On the Correctness of an Intrusion-Tolerant Group Communication 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] Computing Science Department
University of Nijmegen, Nijmegen, The Netherlands
hooman@cs.kun.nl

Abstract. Intrusion-tolerance is the technique of using fault-tolerance to achieve security properties. Assuming that faults, both benign and Byzantine, are unavoidable, the main goal of Intrusion-tolerance is to preserve an acceptable, though possibly degraded, service of the overall system despite intrusions at some of its sub-parts. In this paper, we present a correctness proof of the Intrusion-tolerant Enclaves protocol [1] via an adaptive combination of techniques, namely model checking, theorem proving and analytical mathematics. We use Murphi to verify authentication, then PVS to formally specify and prove proper Byzantine Agreement, Agreement Termination and Integrity, and finally we mathematically prove robustness of the group key management module.

1 Introduction

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 key security properties ranging from *confidentiality* and *authentication* to *atomic transactions* and *non-repudiation* [2, 3]. Nevertheless, all the focus was either on two-party protocols (i.e., involving only a pair of users) or, in the best cases, on group protocols with centralized leadership (i.e., a presumably trusted fault-free server managing a group of users). In the present work, we are concerned with the verification of the intrusion-tolerant Enclaves [1]: a group-membership protocol with a distributed leadership architecture, where the authority of the traditional single server is shared among a set of n independent elementary servers, 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 [20] for a survey). Therefore, the question of whether or not a protocol actually achieves its security goals becomes paramount. To date, most of the research in protocol analysis has been devoted to finding attacks on known, either two-party or centralized protocols. In this paper we are concerned with the verification of a distributed multi-leader group communication protocol.

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 [7, 9, 10] 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 [15]. 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 PVS [21] and formalize the protocol in the style of Timed-Automata [5]. 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. 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 [19].

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 we 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 our 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 [1] is a protocol that enables users to share information and collaborate securely through insecure networks such as the Internet. Enclaves provides services for building and managing groups of users. Access to a given group is granted only to sets of users who have the right credentials to do so. Authorized users can dynamically, and at their will, join, leave, and rejoin, an active group. The group communication service relies on a secure multicasting channel that ensures integrity and confidentiality of group communication. All messages sent by a group member are encrypted and delivered to all 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.

In summary, Enclaves should guarantee the following properties, even in the presence of up to f corrupted 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.

