the environment assumptions, and the easiness of performing verification.

Although we believe we have achieved a promising success in verifying a complex protocol such as Enclaves, we think our results could be further improved. For instance, the feasibility of model checking is always limited to instances with a finite number of states, which may, in some cases, prevent from discovering security flaws in realistic implementations of security protocols. This can be circumvented by the use of rank functions [24]. The role of a rank functions

will be to partition the message space into messages, of positive rank, which the adversary may intercept or infer, and messages, of non-positive rank, that should remain out of the adversary's reach. The verification consists then in finding if, during the protocol execution, some secret information with a non-positive rank can be leaked to the intruder. We believe that using rank functions is a very efficient way to mechanically prove authentication properties (cf. [24]).

The high degree of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, helped us formalize the Byzantine agreement module for any number of leaders, in a way that thoroughly captures the many subtleties on which the correctness arguments of Enclaves rely. We have proved the protocol to satisfy its requirements of termination, integrity and proper agreement. Yet, we have not proved the consistency of group membership when members leave the group. We are planning to address this issue in future work. Finally, one promising direction for further development would be to perform the mathematical analysis of the group key management module mechanically in PVS. This requires the elaboration of some general purpose theories (e.g., number theory, and probabilities) not yet available in PVS. The current specification can be further extended by widening the Byzantine faults capabilities and by modeling the cryptographic primitives that have been abstracted away. Also results about an upper bound on agreement establishment delays can be further investigated.

References

[1] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] M. Archer, C. Heitmeyer, and E. Riccobene. Proving Invariants of I/O Automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[3] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[4] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Report MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.

[5] I. Cervesato. Data access specification and the most powerful symbolic attacker in MSR. In *Software Security - Theories and Systems, LNCS 2609*, pages 384–416, 2002.

[6] I. Cervesato, C. Meadows, and P. Syverson. Dolev-Yao is no better than Machiavelli. In *First Workshop on Issues in the Theory of Security*, pages 87–92, 2000.

[7] J. Clark and J. Jacob. A Survey of Authentication Protocols Literature: Version 1.0. Department of Computer Science, University of York, UK, 1997.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[9] D. Dill, A. Drexler, A. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 11–14, Cambridge, Maryland, USA, 1992.

[10] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(12):198–208, 1983.

[11] B. Dutertre, V. Crettaz, and V. Stavridou. Intrusion-Tolerant Enclaves. In *Proceedings of the IEEE International Symposium on Security and Privacy*, pages 216–224, Oakland, California, USA, 2002.

[12] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. In *Proceedings of the Seventh Symposium on Logics in Computer Science*, Santa-Cruz, California, 1992.

[13] J. Hickey, N. Lynch, and R. van Renesse. Specifications and Proofs for Ensemble Layers. In *Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1579*, pages 119–133, 1999.

[14] C. Ip and D. Dill. Better Verification through Symmetry. In *Proceedings of the International Conference on Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993.

[15] C. Ip and D. Dill. Verifying Systems with Replicated Components in Murphi. In *Computer-Aided Verification, LNCS 1102*, pages 147–158, 1996.

[16] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[17] M. Layouni, J. Hooman, and S. Tahar. Formal Specification and Verification of the Intrusion-Tolerant Enclaves Protocol. Technical report, Concordia University, Department of Electrical and Computer Engineering, 2003.

[18] C. Longo, G. Bella, and L. Paulson. Verifying second-level security protocols. In *Theorem Proving in Higher Order Logics, LNCS 2758*, pages 352–366, 2003.

[19] N. Lynch and M. Tuttle. An introduction to input/output automata. *Centrum voor Wiskunde en Informatica Quarterly Journal*, 2(3):219–246, 1989.

[20] J. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 141–153, Oakland, California, USA, 1997.

[21] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Automated Deduction, LNCS 607*, pages 748–752, 1992.

[22] S. Park and D. Dill. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, Santa Barbara, California, USA, 1995.

[23] R. Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS Algorithm. In *Distributed Algorithms, LNCS 1320*, pages 111–125, 1997.

[24] P. Ryan and S. Schneider. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2000.