Formal Specification and Verification of the Intrusion-Tolerant Enclaves Protocol

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]University of Nijmegen, Nijmegen, The Netherlands
Embedded Systems Institute, Eindhoven, The Netherlands
Jozef.Hooman@embeddedsystems.nl

Technical Report

November 2003

Abstract

In this paper, we present a correctness proof of the Intrusion-tolerant Enclaves protocol [10]. Enclaves is a group-membership protocol. It assumes a Byzantine failure model, and has a maximum resiliency of one third. To carry out the proof, we adaptively combine a number of techniques, namely model checking, theorem proving and analytical mathematics. We use the Murphi model checker to verify authentication, then the PVS theorem prover to formally specify and prove proper Byzantine Agreement, Agreement Termination and Integrity, and finally we mathematically prove the robustness and unpredictability of the group key management module using the random oracle model.

Keywords : Intrusion-Tolerance, Group-Membership Protocols, Model Checking, Theorem Proving, and Provable Security.

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 with assurance 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. Therefore, it is indispensable to formally prove their security.

A substantial progress in the formal verification of cryptographic protocols has been achieved during the last decade. A wide variety of techniques have been developed to verify a number of key security properties ranging from *confidentiality* and *authentication* to *atomic transactions* and *non-repudiation* [27, 25]. 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 [10]: 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, of which at most f 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 an algorithm similar to the consistent broadcast of Bracha and Toueg [4].

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 coordinate their messages in such a way as to mislead non-faulty leaders (the majority) into disagreement, and thus into rejecting user u. Moreover, in order to prevent malicious leaders from leaking sensitive information (e.g., group keys) or providing clients with fake group keys, Enclaves uses a verifiably secure secret sharing scheme.

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 [7] 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.

An important issue that arises in formal verification of Byzantine fault-tolerant protocols, is the modeling of Byzantine behavior. 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 fault behavior? These questions have been extensively studied [6, 18, 19] and continue to be a center of focus. In this paper, faults are only limited by cryptographic constraints. For instance, faulty leaders can arbitrarily send 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. More details about our fault assumptions are discussed in Section 2.

In this work, we discuss a formal analysis of the overall Byzantine fault-tolerant Enclaves protocol. We experiment with an adaptive combination of techniques, chosen according to the nature of the correctness arguments in each module, the environment assumptions, and the easiness of performing verification. For instance, we found it more profitable to model-check the authentication module by taking advantage of the reduction techniques available in 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 use the interactive theorem prover PVS [23] and formalize 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. Finally, the group-key management module is based on a secret sharing scheme whose security relies fundamentally on the hardness of computing discrete logarithms in groups of large prime order. Due to the hardness of expressing the latter correctness arguments in a formal language, we found it more convenient to give a manual proof of the module's *robustness* and *unpredictability* properties, using the Random Oracle model [5].

The remainder of this paper is organized as follows. In Section 2, we give an overview of the architecture and design goals of Enclaves, and explicitly state our system model assumptions. In Section 3, we describe the model checking of the authentication module in Murphi. In Section 4, 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 5, we formulate and prove the Byzantine leader agreement correctness theorems. In Section 6, we briefly give the mathematical proof of robustness and unpredictability of the group key management module. In Section 7, we discuss some related work. Finally in Section 8, we conclude the paper by commenting on our results and stating some perspectives for future work.

2 The Enclaves Protocol

Enclaves [10] 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 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 information to reconstruct 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.