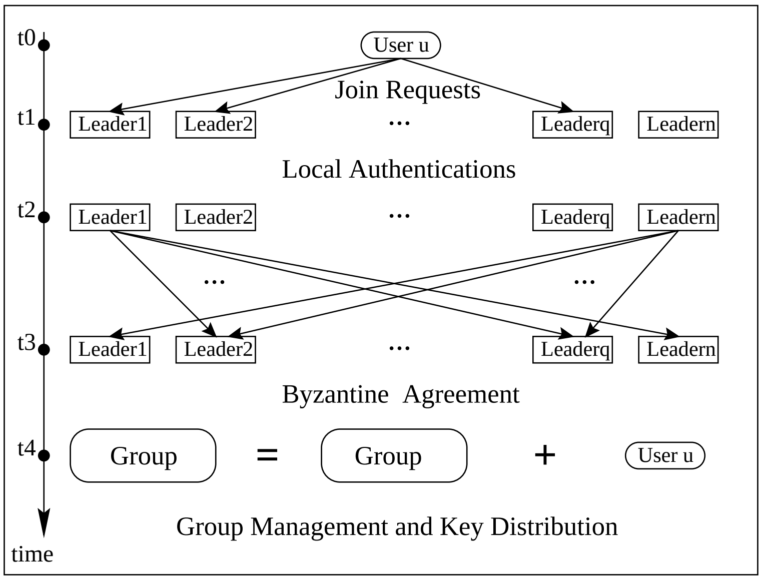

Fig. 1. Enclaves protocol execution

The description of Enclaves in [1] assumes a reliable network where messages eventually reach their destinations within an upper bound delivery time. In this paper we make the same assumptions. Concerning the intruder, we adopt a standard model where an intruder fully monitors the network, proactively augments its knowledge, and chooses to send, either adaptively or randomly, messages on the network. The intruder, however, cannot block messages from reaching their destination and is limited by cryptographic constraints. For instance, the intruder cannot decrypt messages without having the right key, or impersonating other participants by forging cryptographic signatures. 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 2), and (2) the proofs of Proper Agreement, Agreement Termination and Agreement Integrity theorems in PVS (cf. Sections 3 and 4). In addition, we prove the *Confidentiality 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 has a language that supports scalable models. In a scalable model one typically starts with a small protocol configuration and gradually increases the protocol size. In many cases, errors in the general protocol (possibly infinite state) will also show up in down-scaled (finite state) version 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 [16, 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 prints 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.

The main property we are concerned about in this paper is *mutual authentication* between a given pair of leader and client. More precisely, at the end of a protocol execution between a leader L_i and a client C, L_i should be able to assert that it has been talking, indeed, to client C, and vice-versa. The verification has been done by means of invariant checking under the above mentioned assumptions. The *client proper authentication* invariant is given below. It basically states that for each leader i, if it committed to a session with a client, 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).

```
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;
```

In addition to the above invariant, we have checked a similar one for *leaders proper authentication* (i.e., the clients are sure about the identity of the leaders they are communicating with). Table 1 shows the number of reached states and CPU run times taken on a 440 Mhz Sparc machine with 256 MB of memory for different sizes of the protocol. The instances we consider, have been chosen to emphasize the weight of each size parameter. For example, the intruder is modeled to be very powerful (intercepts, replays, and generates messages), so adding a second intruder does not increase the intrusion power, it just multiplies the complexity. Also, the last row in Table 1, shows a non conclusive result, where Murphi runs out of memory before reaching all possible states.

Number of					
Clients	Leaders	Intruders	Network size	States	CPU time
2	4	1	1	4591	13.25 s
2	4	1	3	125793	331.00 s
1	4	2	3	277176	1481.35 s
4	10	1	3	797000	–

Table 1. Model checking experimental results

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

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 $s_0 \stackrel{a_0}{\to} s_1 \stackrel{a_1}{\to} s_2 \stackrel{a_2}{\to} s_3 \stackrel{a_3}{\to} \ldots$ where the s_i are *states*, representing a snapshot of the system during execution and the a_i are the executed *actions*. A particular protocol (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[1].

[1] For more details about the PVS theories and proofs, we refer the reader to the web page: http://hvg.ece.concordia.ca/Research/CRYPTO/Enclaves.html

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.

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

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

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 become 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 briefly outline proofs of the above theorems.

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. □

The above proofs were conducted successfully in PVS and required over 40 lemmas. *Integrity* and *Termination* 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 client C, they proceed with providing it with a group key. We direct our focus here to the Enclaves group key management module [1]. This module is based on a secret sharing scheme which ensures that (1) the f dishonest leaders cannot obtain the group key even if they conspire altogether (at least $(f + 1)$ shares are needed to reconstruct the secret); (2) the group key is renewed every time the group changes (new join or leave); and (3) the clients are able to discern valid key shares from fake ones (possibly issued by malicious leaders).

The group key management protocol of Enclaves is based on previous work of Cachin *et al.* [19]. The security property of the protocol relies on the hardness of computing discrete logarithms in a group of large prime order. Such a group G_q can be constructed by selecting two large prime numbers p and q such that $p = 2q + 1$ and defining G_q as the unique subgroup of order q in $\mathbb{Z}_p{}^*$. The protocol works as follows. Initially, we assume that 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 f-threshold [1] Shamir's secret sharing scheme [18]. The dealer secretly transmits the shares x_i to their corresponding leaders and makes public $h_i = g^{x_i}$ for all leaders $\{L_i\}_{i \leq n}$. We denote by $\tilde{g} = H(G)$ the output of a *hash* function H applied to the most recent set of clients forming the group G. In this scheme, the secret group key to be reconstructed by the clients is $\tilde{g}^x$.
In addition to p, q and g, we assume that H is also known to all the participating leaders. Given the above assumptions, the protocol works as follows:

1. Leader L_i picks randomly $s \in \mathbb{Z}_q$ and computes $(a, b) = (g^s, \tilde{g}^s)$.

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

2. Leader L_i, then, computes $c = H'(y_i, \tilde{g}, a, b)$, where $y_i = \tilde{g}^{x_i}$, and with $H' : {G_q}^4 \rightarrow \mathbb{Z}_q$ a public hash function.
3. Now leader L_i computes $r = s + cx_i$ and sends each client the quadruple (y_i, a, b, r), that is the share y_i and the proof of validity (a, b, r).
4. Now the client computes $c' = H'(y_i, \tilde{g}, a, b)$, supposed to be equal to c, and accepts the share y_i only if the following equations hold:

$$g^r \stackrel{?}{=} a\ {h_i}^{c'} \tag{1}$$

$$\tilde{g}^r \stackrel{?}{=} b\ {y_i}^{c'} \tag{2}$$

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 $(a_i)_{1 \leq i \leq f+1}$ the Lagrange interpolation coefficients[2], such that $\sum_{i=1}^{f+1} a_i x_i = x$, where $a_i = \prod\limits_{j \neq i} \frac{j}{j-i}$.
Given the above shares, the clients recover the secret group key as follows:

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

6.1 Security Analysis : Manual Proof

We sketch proofs of two key properties, namely, robustness and unpredictability.

Theorem 4 (Robustness) *In the random oracle model [3], a dishonest leader cannot forge, with a 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:

$$g^r = a\ {h_i}^c \tag{3}$$

$$\tilde{g}^r = b\ {y_i}^c \tag{4}$$

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 $a = g^\gamma$. Equation (3) gives: $g^r = g^\gamma g^{cx_i}$, which implies: $r = \gamma + cx_i$. Now equation (4) becomes:

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

This yields two possible cases:

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

[2] The a_i depend only on the leaders indexes and hence are publicly known.

[3] In this model, the hash function can be seen as an oracle producing a random value at each query. If the same query is asked twice, an identical answer is given [19].

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

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 $\frac{1}{q}$. On the other hand, if the attacker performs an adaptively chosen message attack by querying an oracle $\mathcal{N}$ times, the probability for the attacker to find 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 $\mathcal{N}$. Now if k is the number of bits in the binary representation of q, then $\mathcal{P}_{Success} \leq \frac{\mathcal{N}}{2^k}$. Since a computationally bounded leader can only try a polynomial number of triplets, then when k is large, the probability of success is negligible $(\mathcal{P}_{Success} = \frac{\mathcal{N}}{2^k} \ll 1)$. $\square$

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

This has been proved by Cachin *et al.* [19] and relies on both:

- The perfect cryptography assumption (i.e., conditional entropy is no greater than simple entropy)

$$S(y_{i_{f+1}} \mid y_{i_1}, y_{i_2}, \cdots, y_{i_j}) = S(y_{i_{f+1}}) \text{ for all } j \leq f$$

- The Computational Diffie-Hellman assumption [22], which states that there is no polynomial time probabilistic algorithm that computes $y_i = \tilde{g}^{x_i}$ given g, $\tilde{g}$, and $h_i = g^{x_i}$, with a non-negligible probability of error.

As a result, the knowledge of up to f shares does not help the attacker to predict any extra valid shares. Therefore, the data to which an attacker might have access is not sufficient to reconstruct the group key with a non-negligible probability of error.

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 [7], while others remain limited to the benign form [8]. A variety of automata formalisms has been adopted to specify such protocols.

Castro and Liskov [7] specified their Byzantine fault-tolerant replication algorithm using the I/O automata of Tuttle and Lynch [6]. 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 [9] analyzed the Asynchronous Binary Byzantine Agreement [19] (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 [11] and Ensemble [14]. 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 [11, 14] we are dealing with the more subtle Byzantine kind of failure.

In [13], 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.