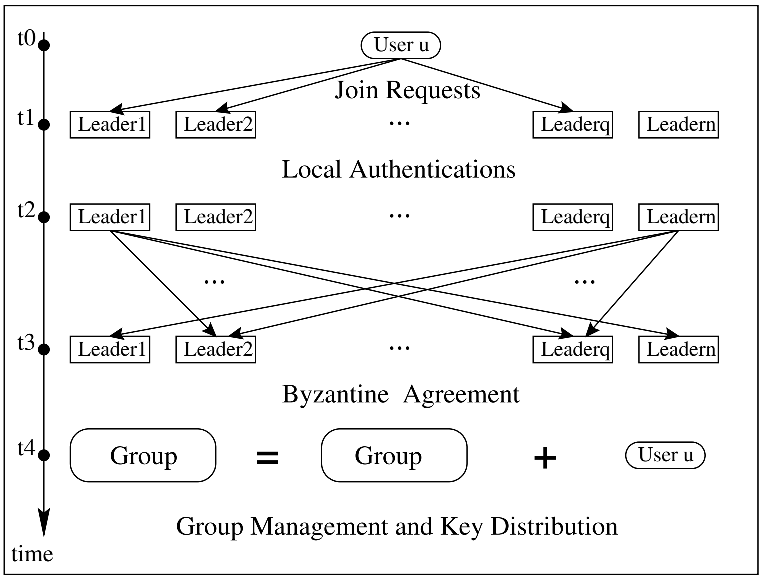

Figure 1: Enclaves protocol execution

In summary, Enclaves should guarantee the following properties, even in the presence of up to f corrupted leaders ($f \leq \lfloor \frac{n-1}{3} \rfloor$, where n is the 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 image of the group at the time the message was sent.

The description of Enclaves in [10] 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. For the leaders agreement module, in particular, 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.

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 3), and (2) the proofs of Proper Agreement, Agreement Termination and Agreement Integrity theorems in PVS (cf. Sections 4 and 5)[1]. In addition, we prove the *Confidentiality*

[1]For more details about the Murphi model and specifications, as well as the PVS theories and proofs, we refer the reader to the web page: https://hvg.ece.concordia.ca/Publications/TECH REP/PVS TR03/PVS TR03.html

of group communication requirement via a mathematical analysis of the Robustness and Unpredictability properties of the group key management module of Enclaves (cf. Section 6).

3 Model Checking Authentication in Murphi

Murphi [9] is a verification tool that has been successfully applied to several industrial protocols, especially in the area of multiprocessor cache coherence protocols, multiprocessor memory models and also authentication protocols [14, 9, 16, 22, 24]. To use Murphi for verification, one has to model the protocol in question in the Murphi language and then augment the model with a specification of the desired properties. Typically one would start with a small protocol configuration and gradually increase the protocol size until the verification does not terminate anymore. In many cases, errors in the general protocol (possibly infinite state) will also show up in down-scaled (finite state) versions 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 [15, 17]. 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 protocol for some specific size parameters.

3.1 System Model

The local authentication module (as shown in Figure 1) aims at mutual authentication between users looking to join an active or a newly created group and the group leaders. Group leaders need to be assured about the users identity in order to convince the rest of the leaders of accepting them in the group, and the users in turn want to have a guarantee that they are not being fooled by some imposter (e.g., a “man in the middle” that pretends a false identity, or title, for the purpose of deception).

This module is designed to work in a malicious environment, in which messages can be overheard, replayed and created by an intruder. The protocol is, however, based on the “perfect” cryptography assumption, i.e., when a message m is encrypted with some participant’s public key K, then only this participant will be able to decrypt the encrypted message $\{m\}_K$.

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 nonce N_1 (i.e., a newly generated random number) 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 $K_{U,i}$.

3.1.1 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;              -- For 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 starts the
  end;                        -- 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 $NumClients$, i.e., a Murphi construct used to denote a subrange like $1 \ldots NumClients$, and to enable automatic symmetry reduction 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 changes its 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 &
      !ismember(j,ClientId) &         -- only 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 second rule of the client is modeled in Murphi as follows:

```
ruleset i: ClientId do
  choose j: net do
    rule "client reacts to nonce received (steps 2/3)"
      clnt[i].state = C_WAIT &
      net[j].pdest = i &
      ismember(net[j].psource,IntruderId)
    ==>
    var
      outM: Message;   -- outgoing message
      inM:  Message;   -- incoming message
    begin
      inM := net[j];
      multisetremove (j,net);

      if inM.key=i then           -- message is encrypted with i's key
        if inM.mType= M_AuthKeyDist then   -- correct message type
          if (inM.nonce1=i &               -- correct nonce and source
             inM.csource=clnt[i].leader) then
            undefine outM;
            outM.psource  := i;
            outM.csource  := i;
            outM.pdest    := clnt[i].leader;
            outM.cdest    := clnt[i].leader;
            outM.key     := inM.sessionKey;
            outM.mType   := M_AuthAckKey;
            outM.nonce1  := inM.nonce2;

            multisetadd (outM,net);
            clnt[i].state := C_ACK;
          else
```

```
        --error "Client received incorrect nonce"
      end;
  ......
end;
```

The condition of the rule is that client i is in the local state C_WAIT and that the destination of the message in network cell $net[j]$ is actually this client. Once enabled, the rule recovers the message and frees the network. The client then checks if it can decrypt the message, if the message type is correct, and if the message contains the correct nonce. If all three conditions hold, an outgoing acknowledgment message $M\_AuthAckKey$ is constructed and added to the network. The client then changes its local state to C_ACK.

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.

3.1.2 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   -- message is 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 message
           alias messages: int[i].messages do
             temp := msg;
             undefine temp.psource;   -- delete useless information
             undefine temp.pdest;
             if multisetcount (l:messages,   -- add only if not there
```

```
                    messages[l].mType = temp.mType &
                     .....
                    (messages[l].mType = M_AuthKeyDist ->
                    messages[l].sessionKey = temp.sessionKey) ) = 0 then
                multisetadd (temp, int[i].messages);
              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 (c.f. Murphi model above).
- The intruder intercepts a message that is intended to another participating agent and then learns all useful message fields. The intruder can also be modeled to block and remove messages from the network.

3.2 Properties Specification

The main property we are interested in is *mutual authentication* between a given pair of leader and client, 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;
```

3.3 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 sizes of the protocol. 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 complexity, then other parameters such as the number of clients, intruders or 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 parameter, of most importance, is the intruder's maximum knowledge (or memory size). For the purpose of this experiment, we have kept it equal to "3" messages (enough for the three steps required in a single session of the protocol).

Besides, 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, but doing so will not bring a major change to the picture, as the number of states grows exponentially with respect to the size parameters. Another efficient way is to adopt powerful model abstraction and reduction techniques [8], which in our case would have to be done manually as Murphi does not support such algorithms.

3.4 Discussion

The focus of this verification work, is on the *mutual authentication* property between any given pair of client and leader. In the case of a client that concurrently runs several authentication requests with different leaders, we consider each session (with a given leader) as independent. In other words, we assume that the correctness of the concurrently running sessions results from the correctness of each session running independently.

In Murphi, the specification can be expressed very intuitively; we simply list the rules of each action the participants can perform in the protocol. Unlike belief-based logics, we do not need to interpret the beliefs that each message would convey to protocol agents. Murphi also offers a lot of freedom, compared to other tools (e.g., SMV [18]), especially when it comes to the definition of the intruder model, often at the center of all security analysis.

On the other hand, efficiency is the main problem. In fact, 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 was equipped with a few techniques to help reduce the state space, the execution

Table 1: Model Checking Experimental Results

# Clients	# Leaders	# Intruders	Network size	States	CPU time
2	2	1	1	274753	515 s
3	2	1	1	–	–
2	3	1	1	1240550	3408 s
2	4	1	1	3723157	18383 s
2	5	1	1	–	–
3	2	1	1	–	–
3	1	1	1	1858746	3161 s
2	2	2	1	–	–
2	2	1	2	–	–
3	1	1	2	–	–
3	1	2	1	–	–
4	6	1	1	–	–
⋮	⋮	⋮	⋮	⋮	⋮
4	2	3	6	–	–

time increased dramatically as we started increasing the protocol size, and the model checker was unable to terminate. This shows the serious limitations of model checking when dealing with security protocols, generally exponential in the number of participants, the network size and the maximum knowledge participants are allowed to remember. One possible way to improve this, is by using rank functions in the context of a theorem prover [27].

4 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.

4.1 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 $s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} s_2 \overset{a_2}{\rightarrow} s_3 \overset{a_3}{\rightarrow} \ldots$ 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) }
```

4.2 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.

4.3 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.