8 Conclusion and Future Work

This paper reports results about the formal verification of an Intrusion-Tolerant protocol. We experimented with an adaptively chosen combination of techniques based on the nature of the correctness arguments in each module of the protocol, the environment assumptions and the easiness of performing verification.

We believe to have achieved a promising success in verifying a complex protocol such as Enclaves. Nevertheless, our results could be improved further in various aspects. 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 [2]. 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 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 layers 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. B. Dutertre, V. Crettaz and V. Stavridou. Intrusion-Tolerant Enclaves. In Proc. IEEE Intl. Symposium on Security and Privacy, pp. 216-226, 2002.
2. P. Ryan and S. Schneider. The Modelling and Analysis of Security Protocols: the CSP Approach. Addison-Wesley, 2000.
3. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. Journal of Computer Security, 6:85-128, 1998.
4. G. Bracha and S. Toueg. Resilient consensus protocols. In Proc. ACM Symposium on Principles of Distributed Computing, pp.12-26, 1983.
5. R. Alur and D. L. Dill. A Theory of Timed Automata. In Theoretical Computer Science 126:183-235, 1994.
6. N. Lynch and M. Tuttle. An Introduction to Input/Output automata. In Centrum voor Wiskunde en Informatica Quarterly Journal, 2(3):219-246, 1989.
7. M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Tech. Memo MIT/LCS/TM-590, 1999.
8. A. J. Hu, R. Li, X. Shi and S. T. Vuong: Model-Checking a Secure Group Communication Protocol: A Case Study. In Proc. FORTE/PSTV, pp. 469-478, 1999.
9. M. Kwiatkowska and G. Norman. Verifying Randomized Byzantine Agreement. In Proc. FORTE (LNCS 2529), pp. 194-209, 2002.
10. P. Lincoln and J. Rushby. A Formally Verified Algorithm for Interactive Consistency under a Hybrid Fault Model. In Proc. Fault Tolerant Computing Symposium, pp. 304-313, 1993.
11. R. De Prisco, B. Lampson, N. Lynch. Revisiting the PAXOS Algorithm. In Proc. Intl. Workshop on Distributed Algorithms, (LNCS 1320), pp. 111-125, 1997.
12. N. Lynch and F. Vaandrager. Action Transducers and Timed Automata. In Formal Aspects of Computing, 8(5):499-538, 1996.
13. M. Archer, C. Heitmeyer and E. Riccobene. Proving Invariants of I/O Automata with TAME. In Automated Software Engineering, 9(3): 201-232, 2002.
14. J. Hickey, N. Lynch, and R. van Renesse. Specifications and Proofs for Ensemble Layers. In Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, (LNCS 1579), pp. 119-133, 1999.
15. D. L. Dill, A. J. Drexler, A. J. Hu and C. H. Yang. Protocol Verification as a Hardware Design Aid. In Proc. Intl. Conf. on Computer Design, pp. 522-525, 1992.
16. C. Norris Ip and D. L. Dill. Better Verification through Symmetry. In Proc. Intl. Conf. on Computer Hardware Description Languages, pp. 87-100, 1993.
17. C. Norris Ip and D. L. Dill. Verifying Systems with Replicated Components in Murphi. In Proc. of the Intl. Conf. on Computer-Aided Verification (LNCS 1102), pp. 147-158, 1996.
18. A. Shamir. How to Share a Secret. In Comm. of the ACM, 22:612-613, 1979.
19. C. Cachin, K. Kursawe and V. Shoup. Random Oracles in Constantipole: Practical Asynchronous Byzantine Agreement Using Cryptography. In Proc. of the ACM Symposium on Principles of Distributed Computing, pp. 123-132, 2000.
20. J. Clark and J. Jacob. A Survey of Authentication Protocols Literature: Version 1.0. Draft paper available at http://www-users.cs.york.ac.uk/~jac
21. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Proc. Intl. Conf. on Automated Deduction, (LNCS 607), pp. 748-752, 1992.
22. D. Boneh. The decision Diffie-Hellman problem. In Proc. of the Third Algorithmic Number Theory Symposium, (LNCS 1423), pp. 48–63, 1998.