4.4 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
```

4.5 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`).

5 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. 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 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 addition, the PVS theorem prover provides a collection of powerful primitive 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 many similar theorems to be proved efficiently, permitted many proofs to be easily adjusted after modifications in the specification, and helped produce readable proofs.

Using all 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*.

6 Group Key Management : Mathematical Proof

In the previous sections we discussed authentication and leaders agreement. We saw also that once the leaders agree on accepting a user U, they proceed with providing it with a group key. We direct our focus here to Enclaves' Group Key Management module [10]. This module is based on a secret sharing scheme that guarantees (1) the f dishonest leaders cannot obtain the group key even if they conspire and pool their shares together; (2) the group key is renewed every time the group changes (new join or leave); and (3) the users are able to distinguish valid key shares from invalid ones (possibly issued by malicious leaders).

The analysis of the group key management module involves arguments from number theory and probability theory. Given the current state of the art in formal methods, expressing and manipulating notions from those theories is still not possible. As a consequence, we do not analyze the group key management module with the techniques of the previous sections. For completeness though, we give here an overview of the group key management module, as well as a sketch of the manual proofs guaranteeing its security.

The group key management protocol of Enclaves has an architecture similar to that of Cachin *et al.* [5]. The security of the protocol relies on the hardness of the discrete logarithms problem in a group of large prime order. Such a group G_q can be obtained by selecting two large prime numbers p and q such that $p = 2q + 1$. G_q is then chosen as the unique subgroup of order q in $\mathbb{Z}_p^*$. The protocol works as follows. Initially, a dealer chooses a generator g of G_q and a random secret integer $x \in \mathbb{Z}_q$. The dealer then generates n shares $x_1, \cdots, x_n \in \mathbb{Z}_q$ using an (n, f) threshold[2] Shamir secret sharing scheme [28]. The dealer secretly transmits the shares x_i to their corresponding leaders and makes public the values $h_i = g^{x_i} \mod p$, $i \leq n$. We denote by $\tilde{g} = H(G)$ the *hash* of the most recent set of users G, where $H : \{0,1\}^* \rightarrow G_q$. The secret group key to be reconstructed by the users is $\tilde{g}^x$. The protocol parameters p, q and g, as well as H are all known to the participating leaders. Given the above, the protocol works as follows:

1. Leader L_i randomly picks $s \in \mathbb{Z}_q$, and computes $(a, b) = (g^s, \tilde{g}^s) \mod p$.
2. Leader L_i computes $c = H'(y_i, \tilde{g}, a, b)$, where $y_i = \tilde{g}^{x_i} \mod p$, and $H' : {G_q}^4 \rightarrow \mathbb{Z}_q$ is a public hash function.
3. Leader L_i computes $r = s + cx_i \mod q$ and sends each client the tuple (y_i, a, b, r): the share y_i, and (a, b, r) a proof of its validity.
4. Now the client computes $c' = H'(y_i, \tilde{g}, a, b)$, and accepts the share y_i if and only if the following equations hold:

$$g^r \stackrel{?}{=} a\, {h_i}^{c'} \mod p \tag{1}$$
$$\tilde{g}^r \stackrel{?}{=} b\, {y_i}^{c'} \mod p \tag{2}$$

[2]The secret cannot be reconstructed unless $(f + 1)$ valid shares are known.

Let S be any set of $f+1$ (or more) shares y_i that a given client has received. For simplicity, assume $S = \{y_1, y_2, ..., y_{f+1}\}$. We denote by $(a_i)_{1 \le i \le f+1}$ the Lagrange interpolation coefficients[3] leading to $x = \sum_{i=1}^{f+1} a_i x_i$. Given the set of shares S, the clients recover the secret group key as follows:

$$\tilde{g}^x \quad = \quad \tilde{g}^{\left(\sum_{i=1}^{f+1} a_i x_i\right)} \quad = \quad \prod_{i=1}^{f+1} (\tilde{g}^{x_i})^{a_i} \quad = \quad \prod_{i=1}^{f+1} {y_i}^{a_i} \mod p$$

In the following, we present a sketch proof of the module's two main properties, namely, robustness and unpredictability.

Theorem 4 (Robustness) *In the random oracle model, a dishonest leader cannot forge, with non-negligible probability, a valid proof for a non valid share.*

Proof sketch Let y_i be the share provided by leader L_i and (a, b, r) be the corresponding correctness proof. y_i, a, b and r should then satisfy the following equations:

$$g^r \quad = \quad a\, {h_i}^c \mod p \tag{3}$$
$$\tilde{g}^r \quad = \quad b\, {y_i}^c \mod p \tag{4}$$

where $c = H'(y_i, a, b, \tilde{g})$. Equation (3) yields $a \in G_q$, since ${h_i}^c$ and g^r are both in G_q (Closure of G_q under multiplication). The latter implies that it exists $\gamma \in \mathbb{Z}_q$ such that $a = g^\gamma \mod p$. Equation (3) gives: $g^r = g^\gamma g^{cx_i} \mod p$, which implies: $r = \gamma + cx_i \mod q$. Now equation (4) becomes:

$$\begin{aligned} \tilde{g}^r = b\, {y_i}^c \mod p \iff \tilde{g}^{(\gamma + cx_i)} \quad &= \quad b\, {y_i}^c \mod p \\ \iff \tilde{g}^\gamma\, b^{-1} \quad &= \quad (\tilde{g}^{-x_i}\, y_i)^c \mod p \end{aligned}$$

This yields two possible cases:

1. $y_i = \tilde{g}^{x_i} \mod p$. In this case, the share is correct. $b = \tilde{g}^\gamma \mod p$ and for all $c \in \mathbb{Z}_q$ the verifier equations trivially hold.

2. $y_i \neq \tilde{g}^{x_i} \mod p$. In this case, we must have $c = \log_{(\tilde{g}^{-x_i}\, y_i)}(\tilde{g}^\gamma\, b^{-1}) \mod q$.

Once the triplet (y_i, a, b) is chosen, if y_i is not a valid share, then there exists a unique $c \in \mathbb{Z}_q$ that satisfies the verifier equations. In the random oracle model, the hash function H' is assumed to be perfectly random. Therefore, the probability that $H'(y_i, a, b, \tilde{g})$ equals c, once (y_i, a, b) fixed, is $\frac{1}{q}$. If an attacker adaptively queries an oracle $\mathcal{N}$ times, the probability he finds a triplet (y_i, a, b), such that $c = H'(y_i, a, b, \tilde{g})$, is $\mathcal{P}_{Success} = 1 - (1 - \frac{1}{q})^{\mathcal{N}} \approx \frac{\mathcal{N}}{q}$ for large q and $\mathcal{N}$. If $|q| = k$, then $\mathcal{P}_{Success} \le \frac{\mathcal{N}}{2^k}$, which is negligible assuming $\mathcal{N}$ polynomial in the security parameter k. $\square$

Theorem 5 (Unpredictability) *An attacker that corrupts up to f leaders cannot, with a non-negligible probability, learn the secret group key $\tilde{g}^x$.*

The proof of this theorem is similar to the previous one, but is outside the scope of this paper. Its correctness relies on the security of the secret sharing scheme, and the hardness of the Decision Diffie-Hellman problem.

[3]The a_i's depend only on the leaders indexes, and are hence public.

7 Related Work

Much work has been done to formally verify fault-tolerance in distributed protocols. Some of these verifications deal with the Byzantine failure model [6], while others remain limited to the benign form [14]. A variety of automata formalisms has been adopted to specify such protocols.

Castro and Liskov [6] specified their Byzantine fault-tolerant replication algorithm using the I/O automata of Tuttle and Lynch [20]. They have manually proved their algorithm's safety, but not its liveness, using invariant assertions and simulation relations. This work, although similar to our Byzantine agreement module, has never been mechanized in any theorem prover.

Kwiatkowska and Norman [18] analyzed the Asynchronous Binary Byzantine Agreement [5] (based on a concept similar 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.

Timed automata were also used to model the fault-tolerant protocols PAXOS [26] and Ensemble [13]. The authors assume a partially synchronous network and support only benign failures. This bears some similarities with our Enclaves verification in the sense that we assume some bounds on timing, but unlike the work in [26, 13] we are dealing with the more subtle Byzantine kind of failure.

In [2], Archer *et al.* presented the formal verification of some distributed protocols using the Timed Automata Modeling Environment (TAME). TAME provides a set of theory templates to specify and prove I/O automata similar to those we use in our specification.

In [11], Paulson *et al.* extend their inductive approach [25] to cope with the so-called *second-level* security protocols and illustrate their method on a two-layer certified email delivery protocol. Our approach is similar to theirs in principle, except that we do not use Paulson's induction, and our protocols are more complex than theirs.

8 Conclusion and Future Work

This paper describes our results about the formal verification of an Intrusion-Tolerant group-membership protocol. We experimented with a combination of techniques, namely model checking with Murphi, theorem proving with PVS, and we manually conducted a mathematical using the random oracle model. 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 to have achieved a promising success in verifying a complex protocol such as Enclaves, our results could be improved further. 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 the protocols. This can be improved by the use of rank functions [27]. The role of a rank function will be to partition the message space into messages that the intruder might be able to intercept or infer (positive rank), and messages that will certainly remain out of his reach (non-positive rank). The verification consists then in finding if some secret information, with a non-positive rank, can be leaked during the protocol execution and results in a message with a non-positive rank in the intruder's knowledge supposed to be of positive rank. We believe that using rank functions is a very efficient way to mechanically prove authentication properties and we are considering it among our future work plans.

Thanks to the high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, we succeeded to 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. This is also among our 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., probabilities) not yet available in PVS. The current specification can be further extended by widening the Byzantine faults capabilities and by introducing the joint cryptographic modules that have been abstracted away. Also results about an upper bound on Agreement establishment delays can be further investigated.

Acknowledgments

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

References

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

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

[3] D. Boneh. The decision diffie-hellman problem. In *Third International Symposium on Algorithmic Number Theory*, volume 1423 of Lecture Notes in Computer Science, pages 48–63. Springer Verlag, 1998.

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

[5] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. In *Proceedings of the 19th Annual Symposium on Principles of Distribtued Computing*, Portland, Oregon, USA, July 2000. ACM Press.

[6] 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.

[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. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[9] D. Dill, A. Drexler, A. J. 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, October 1992. IEEE Computer Society.

[10] B. Dutertre, V. Crettaz, and V. Stavridou. Intrusion-Tolerant Enclaves. In *Proceedings of the IEEE International Symposium on Security and Privacy*, Oakland, California, USA, May 2002. IEEE Computer Society.

[11] C. Longo G. Bella and L. C. Paulson. Verifying second-level security protocols. In *16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of Lecture Notes in Computer Science, pages 352–366. Springer Verlag, 2003.

[12] T.A. 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, June 1992. IEEE Computer Society.

[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*, volume 1579 of Lecture Notes in Copmuter Science, pages 119–133. Springer Verlag, 1999.

[14] A.J. Hu, R. Li, X. Shi, and S.T. Vuong. Model-Checking a Secure Group Communication Protocol: A Case Study. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, pages 5–8, Beijing, China, October 1999. Kluwer Academics.

[15] C.N. 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, Ontario, Canada, April 1993. North-Holland.

[16] C.N. Ip and D. Dill. Efficient Verification of Symmetric Concurrent Systems. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, Cambridge, Maryland, USA, October 1993. IEEE Computer Society.

[17] C.N. Ip and D. Dill. Verifying Systems with Replicated Components in Murphi. In *Computer-Aided Verification*, volume 1102 of Lecture Notes in Computer Science, pages 147–158. Springer Verlag, 1996.

[18] M. Kwiatkowska and G. Norman. Verifying Randomized Byzantine Agreement. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of Lecture Notes in Computer Science, pages 194–209. Springer Verlag, 2002.

[19] P. Lincoln and J.M. Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

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

[21] N. Lynch and F. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[22] J.C. 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, May 1997. IEEE Computer Society.

[23] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Automated Deduction*, volume 607 of Lecture Notes in Computer Science, pages 748–752. Springer Verlag, 1992.

[24] 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, July 1995. ACM Press.

[25] L.C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[26] R.D. Prisco, B.W. Lampson, and N.A. Lynch. Revisiting the PAXOS Algorithm. In *Distributed Algorithms*, volume 1320 of Lecture Notes in Computer Science, pages 111–125. Springer Verlag, 1997.

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

[28] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22:612–613, 1979.